THE EXPERT'S VOICE®

SECOND EDITION

# Pro Git

*EVERYTHING YOU NEED TO KNOW ABOUT GIT*

Scott Chacon and Ben Straub

Apress®

Pro Git

# Table of Contents

# Pro Git

# Scott Chaconによる序文

Pro Git第2版へようこそ。 本書の第1版が出版されたのは、今から4年以上前のことでした。 あれからいろいろ変わったこともあれば、変わっていないこともあります。 基本的なコマンドや考えかたは、第1版当時から何も変わっていないでしょう。というのも、Gitコアチームは過去との互換性をきちんと守っているからです。 でも、大きな機能追加もあれば、Gitを取り巻くコミュニティにも変化がありました。 そのあたりに対応するするために作ったのが、この第2版です。新しいユーザーにとっても、役立つことでしょう。

第1版を書いていたころのGitは、決して使いやすいとは言えず、筋金入りのハッカーたちだけにしか受け入れられていませんでした。 いくつかのコミュニティが活発に動き出しつつあったものの、今のように広く普及するには至らなかったのです。 今や、ほとんどのオープンソースコミュニティが、Gitに移行しています。 Windows版のGitもまともに動くようになり、グラフィカルなユーザーインターフェイスも急増し、 IDEでの対応や業務での利用も増えてきました。 4年前にPro Gitを執筆していたころには思いもよらなかったことです。 第2版の主な狙いのひとつは、Gitコミュニティにおけるこうした新たな動きについてとりあげることでした。

オープンソースコミュニティにおけるGitの採用も、急増しています。 第1版の執筆に取りかかった約5年前（第1版を書き上げるのには時間がかかったのです…）、 私はほぼ無名に等しい企業で働き始めました。その会社では、GitHubという名前のGitホスティングWebサイトを作っていました。 第1版を出版した当時、GitHubのユーザーはたかだか数千人程度で、社員は4人だけでした。 今この序文を書いている時点で、GitHub上のプロジェクト数は1000万を突破しています。 登録ユーザー数は500万をこえて、社員の数も230人になりました。 その善し悪しは別として、GitHubはオープンソースコミュニティを大きく変えてしまったのです。 第1版を書き始めたころには、まさかそんなことになるとは思ってもいませんでした。

Pro Gitの第1版におけるGitHubの扱いは、Gitのホスティングサイトの一例として簡単に紹介した程度でした。個人的に、あまり気分のいいものではありませんでした。 私はコミュニティのリソースについて書こうとしていたのですが、そこで自分の勤務先のことについても語るのは、居心地が悪いものだったのです。 その考えは今でも変わりませんが、Gitコミュニティにおいて、GitHubの存在はもはや無視できないレベルになっています。 そこで、Gitのホスティングの一例として紹介するのではなく、 GitHubについてのより詳しい紹介と、そのうまい使いかたを、もう少し深く掘り下げて説明することにしました。 Gitの使いかたを覚えた上で、GitHubの使いかたを身につければ、大規模なコミュニティに参加するときにも役立つでしょう。 そのコミュニティがGitHubを使っているかどうかにかかわらず、その知識は役立ちます。

第1版以降の、もうひとつの大きな変化は、GitのネットワークトランザクションにHTTPを使うことが増えてきたことでしょう。 本書の例の大半を、SSHを使ったものからHTTPを使うものに書き換えました。そのほうが、ずっとシンプルになるからです。

かつては無名のバージョン管理システムであったGitが、今や商用製品を含めたバージョン管理システム界を制覇するようになるとは、まさに驚くべきことです。 Pro Gitがこれまでうまくやってこられたことに満足しています。また、オープンソースで作られており、かつ成功しているという技術書の一員になれたことを、ありがたく思います。

今回の新版も、ぜひお楽しみください。

# Ben Straubによる序文

私がGitにはまるきっかけになったのが、本書の第1版でした。 それまでに経験したことのない、より自然な感覚でソフトウェアを作れるようなスタイルを、教えてもらいました。 それ以前にも開発者として数年の経験はあったのですが、第1版が私の転機になりました。 今よりもずっとおもしろい道があることを知ったのです。

あれから何年かたった今、私はGitの実装にも貢献するようになりました。 世界最大のGitホスティングサービスを運営する企業で働き、Gitについて教えるために世界中を飛び回っています。 Scottから第2版の執筆の話を持ちかけられたときは、考えるまでもなくイエスと答えました。

本書の執筆にかかわれて、うれしく思います。かつての私がそうであったように、本書が皆さんの助けになれば幸いです。

# 謝辞

*妻のBeckyへ。君がいなければ、とてもこんな冒険はできなかったよ。 — Ben*

*この第2版を、妻と娘に捧げる。 妻のJessicaは、これまでずっと私を支えてくれた。 そして娘のJosephineは、年老いていく私を支えてくれることだろう。 — Scott*

# 使い始める

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

## バージョン管理に関して

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

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

### ローカル・バージョン管理システム

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

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

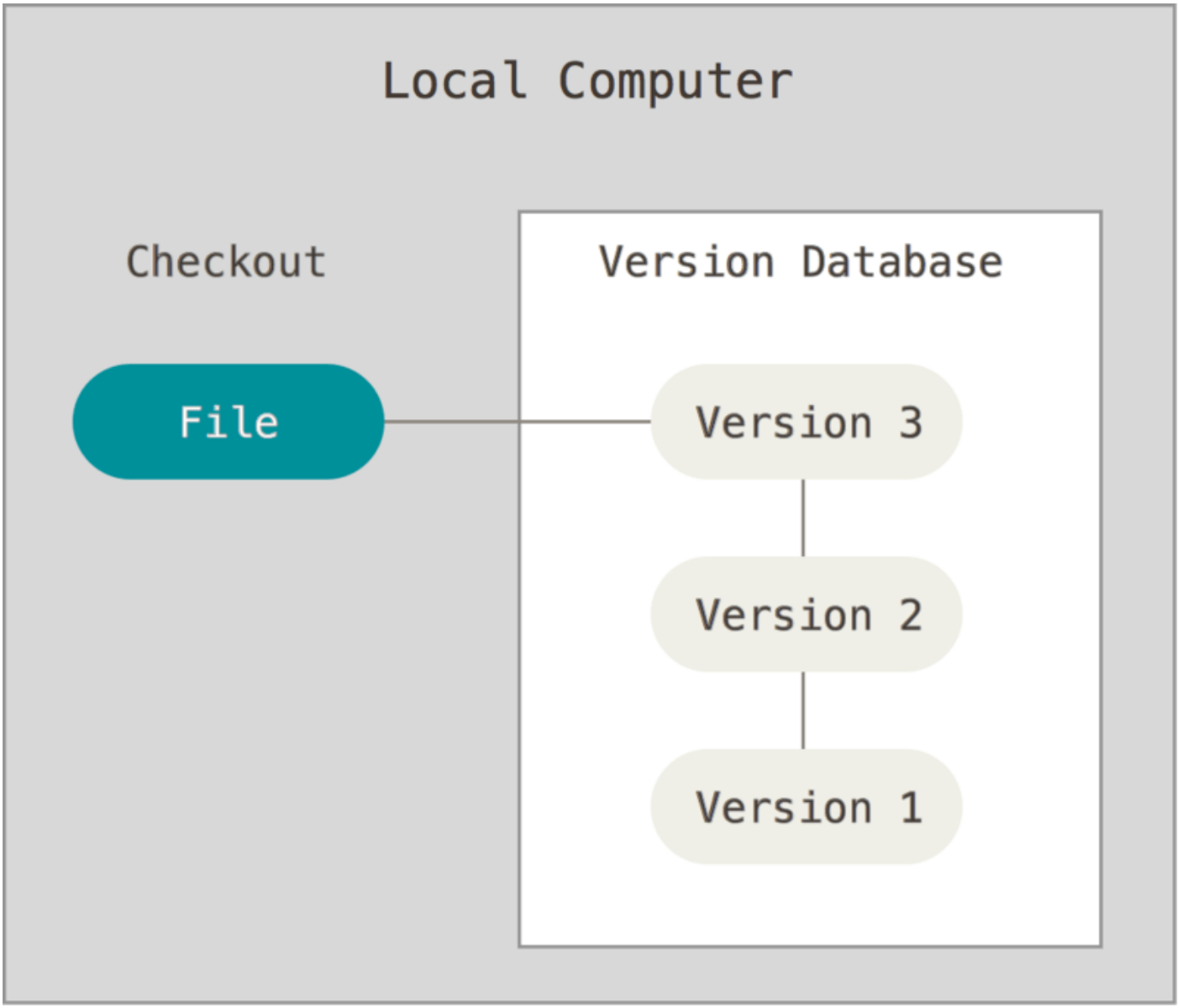



*Figure 1. ローカル・バージョン管理図解*

もっとも有名なVCSツールの一つは、RCSと呼ばれるシステムでした。今日でも、依然として多くのコンピューターに入っています。 人気のMac OS Xオペレーティング・システムでも、開発者ツールをインストールすると`rcs`コマンドが入っています。 このツールは基本的に、リビジョン間のパッチ（ファイル間の差分）の集合を特殊なフォーマットでディスク上に保持するという仕組みで動いています。こうすることで、任意のファイルについて、それが過去の任意の時点でどういうものだったかということを、パッチを重ね上げていくことで再現することができます。

## 集中バージョン管理システム

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

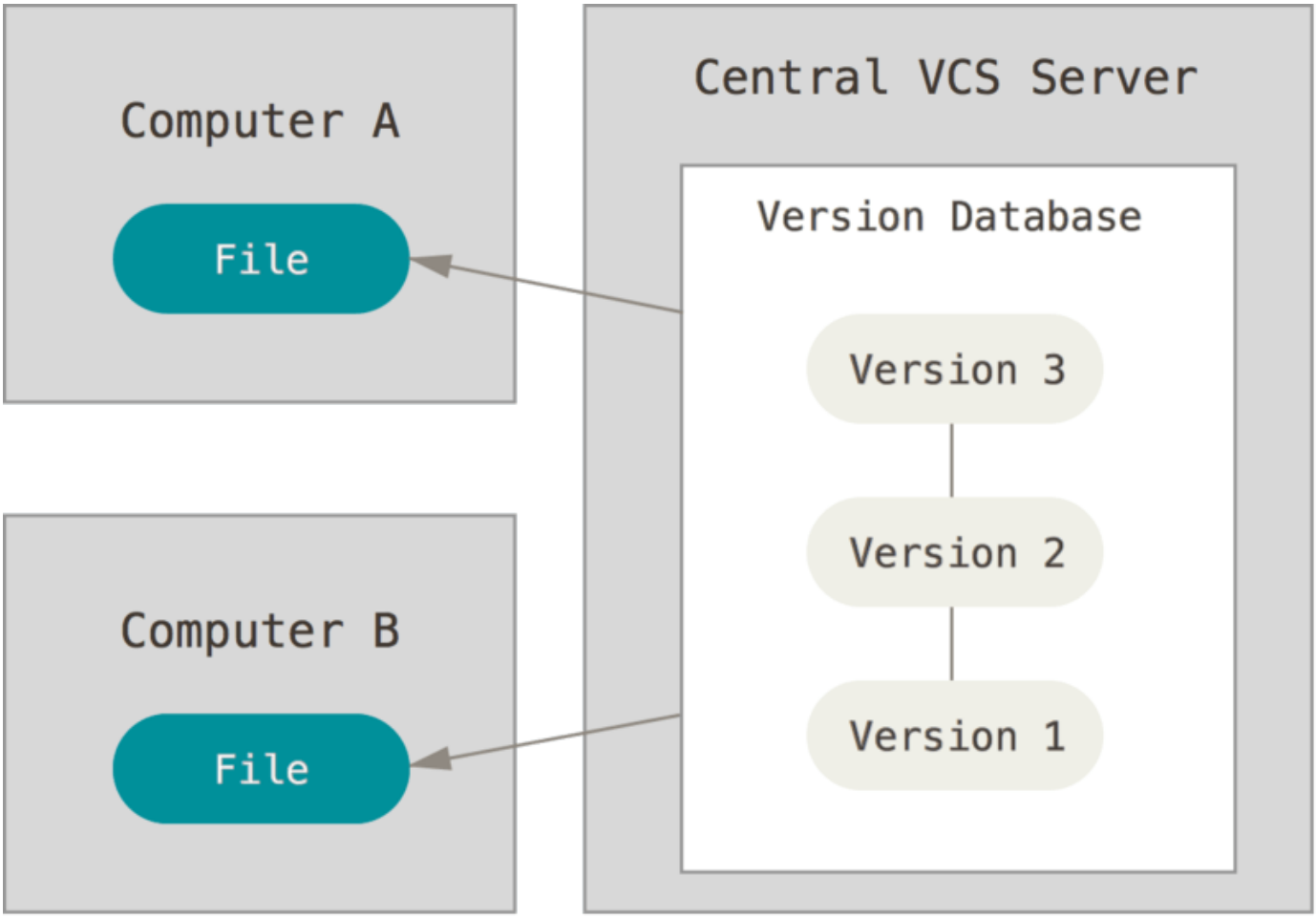



*Figure 2. 集中バージョン管理図解*

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

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

## 分散バージョン管理システム

ここで分散バージョン管理システム(DVCS)の出番になります。 DVCS(Git、Mercurial、Bazaar、Darcsのようなもの)では、クライアントはファイルの最新スナップショットをチェックアウト（訳者注：バージョン管理システムから、作業ディレクトリにファイルやディレクトリをコピーすること）するだけではありません。リ

ポジトリ（訳者注：バージョン管理の対象になるファイル、ディレクトリ、更新履歴などの一群）全体をミラーリングするのです。 そのため、あるサーバーが故障して、DVCSがそのサーバーを介して連携していたとしても、どれでもいいのでクライアント・リポジトリの一つをサーバーにコピーすれば修復できます。 クローンは全て、実際は全データの完全バックアップなのです。

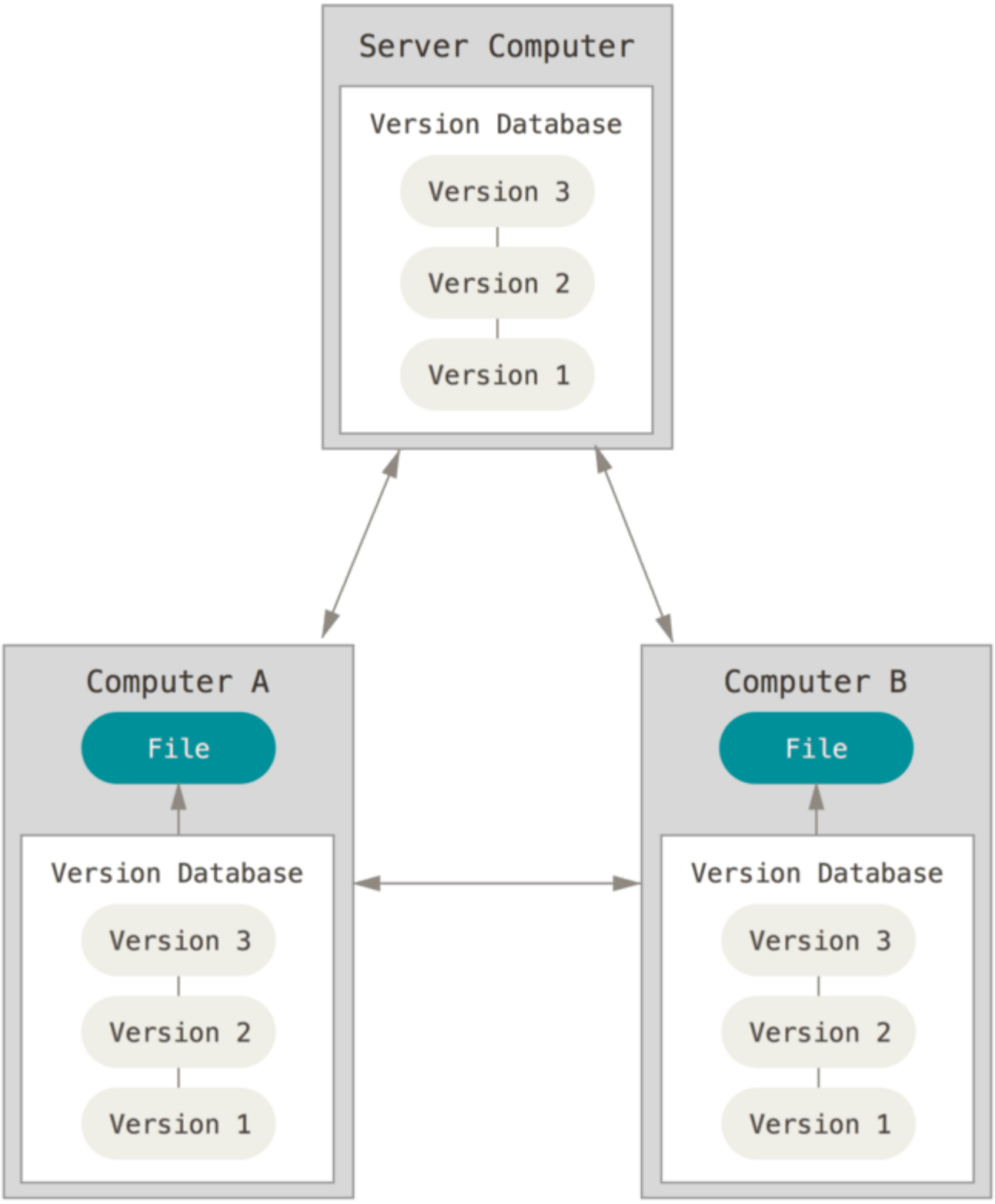



*Figure 3. 分散バージョン管理図解*

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

# Git略史

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

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

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

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

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

# Gitの基本

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

## スナップショットで、差分ではない

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

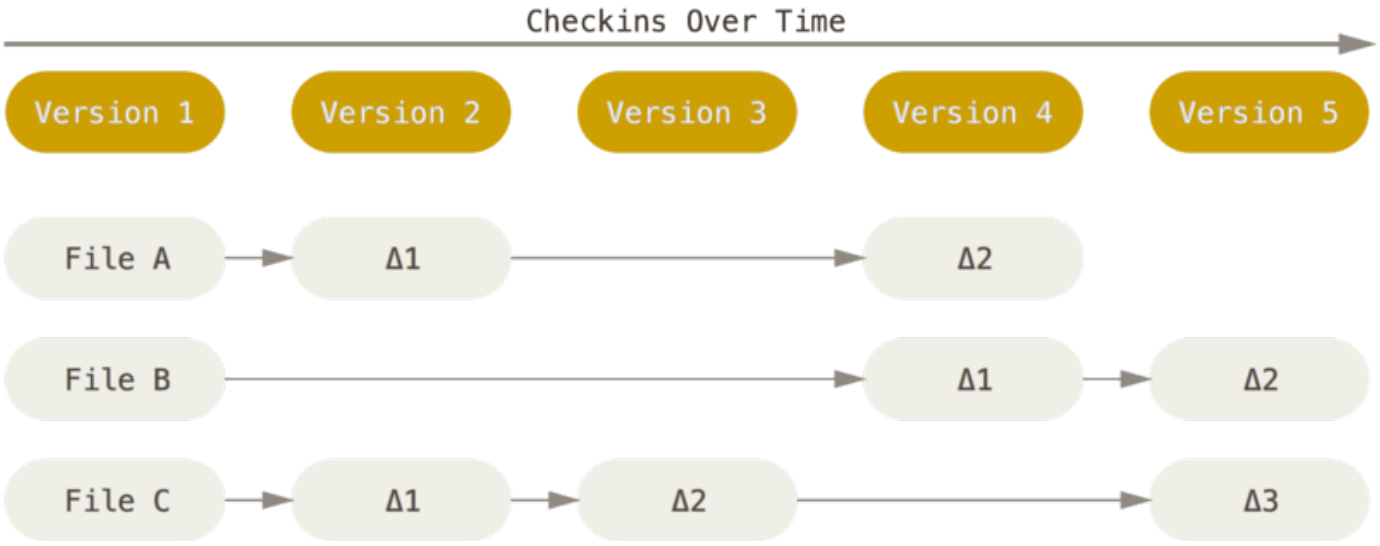



*Figure 4. 他のシステムは、データをそれぞれのファイルの基本バージョンへの変更として格納する傾向があります。*

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

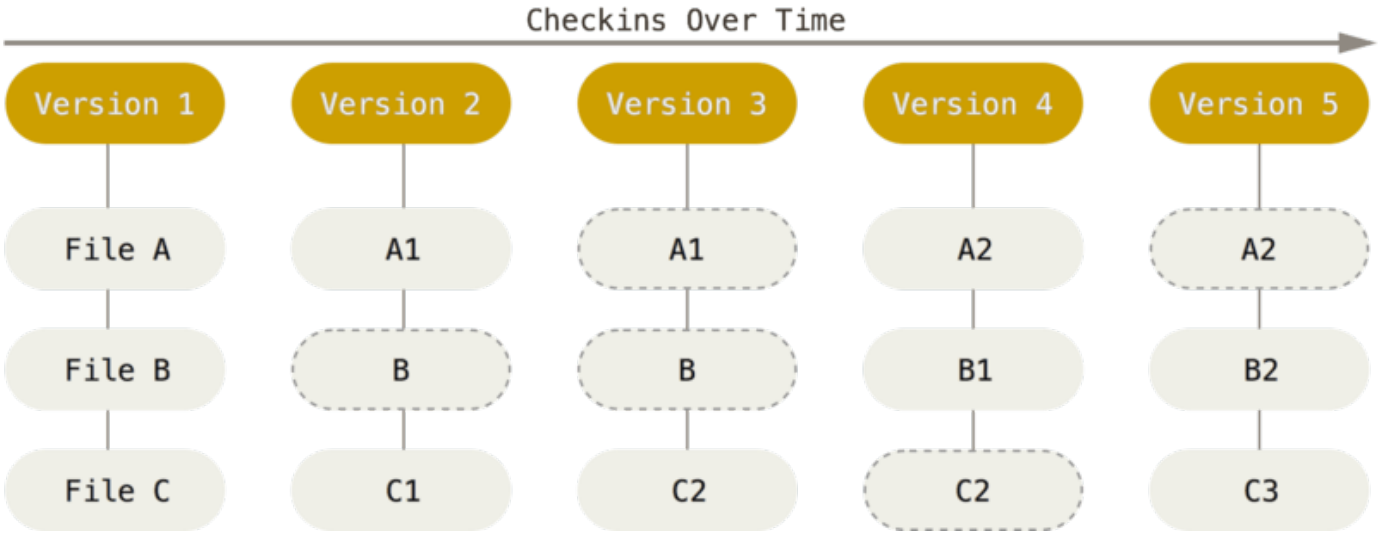



*Figure 5. Gitは時間を通じたプロジェクトのスナップショットとしてデータを格納します。*

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

## ほとんど全ての操作がローカル

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

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

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

## Gitは完全性を持つ

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

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

```
24b9da6552252987aa493b52f8696cd6d3b00373
```

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

## Gitは通常はデータを追加するだけ

Gitで行動するとき、ほとんど全てはGitデータベースにデータを追加するだけです。 システムにいかなる方法でも、UNDO不可能なこと、もしくはデータを消させることをさせるのは困難です。 あらゆるVCSと同様に、まだコミットしていない変更は失ったり、台無しにできたりします。しかし、スナップショットをGitにコミ

ットした後は、特にもし定期的にデータベースを他のリポジトリにプッシュ（訳注：pushはGitで管理するあるリポジトリのデータを、他のリポジトリに転送する操作。詳細は後の章を参照）していれば、変更を失うことは大変難しくなります。

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

## 三つの状態

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

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

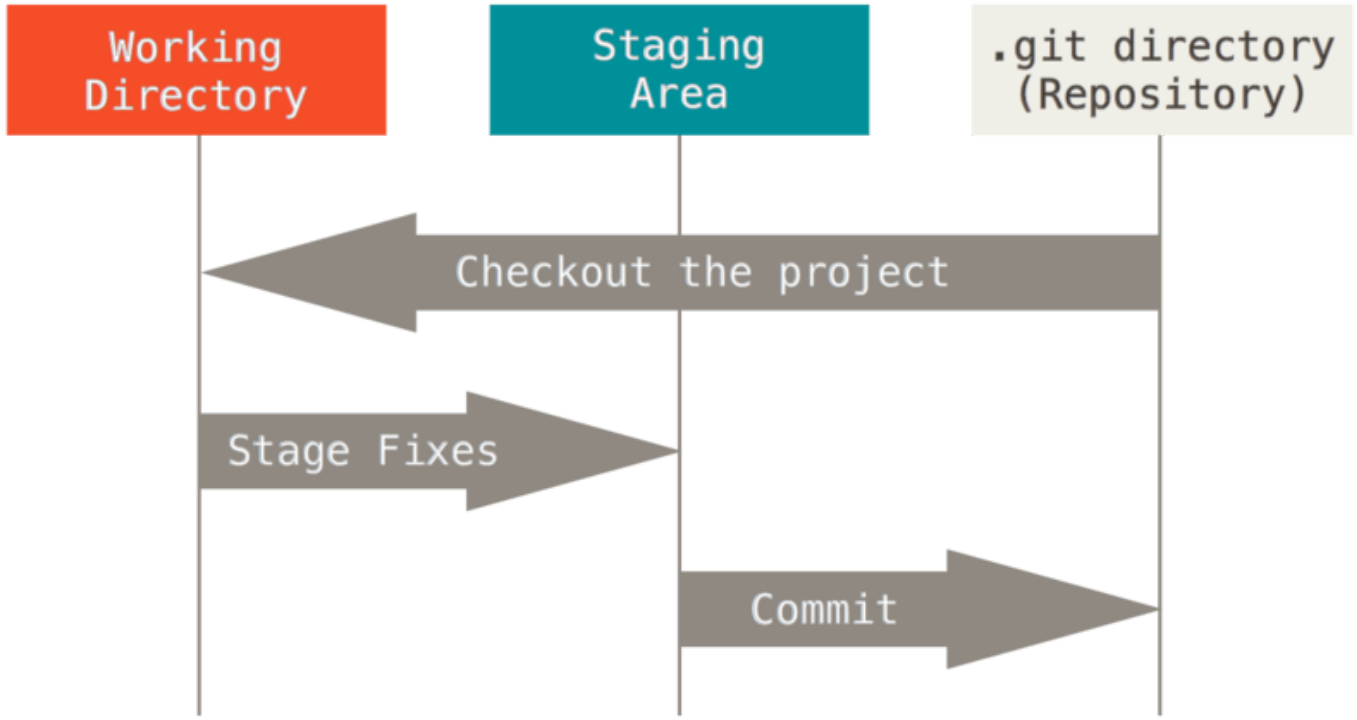



*Figure 6. 作業ディレクトリ、ステージング・エリア、Gitディレクトリ*

Gitディレクトリは、プロジェクトのためのメタデータ（訳者注：Gitが管理するファイルやディレクトリなどのオブジェクトの要約）とオブジェクトのデータベースがあるところです。 これは、Gitの最も重要な部分で、他のコンピューターからリポジトリをクローン（訳者注：コピー元の情報を記録した状態で、Gitリポジトリをコピーすること）したときに、コピーされるものです。

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

す。

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

基本的なGitのワークフローは、このような風に進みます：

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

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

# コマンドライン

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

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

# Gitのインストール

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

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

## Linuxにインストール

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

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

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

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

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

## Macにインストール

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

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


*Figure 7. Git OS Xインストーラー*

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

## Windowsにインストール

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

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

## ソースからのインストール

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

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

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

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

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

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

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

バイナリー名が異なるために生じる問題を解消するためです。

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

ダウンロードが終わったら、コンパイルしてインストールします：

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

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

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

# 最初のGitの構成

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

Gitには、`git config`と呼ばれるツールが付属します。これで、どのようにGitが見えて機能するかの全ての面を制御できる設定変数を取得し、設定することができます。 これらの変数は三つの異なる場所に格納されうります：

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

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

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

## 個人の識別情報

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

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

また、もし`--global`オプションを指定するのであれば、Gitはその後、そのシステム上で行なう（訳者注：あるユーザーの）全ての操作に対して常にこの情報を使うようになるため、この操作を行なう必要はたった一度だけです。 もし、違う名前とEmailアドレスを特定のプロジェクトで上書きしたいのであれば、そのプロジェクトの（訳者注：Gitディレクトリの）中で、`--global`オプション無しでこのコマンドを実行することができます。

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

## エディター

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

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

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

32bit版Windowsの場合

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

64bit版Windowsの場合

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

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

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

### 設定の確認

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

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

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

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

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

## ヘルプを見る

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

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

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

```
$ git help config
```

これらのコマンドは、オフラインのときでさえ、どこでも見る事ができるので、すばらしいです。 も

しmanpageとこの本が十分でなく、人の助けが必要であれば、フリーノードIRCサーバー（irc.freenode.net）の`#git`もしくは`#github`チャンネルにアクセスしてみてください。 これらのチャンネルはいつも、Gitに関してとても知識があり、よく助けてくれようとする数百人の人々でいっぱいです。

# まとめ

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

# Git の基本

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

## Git リポジトリの取得

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

### 既存のディレクトリでのリポジトリの初期化

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

```
$ git init
```

これを実行すると `.git` という名前の新しいサブディレクトリが作られ、リポジトリに必要なすべてのファイル (Git リポジトリのスケルトン) がその中に格納されます。 この時点では、まだプロジェクト内のファイルは一切管理対象になっていません (今作った `.git` ディレクトリに実際のところどんなファイルが含まれているのかについての詳細な情報は、Gitの内側を参照ください)。

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

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

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

### 既存のリポジトリのクローン

既存の Git リポジトリ (何か協力したいと思っているプロジェクトなど) のコピーを取得したい場合に使うコマンドが、`git clone` です。 Subversion などの他の VCS を使っている人なら「"checkout" じゃなくて "clone" なのか」と気になることでしょう。 これは重要な違いです。ワーキングコピーを取得するのではなく、Git はサーバーが保持しているデータをほぼすべてコピーするのです。 そのプロジェクトのすべてのファ

イルのすべての歴史が、デフォルトでは `git clone` で手元にやってきます。
実際、もし仮にサーバーのディスクが壊れてしまったとしても、どこかのクライアントに残っているクローンをサーバーに戻せばクローンした時点まで多くの場合は復元できるでしょう(サーバーサイドのフックなど一部の情報は失われてしまいますが、これまでのバージョン管理履歴はすべてそこに残っています。サーバー用の Git の取得で詳しく説明します)。

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

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

これは、まず``'libgit2'というディレクトリを作成してその中で `.git` ディレクトリを初期化し、リポジトリのすべてのデータを引き出し、そして最新バージョンの作業コピーをチェックアウトします。新しくできた `libgit2` ディレクトリに入ると、プロジェクトのファイルをごらんいただけます。もし``libgit2'ではない別の名前のディレクトリにクローンしたいのなら、コマンドラインオプションでディレクトリ名を指定します。

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

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

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

## 変更内容のリポジトリへの記録

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

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

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

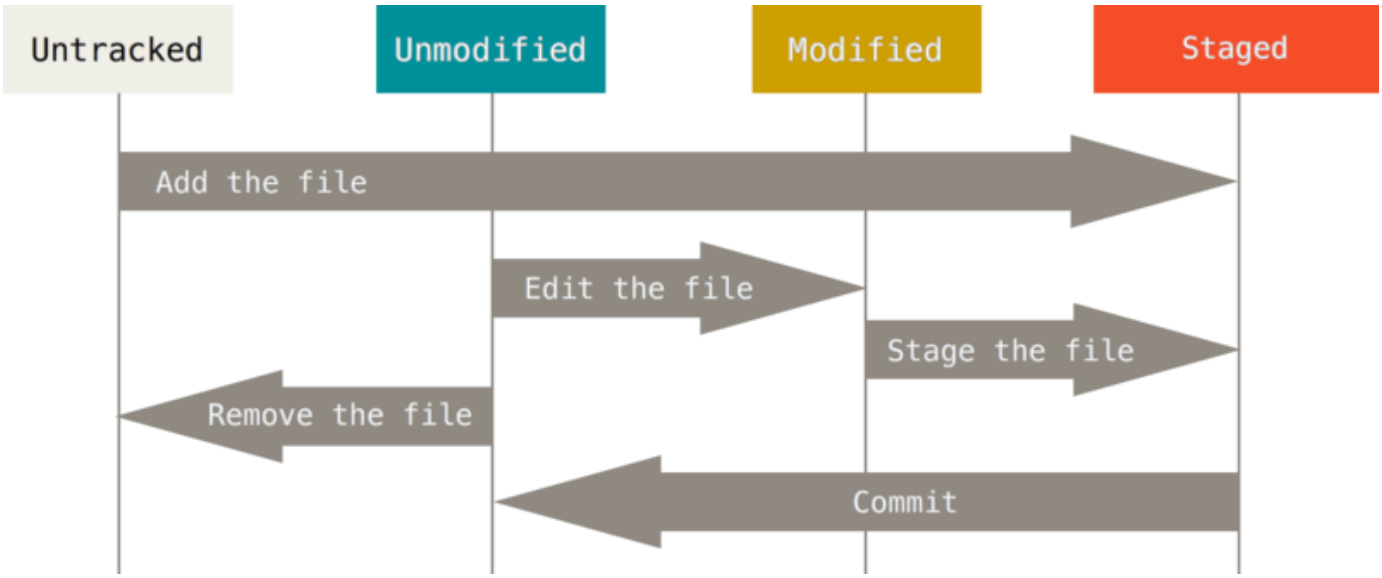



*Figure 8. ファイルの状態の流れ*

## ファイルの状態の確認

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

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

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

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

```
$ echo 'My Project' > README
$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

    README

nothing added to commit but untracked files present (use "git add" to
track)
```

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

### 新しいファイルの追跡

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

```
$ git add README
```

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

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

    new file:   README
```

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

## 変更したファイルのステージング

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

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

    new file:   README

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

    modified:   CONTRIBUTING.md
```

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

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

    new file:   README
    modified:   CONTRIBUTING.md
```

両方のファイルがステージされました。これで、次回のコミットに両方のファイルが含まれるようになります。 ここで、さらに CONTRIBUTING.md にちょっとした変更を加えてからコミットしたくなったとしましょう。 ファイルを開いて変更を終え、コミットの準備が整いました。 しかし、git status を実行してみると何か変です。

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

    new file:   README
    modified:   CONTRIBUTING.md

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

    modified:   CONTRIBUTING.md
```

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

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

    new file:   README
    modified:   CONTRIBUTING.md
```

### 状態表示の簡略化

`git status` の出力はとてもわかりやすいですが、一方で冗長でもあります。 Gitにはそれを簡略化するためのオプションもあり、変更点をより簡潔に確認できます。 `git status -s`や`git status --short`コマンドを実行して、簡略化された状態表示を見てみましょう。

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

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

## ファイルの無視

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

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

最初の行は “.o” あるいは “.a” で終わる名前のファイル (コードをビルドする際にできるであろうオブジェクトファイルとアーカイブファイル) を無視するよう Git に伝えています。次の行で Git に無視させているのは、チルダ (~) で終わる名前のファイルです。Emacs をはじめとする多くのエディタが、この形式の一時ファイルを作成します。これ以外には、たとえば log、tmp、pid といった名前のディレクトリや自動生成されるドキュメントなどもここに含めることになるでしょう。実際に作業を始める前に `.gitignore` ファイルを準備しておくことをお勧めします。そうすれば、予期せぬファイルを間違って Git リポジトリにコミットしてしまう事故を防げます。

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

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

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

では、`.gitignore` ファイルの例をもうひとつ見てみましょう。

```
# no .a files
*.a

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

# only ignore the TODO file in the current directory, not subdir/TODO
/TODO

# ignore all files in the build/ directory
build/

# ignore doc/notes.txt, but not doc/server/arch.txt
doc/*.txt

# ignore all .pdf files in the doc/ directory
doc/**/*.pdf
```

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

## ステージされている変更 / されていない変更の閲覧

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

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

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

    modified:   README

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

    modified:   CONTRIBUTING.md
```

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

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

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

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

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

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

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

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

```
$ git add CONTRIBUTING.md
$ echo '# test line' >> CONTRIBUTING.md
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   CONTRIBUTING.md

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

    modified:   CONTRIBUTING.md
```

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

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

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

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

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

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

**NOTE**

*GitのDiffを他のツールで見る*

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

## 変更のコミット

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

```
$ git commit
```

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

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

```
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
# Changes to be committed:
#   new file:   README
#   modified:   CONTRIBUTING.md
#
~
~
~
".git/COMMIT_EDITMSG" 9L, 283C
```

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

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

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

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

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

## ステージングエリアの省略

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

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

    modified:   CONTRIBUTING.md

no changes added to commit (use "git add" and/or "git commit -a")
$ git commit -a -m 'added new benchmarks'
[master 83e38c7] added new benchmarks
 1 file changed, 5 insertions(+), 0 deletions(-)
```

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

## ファイルの削除

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

単に作業ディレクトリからファイルを削除しただけの場合は、`git status` の出力の中では “Changed but

not updated” (つまり *ステージされていない*) 欄に表示されます。

```
$ rm PROJECTS.md
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working
directory)

        deleted:    PROJECTS.md

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

`git rm` を実行すると、ファイルの削除がステージされます。

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

    deleted:    PROJECTS.md
```

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

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

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

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

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

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

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

このコマンドは、`~` で終わるファイル名のファイルをすべて削除します。

## ファイルの移動

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

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

```
$ git mv file_from file_to
```

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

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

    renamed:    README.md -> README
```

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

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

Git はこれが暗黙的なファイル名の変更であると理解するので、この方法であろうが `mv` コマンドを使おうがどちらでもかまいません。 唯一の違いは、この方法だと 3 つのコマンドが必要になるかわりに `mv` だとひとつのコマンドだけで実行できるという点です。 より重要なのは、ファイル名の変更は何でもお好みのツールで行えるということです。あとでコミットする前に add/rm を指示してやればいいのです。

# コミット履歴の閲覧

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

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

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

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

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

    changed the version number

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

    removed unnecessary test

commit a11bef06a3f659402fe7563abf99ad00de2209e6
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 10:31:28 2008 -0700

    first commit
```

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

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

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

```
$ git log -p -2
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    changed the version number

diff --git a/Rakefile b/Rakefile
index a874b73..8f94139 100644
--- a/Rakefile
+++ b/Rakefile
@@ -5,7 +5,7 @@ require 'rake/gempackagetask'
 spec = Gem::Specification.new do |s|
     s.platform  =   Gem::Platform::RUBY
     s.name      =   "simplegit"
-    s.version   =   "0.1.0"
+    s.version   =   "0.1.1"
     s.author    =   "Scott Chacon"
     s.email     =   "schacon@gee-mail.com"
     s.summary   =   "A simple gem for using Git in Ruby code."

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

    removed unnecessary test

diff --git a/lib/simplegit.rb b/lib/simplegit.rb
index a0a60ae..47c6340 100644
--- a/lib/simplegit.rb
+++ b/lib/simplegit.rb
@@ -18,8 +18,3 @@ class SimpleGit
     end

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

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

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

    changed the version number

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

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

    removed unnecessary test

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

commit a11bef06a3f659402fe7563abf99ad00de2209e6
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 10:31:28 2008 -0700

    first commit

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

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

もうひとつの便利なオプションが `--pretty` です。 これは、ログをデフォルトの書式以外で出力します。 あらかじめ用意されているいくつかのオプションを指定することができます。 `oneline` オプションは、各コミットを一行で出力します。 これは、大量のコミットを見る場合に便利です。 さらに `short` や `full` そして `fuller` といったオプションもあり、 これは標準とほぼ同じ書式だけれども情報量がそれぞれ少なめあるいは多めになります。

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

もっとも興味深いオプションは `format` で、これは独自のログ出力フォーマットを指定することができます。 これは、出力結果を機械にパースさせる際に非常に便利です。 自分でフォーマットを指定しておけば、将来

Git をアップデートしても結果が変わらないようにできるからです。

```
$ git log --pretty=format:"%h - %an, %ar : %s"
ca82a6d - Scott Chacon, 6 years ago : changed the version number
085bb3b - Scott Chacon, 6 years ago : removed unnecessary test
a11bef0 - Scott Chacon, 6 years ago : first commit
```

`git log --pretty=format` 用の便利なオプション は、format で使用できる便利なオプションをまとめたものです。

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

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

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

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

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

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

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

*Table 2. `git log`のよく使われるオプション*

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

## ログ出力の制限

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

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

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

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

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

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

```
$ git log -Sfunction_name
```

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

`git log`の出力を制限するためのオプション に、これらのオプションとその他の一般的なオプションをまとめました。

*Table 3. `git log`の出力を制限するためのオプション*

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

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

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

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

## 作業のやり直し

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

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

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

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

コミットメッセージのエディタが同じように立ち上がりますが、既に前回のコミット時のメッセージが書き込まれた状態になっています。 ふだんと同様にメッセージを編集できますが、前回のコミット時のメッセージがその内容で上書きされます。

たとえば、いったんコミットした後、何かのファイルをステージするのを忘れていたのに気づいたとしましょう。そんな場合はこのようにします。

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

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

## ステージしたファイルの取り消し

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

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

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

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

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

    renamed:    README.md -> README

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

    modified:   CONTRIBUTING.md
```

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

**NOTE**

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

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

## ファイルへの変更の取り消し

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

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

    modified:   CONTRIBUTING.md
```

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

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

    renamed:    README.md -> README
```

変更が取り消されたことがわかります。

**IMPORTANT**

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

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

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

# リモートでの作業

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

## リモートの表示

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

```
$ git clone https://github.com/schacon/ticgit
Cloning into 'ticgit'...
remote: Reusing existing pack: 1857, done.
remote: Total 1857 (delta 0), reused 0 (delta 0)
Receiving objects: 100% (1857/1857), 374.35 KiB | 268.00 KiB/s, done.
Resolving deltas: 100% (772/772), done.
Checking connectivity... done.
$ cd ticgit
$ git remote
origin
```

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

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

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

```
$ cd grit
$ git remote -v
bakkdoor  https://github.com/bakkdoor/grit (fetch)
bakkdoor  https://github.com/bakkdoor/grit (push)
cho45     https://github.com/cho45/grit (fetch)
cho45     https://github.com/cho45/grit (push)
defunkt   https://github.com/defunkt/grit (fetch)
defunkt   https://github.com/defunkt/grit (push)
koke      git://github.com/koke/grit.git (fetch)
koke      git://github.com/koke/grit.git (push)
origin    git@github.com:mojombo/grit.git (fetch)
origin    git@github.com:mojombo/grit.git (push)
```

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

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

### リモートリポジトリの追加

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

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

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

```
$ git fetch pb
remote: Counting objects: 43, done.
remote: Compressing objects: 100% (36/36), done.
remote: Total 43 (delta 10), reused 31 (delta 5)
Unpacking objects: 100% (43/43), done.
From https://github.com/paulboone/ticgit
 * [new branch]      master     -> pb/master
 * [new branch]      ticgit     -> pb/ticgit
```

Paul の master ブランチは、ローカルでは `pb/master` としてアクセスできます。 これを自分のブランチにマージしたり、ローカルブランチとしてチェックアウトして中身を調べたりといったことが可能となります。（ブランチの役割と使い方については、 Git のブランチ機能 で詳しく説明します。）

### リモートからのフェッチ、そしてプル

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

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

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

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

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

### リモートへのプッシュ

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

```
$ git push origin master
```

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

## リモートの調査

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

```
$ git remote show origin
* remote origin
  Fetch URL: https://github.com/schacon/ticgit
  Push  URL: https://github.com/schacon/ticgit
  HEAD branch: master
  Remote branches:
    master                               tracked
    dev-branch                           tracked
  Local branch configured for 'git pull':
    master merges with remote master
  Local ref configured for 'git push':
    master pushes to master (up to date)
```

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

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

```
$ git remote show origin
* remote origin
  URL: https://github.com/my-org/complex-project
  Fetch URL: https://github.com/my-org/complex-project
  Push  URL: https://github.com/my-org/complex-project
  HEAD branch: master
  Remote branches:
    master                               tracked
    dev-branch                           tracked
    markdown-strip                       tracked
    issue-43                             new (next fetch will store in
remotes/origin)
    issue-45                             new (next fetch will store in
remotes/origin)
    refs/remotes/origin/issue-11     stale (use 'git remote prune' to
remove)
  Local branches configured for 'git pull':
    dev-branch merges with remote dev-branch
    master     merges with remote master
  Local refs configured for 'git push':
    dev-branch                           pushes to dev-branch
(up to date)
    markdown-strip                       pushes to markdown-strip
(up to date)
    master                               pushes to master
(up to date)
```

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

### リモートの削除・リネーム

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

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

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

何らかの理由でリモートを削除したい場合 (サーバーを移動したとか特定のミラーを使わなくなったとか、あ

るいはプロジェクトからメンバーが抜けたとかいった場合) は `git remote rm` を使用します。

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

# タグ

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

## タグの一覧表示

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

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

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

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

```
$ git tag -l "v1.8.5*"
v1.8.5
v1.8.5-rc0
v1.8.5-rc1
v1.8.5-rc2
v1.8.5-rc3
v1.8.5.1
v1.8.5.2
v1.8.5.3
v1.8.5.4
v1.8.5.5
```

## タグの作成

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

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

ません。

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

## 注釈付きのタグ

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

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

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

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

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

my version 1.4

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

    changed the version number
```

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

## 軽量版のタグ

コミットにタグをつけるもうひとつの方法が、軽量版のタグです。 これは基本的に、コミットのチェックサ

ムだけを保持するもので、それ以外の情報は含まれません。 軽量版のタグを作成するには `-a`、`-s` あるいは `-m` といったオプションをつけずにコマンドを実行します。

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

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

```
$ git show v1.4-lw
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    changed the version number
```

### 後からのタグ付け

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

```
$ git log --pretty=oneline
15027957951b64cf874c3557a0f3547bd83b3ff6 Merge branch 'experiment'
a6b4c97498bd301d84096da251c98a07c7723e65 beginning write support
0d52aaab4479697da7686c15f77a3d64d9165190 one more thing
6d52a271eda8725415634dd79daabbc4d9b6008e Merge branch 'experiment'
0b7434d86859cc7b8c3d5e1dddfed66ff742fcbc added a commit function
4682c3261057305bdd616e23b64b0857d832627b added a todo file
166ae0c4d3f420721acbb115cc33848dfcc2121a started write support
9fceb02d0ae598e95dc970b74767f19372d61af8 updated rakefile
964f16d36dfccde844893cac5b347e7b3d44abbc commit the todo
8a5cbc430f1a9c3d00faaeffd07798508422908a updated readme
```

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

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

これで、そのコミットにタグがつけられたことが確認できます。

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

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

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

    updated rakefile
...
```

### タグの共有

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

```
$ git push origin v1.5
Counting objects: 14, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (12/12), done.
Writing objects: 100% (14/14), 2.05 KiB | 0 bytes/s, done.
Total 14 (delta 3), reused 0 (delta 0)
To git@github.com:schacon/simplegit.git
 * [new tag]         v1.5 -> v1.5
```

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

```
$ git push origin --tags
Counting objects: 1, done.
Writing objects: 100% (1/1), 160 bytes | 0 bytes/s, done.
Total 1 (delta 0), reused 0 (delta 0)
To git@github.com:schacon/simplegit.git
 * [new tag]         v1.4 -> v1.4
 * [new tag]         v1.4-lw -> v1.4-lw
```

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

### タグのチェックアウト

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

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

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

## Git エイリアス

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

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

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

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

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

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

こうすれば、次のふたつのコマンドが同じ意味となります。

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

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

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

こうすれば、直近のコミットの情報を見ることができます。

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

    test for current head

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

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

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

# まとめ

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

# Git のブランチ機能

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

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

## ブランチとは

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

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

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

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

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

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

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

ェクトです。

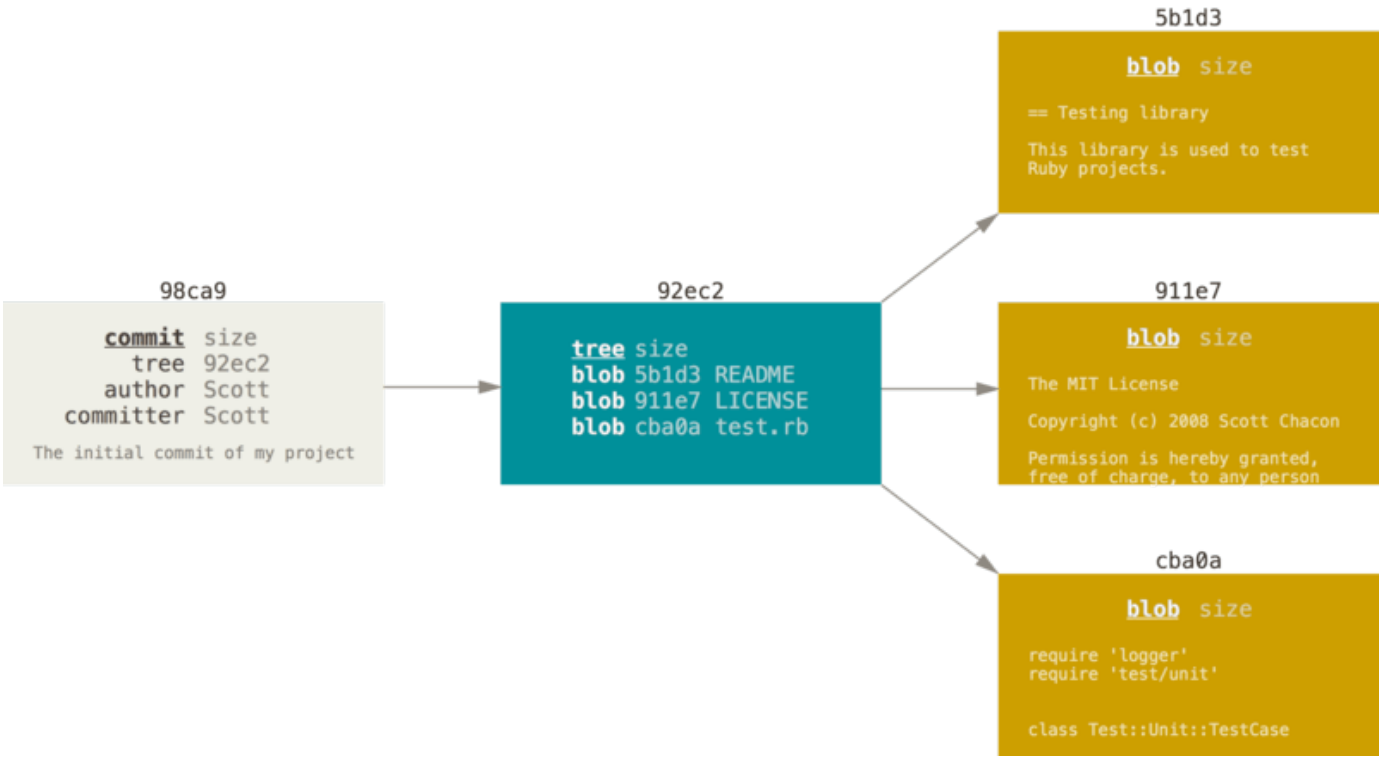



*Figure 9. コミットおよびそのツリー*

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

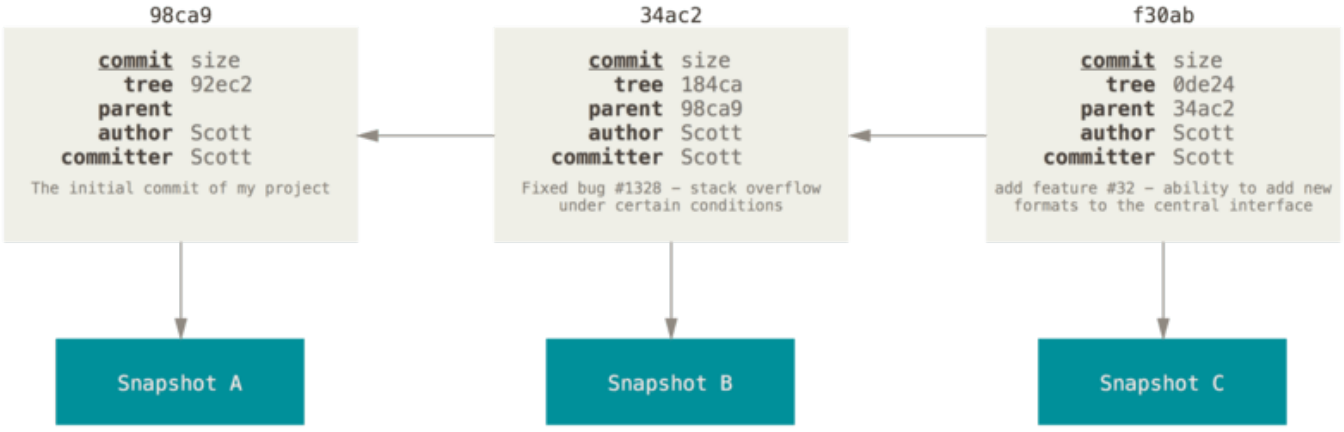



*Figure 10. コミットおよびその親*

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

**NOTE**

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

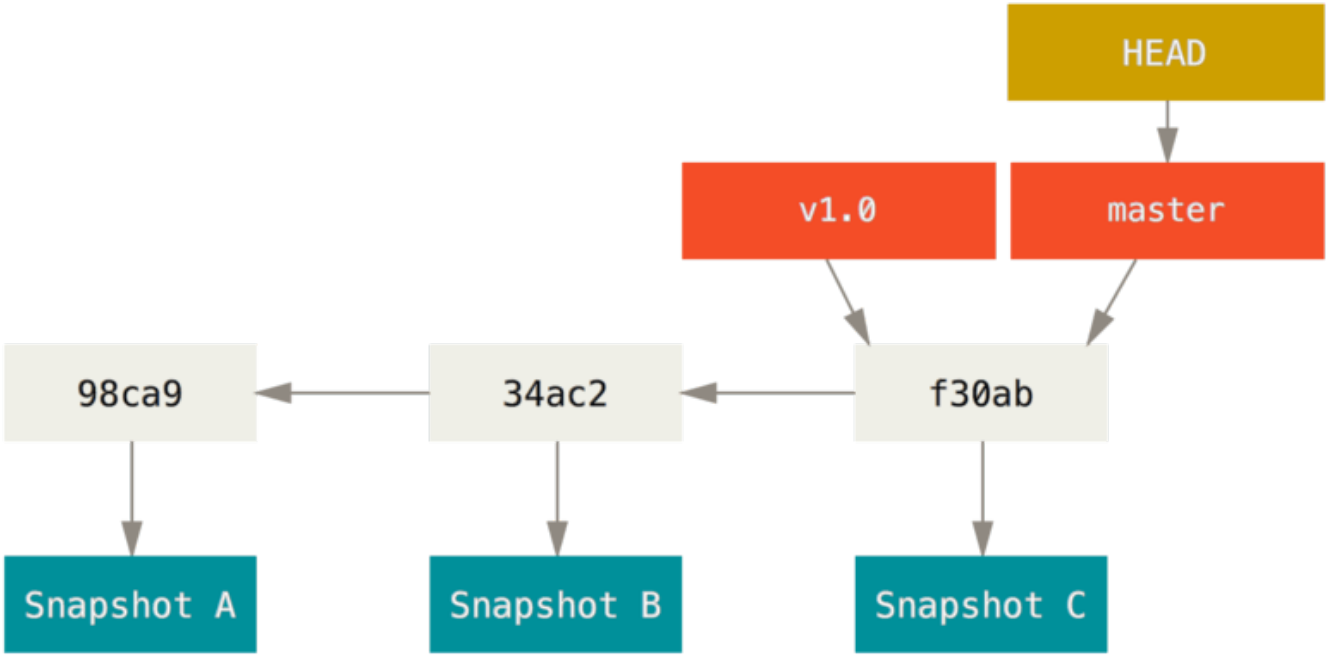



*Figure 11. ブランチおよびそのコミットの歴史*

## 新しいブランチの作成

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

```
$ git branch testing
```

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

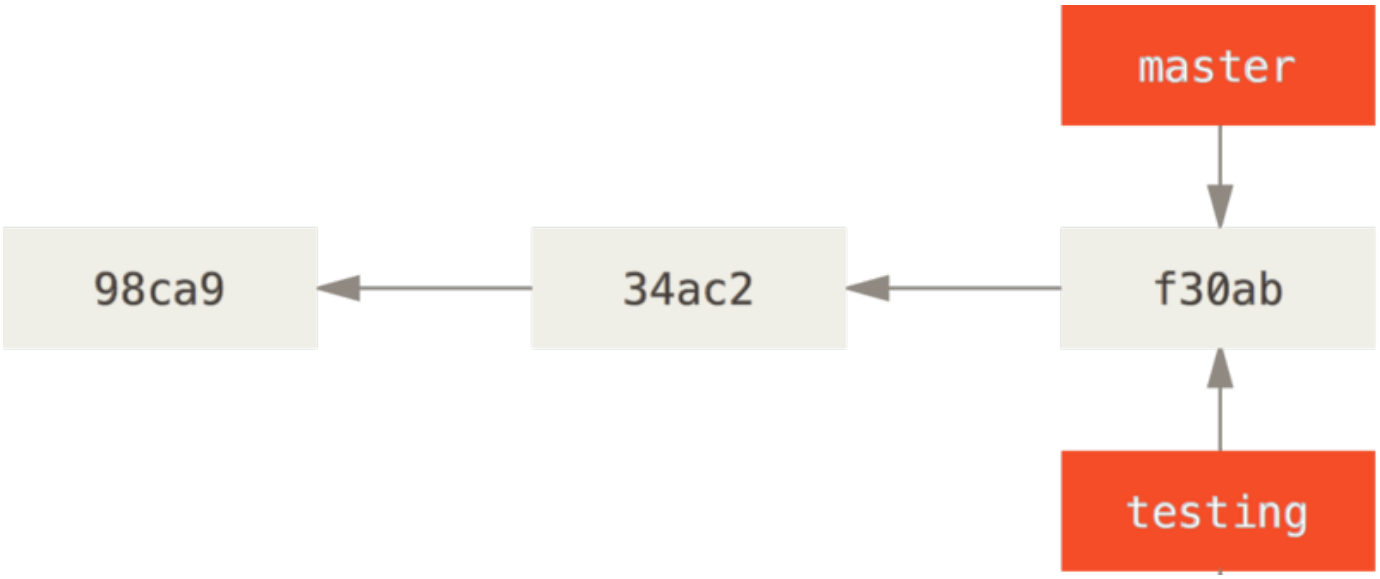



*Figure 12. ふたつのブランチが同じ一連のコミットを指す*

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

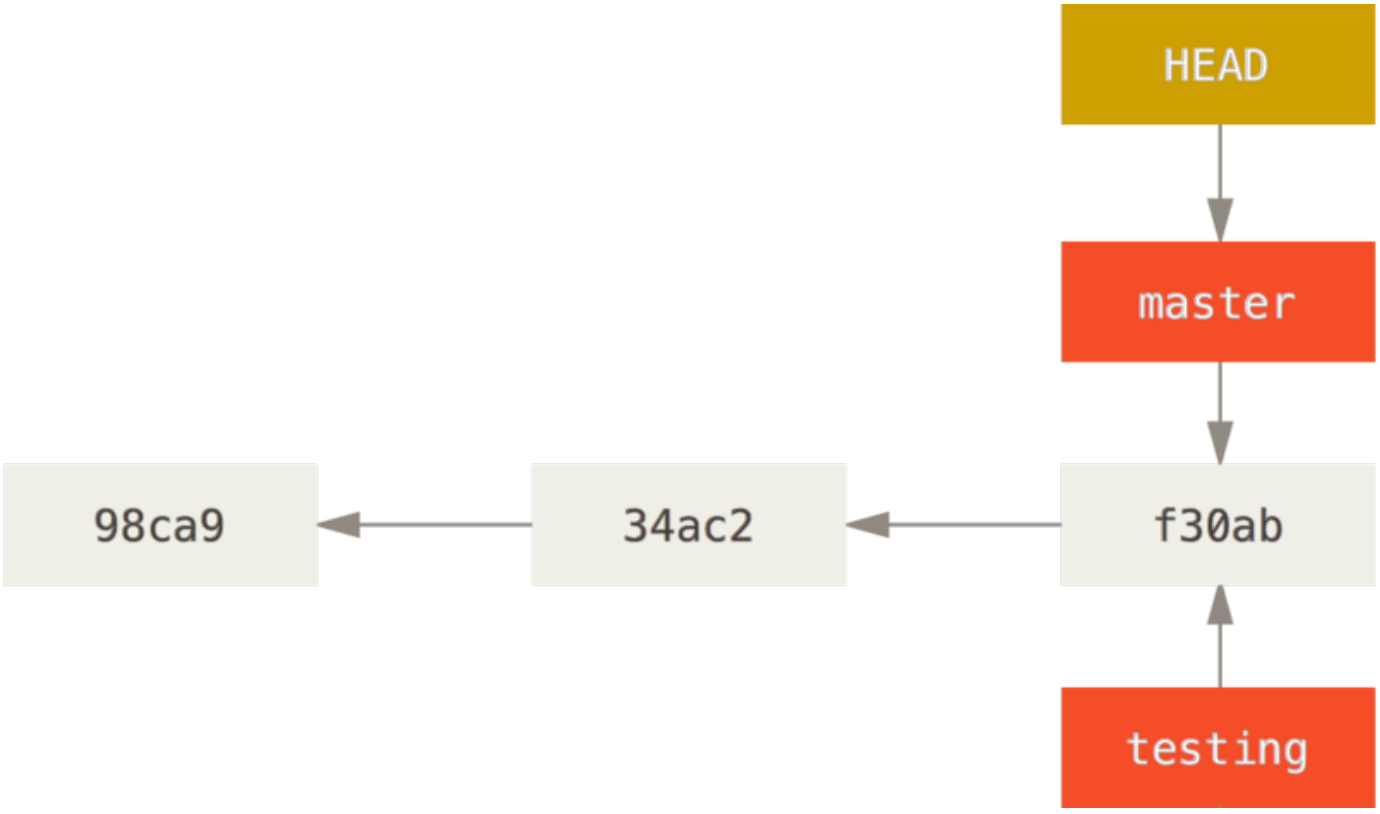



*Figure 13. ブランチを指す HEAD*

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

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

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

### ブランチの切り替え

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

```
$ git checkout testing
```

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

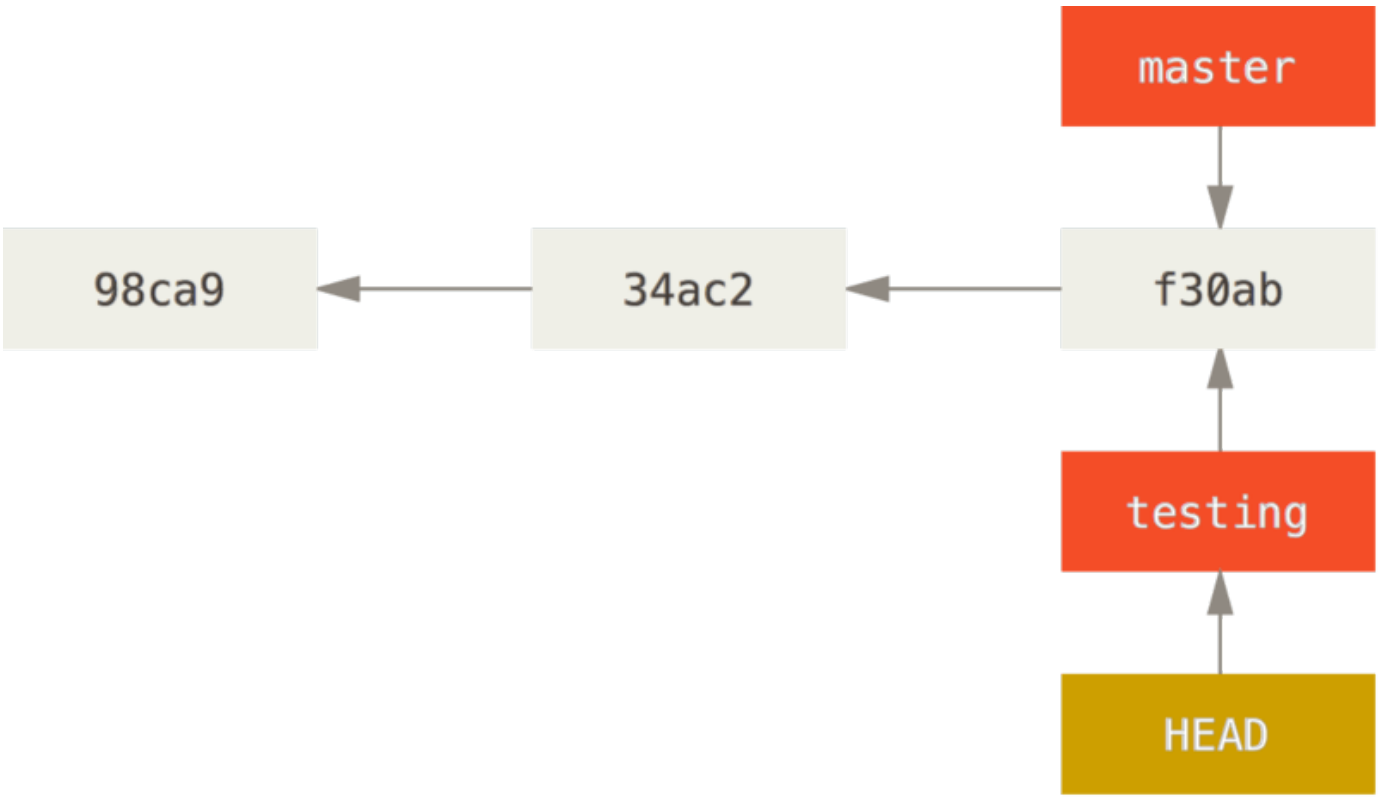



*Figure 14. HEAD は現在のブランチを指す*

それがどうしたって? では、ここで別のコミットをしてみましょう。

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

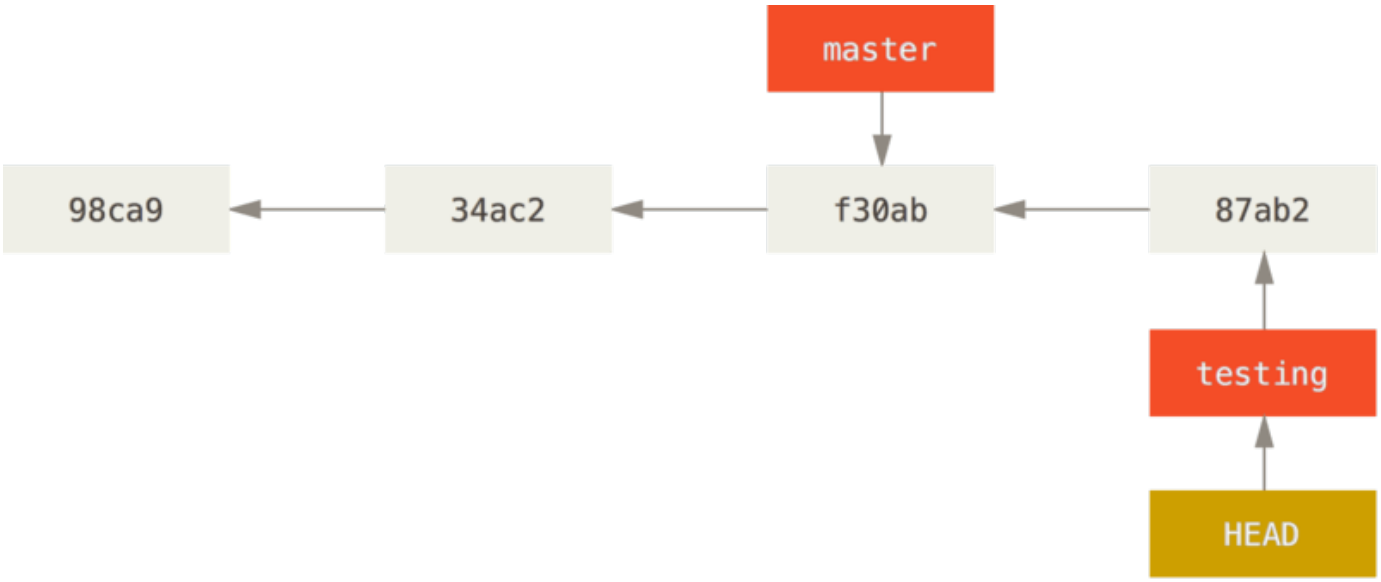



*Figure 15. HEAD が指すブランチが、コミットによって移動する*

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

```
$ git checkout master
```

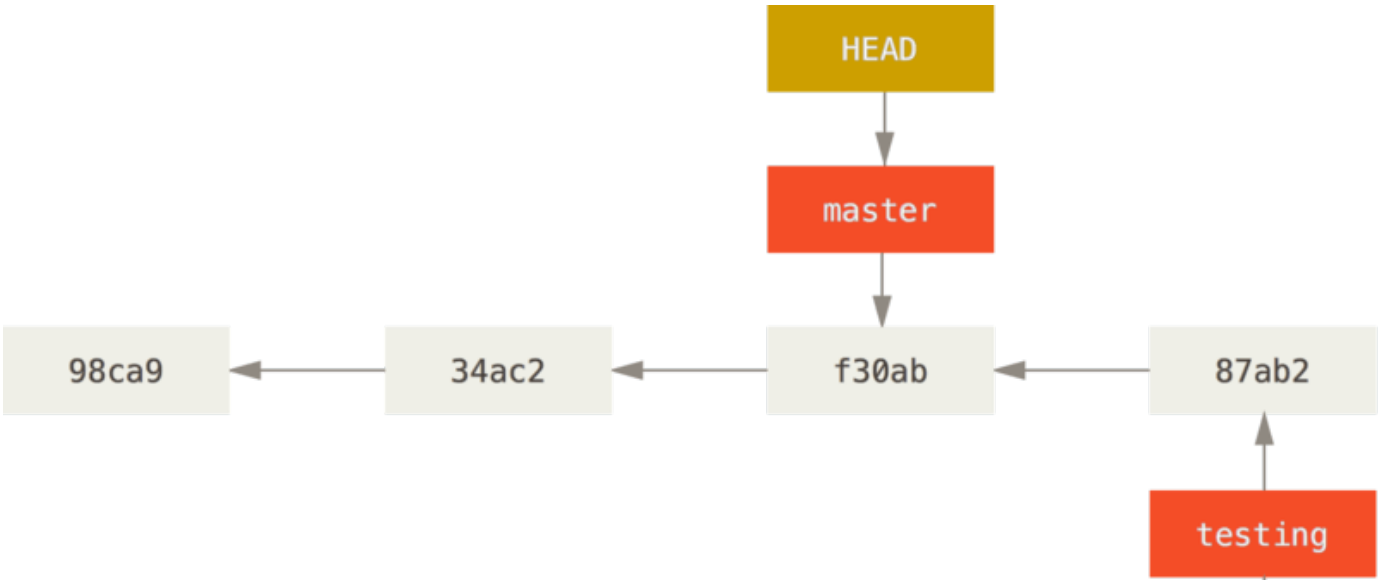



*Figure 16. チェックアウトによって HEAD が移動する*

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

なります。

**NOTE**

*ブランチを切り替えると、作業ディレクトリのファイルが変更される*

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

それでは、ふたたび変更を加えてコミットしてみましょう。

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

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

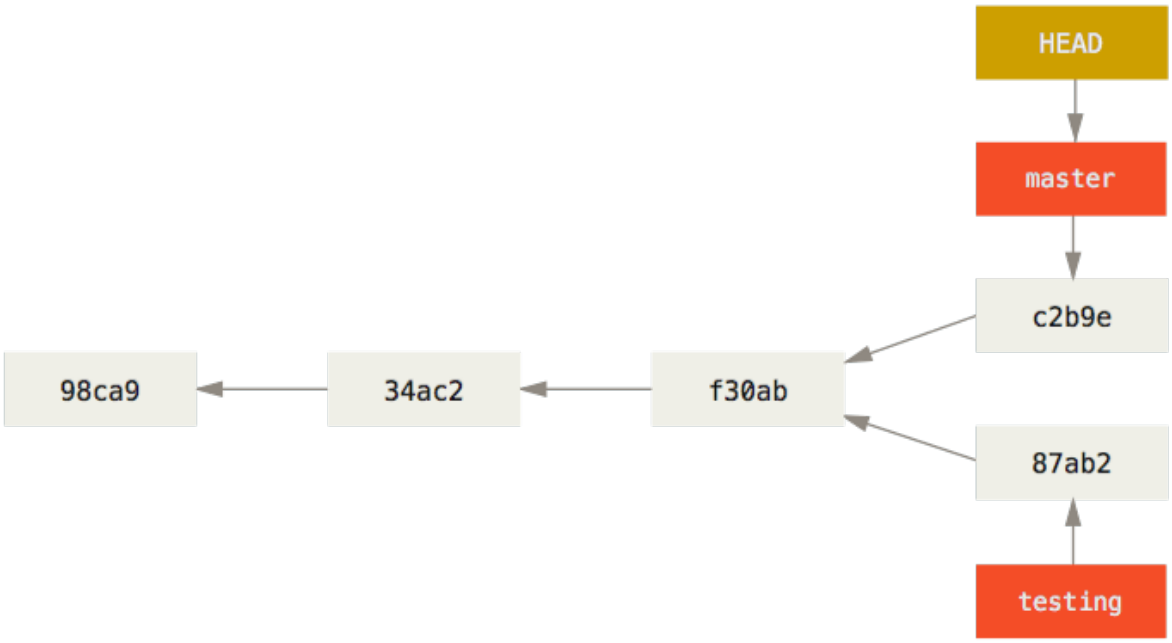



*Figure 17. 分裂した歴史*

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

いるのかや、歴史がどこで分裂したのかも表示します。

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

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

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

では、なぜブランチを切るべきなのかについて見ていきましょう。

# ブランチとマージの基本

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

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

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

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

## ブランチの基本

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

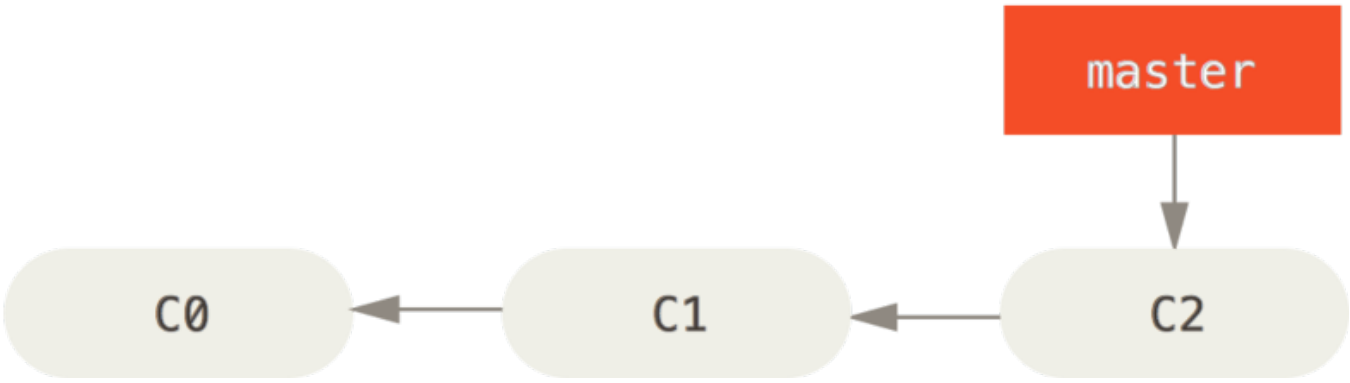



*Figure 18. 単純なコミットの歴史*

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

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

これは、次のコマンドのショートカットです。

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

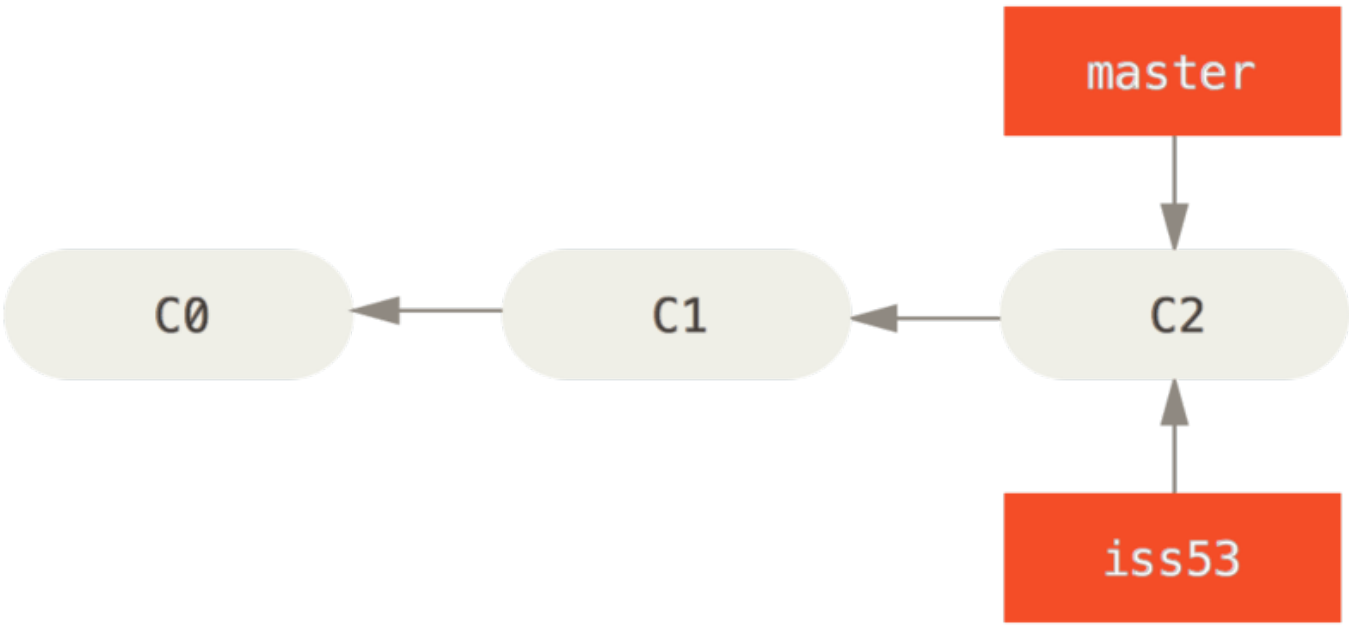



*Figure 19. 新たなブランチポインタの作成*

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

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

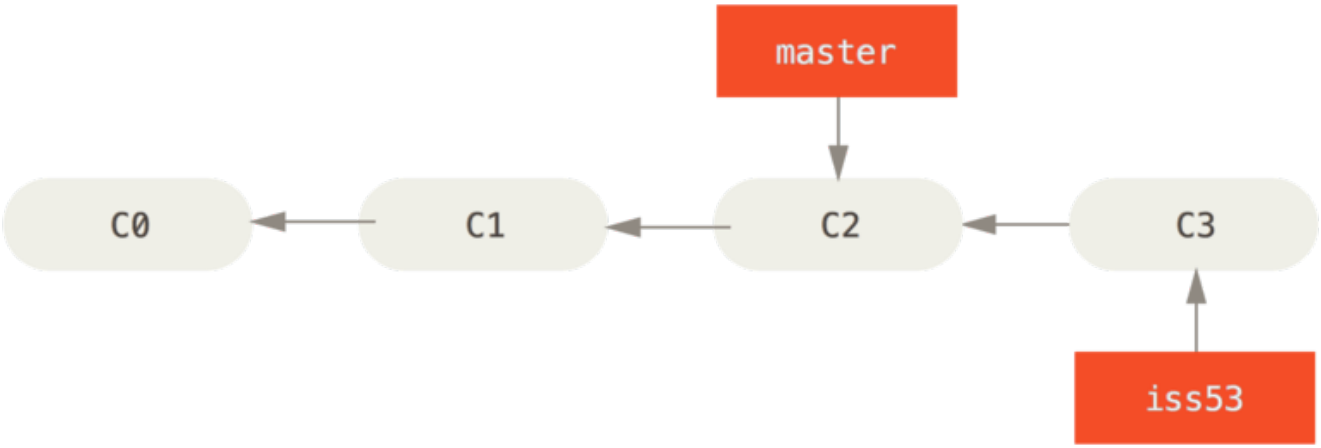



*Figure 20. 作業した結果、iss53 ブランチが移動した*

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

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

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

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

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

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

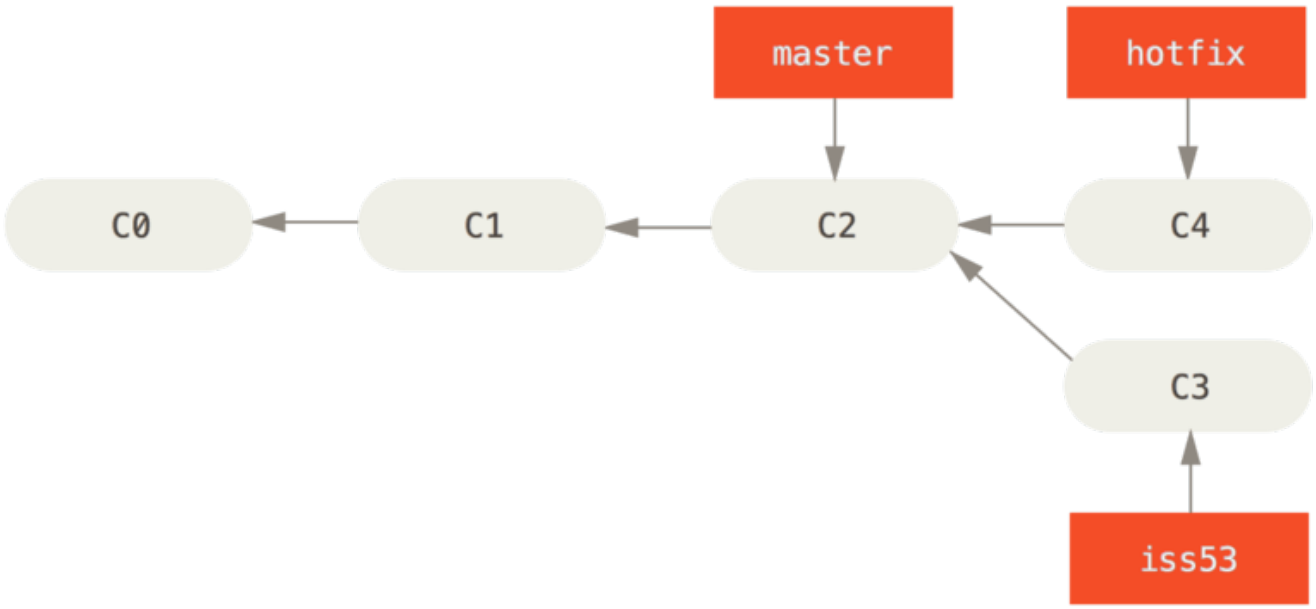



*Figure 21. `master` から新たに作成した hotfix ブランチ*

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

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

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

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

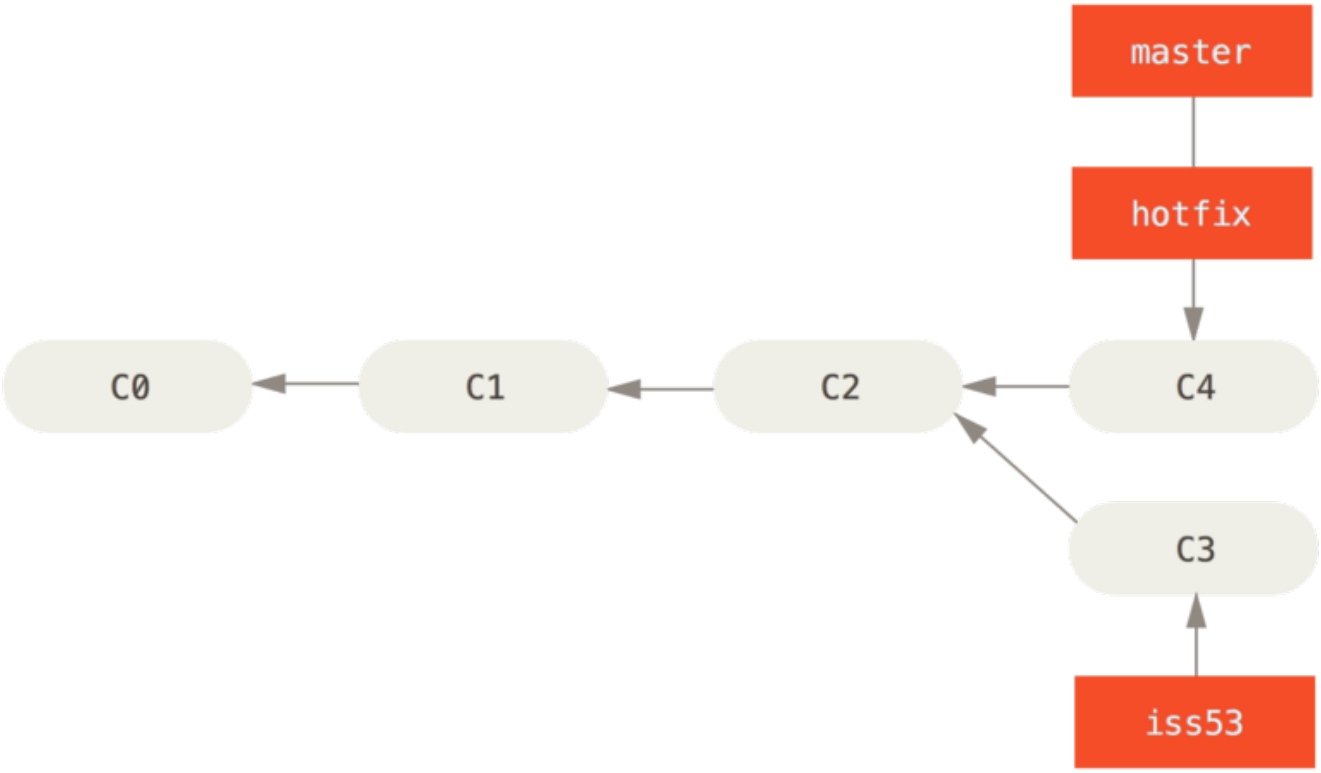



*Figure 22. `master`が`hotfix`にfast-forwardされた*

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

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

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

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

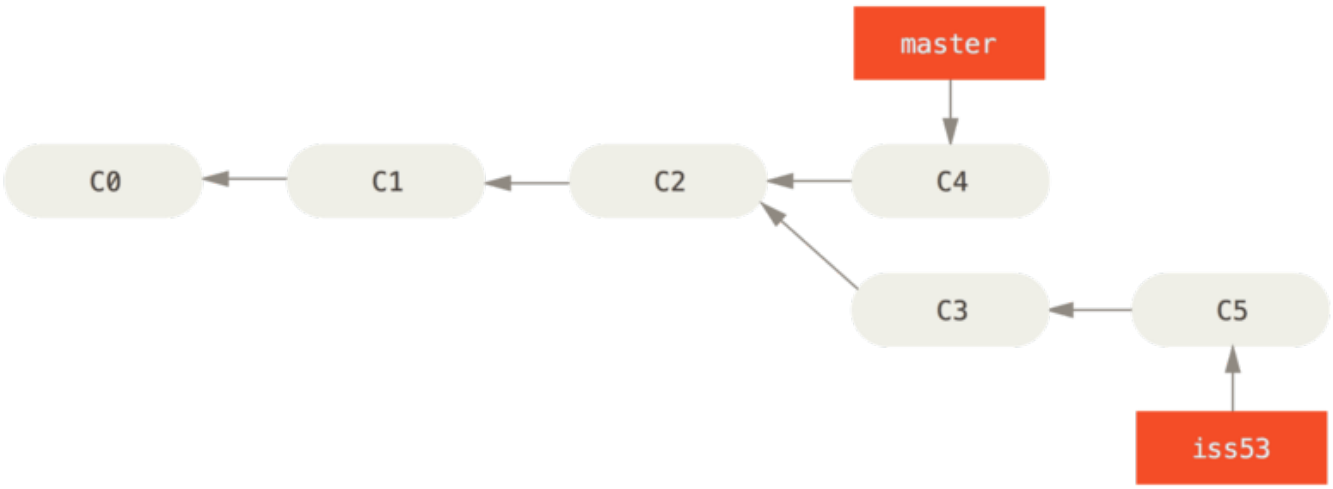



*Figure 23. iss53 の作業を続ける*

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

## マージの基本

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

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

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

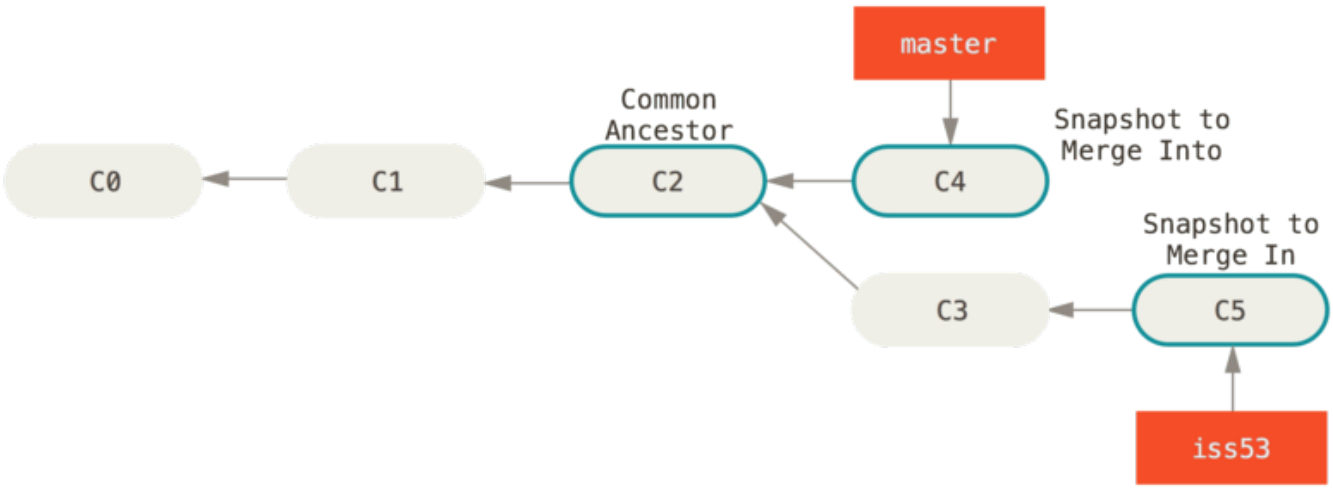



*Figure 24. 三つのスナップショットを使ったマージ*

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

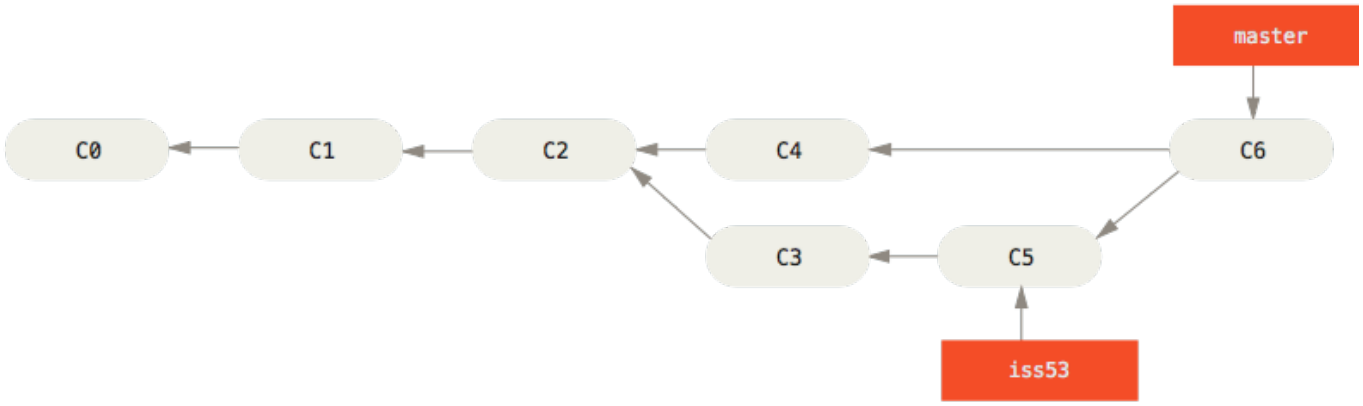



*Figure 25. マージコミット*

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

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

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

## マージ時のコンフリクト

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

```
$ git merge iss53
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
Automatic merge failed; fix conflicts and then commit the result.
```

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

```
$ git status
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")

Unmerged paths:
  (use "git add <file>..." to mark resolution)

    both modified:      index.html

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

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

```
<<<<<<< HEAD:index.html
<div id="footer">contact : email.support@github.com</div>
=======
<div id="footer">
 please contact us at support@github.com
</div>
>>>>>>> iss53:index.html
```

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

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

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

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

```
$ git mergetool

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

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

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

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

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

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

Changes to be committed:

    modified:   index.html
```

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

```
Merge branch 'iss53'

Conflicts:
    index.html
#
# It looks like you may be committing a merge.
# If this is not correct, please remove the file
#   .git/MERGE_HEAD
# and try again.


# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
# All conflicts fixed but you are still merging.
#
# Changes to be committed:
#   modified:   index.html
#
```

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

## ブランチの管理

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

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

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

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

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

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

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

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

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

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

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

```
$ git branch -d testing
error: The branch 'testing' is not fully merged.
If you are sure you want to delete it, run 'git branch -D testing'.
```

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

セージで教えてくれていますね。

# ブランチでの作業の流れ

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

## 長期稼働用ブランチ

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

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

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

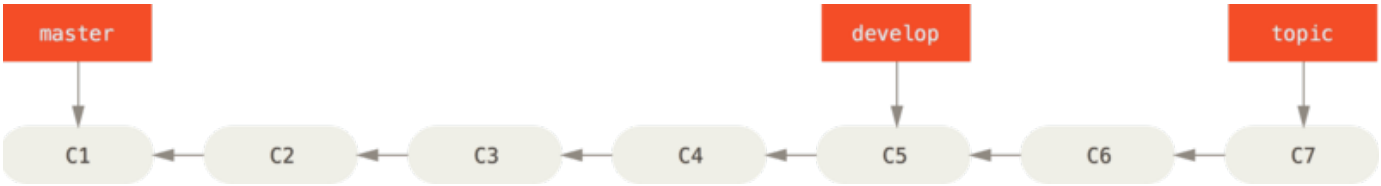



*Figure 26. 安定版と開発版のブランチの線形表示*

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

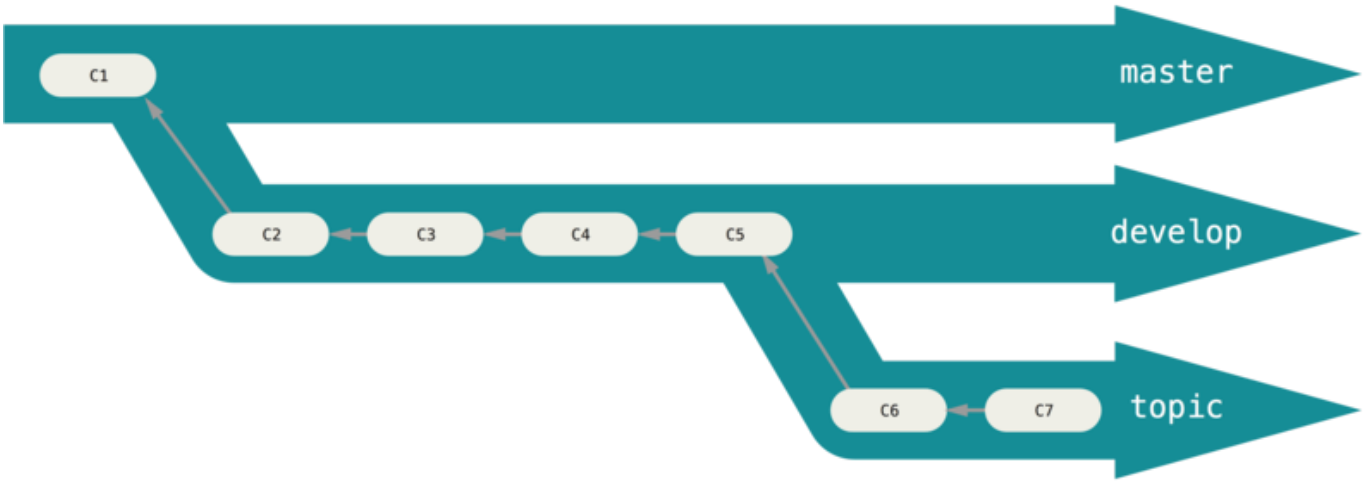



*Figure 27. 安定版と開発版のブランチの “サイロ” 表示*

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

## トピックブランチ

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

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

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

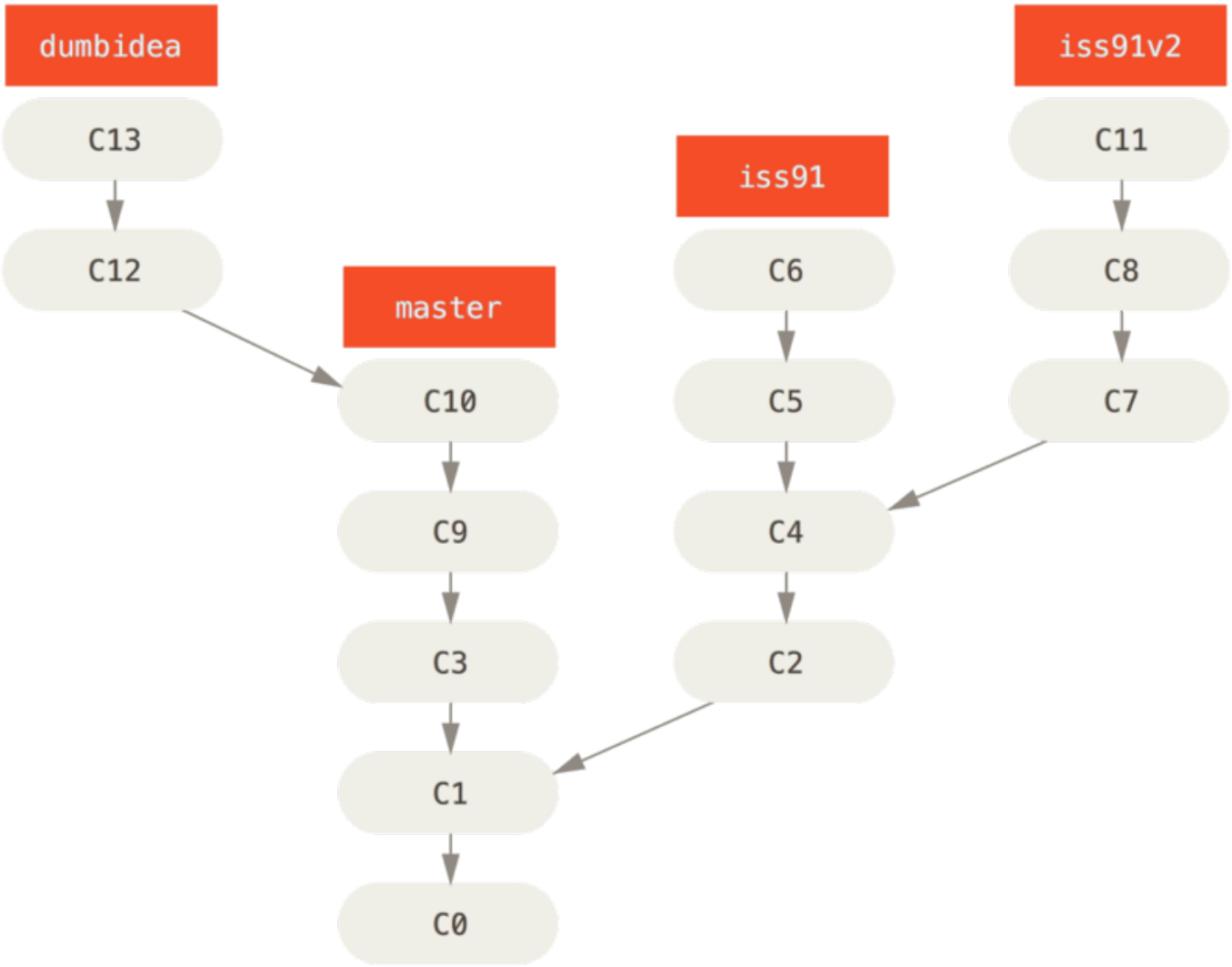



*Figure 28. 複数のトピックブランチ*

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

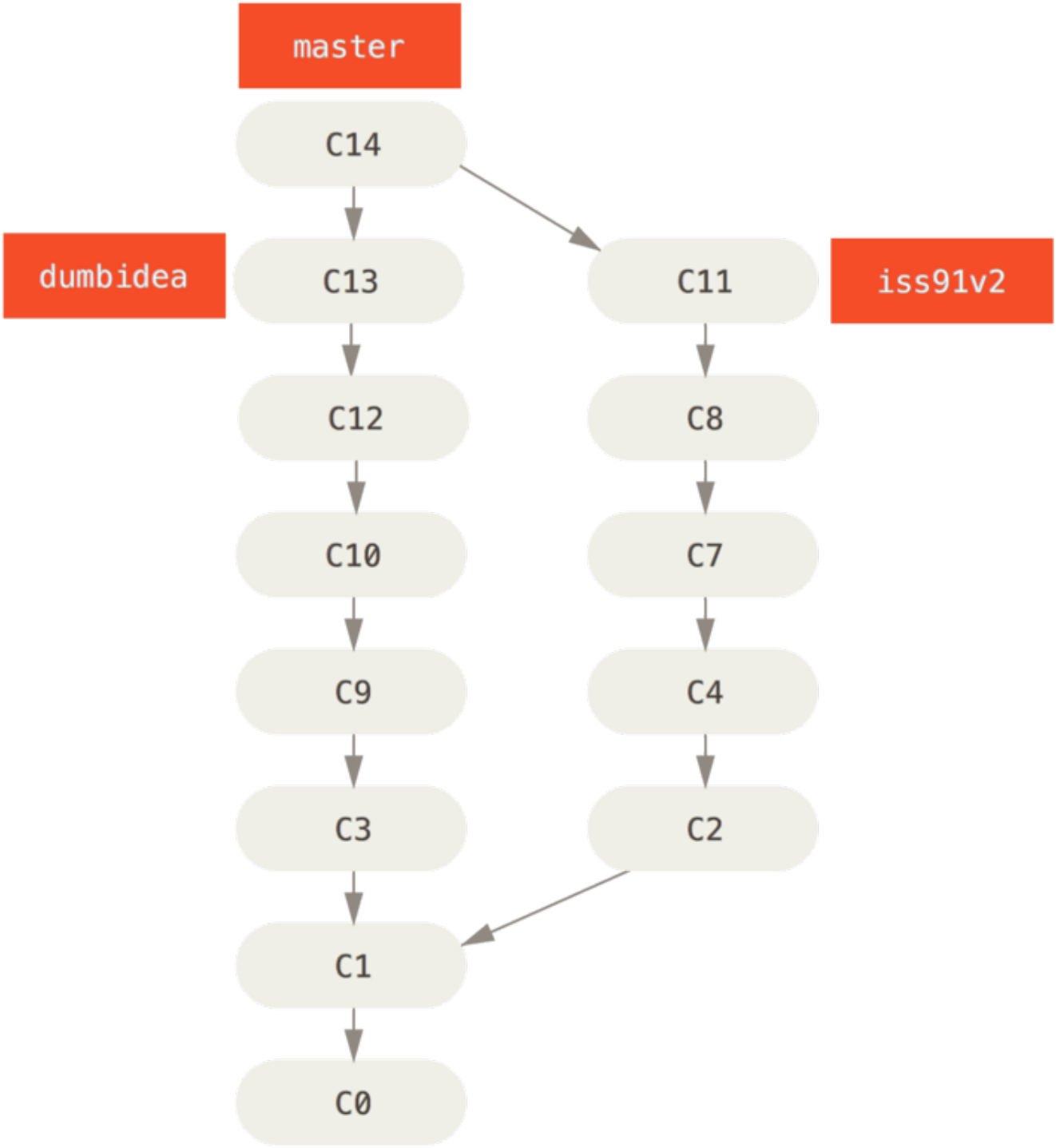



*Figure 29.* `dumbidea` と `iss91v2` をマージした後の歴史

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

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

# リモートブランチ

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

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

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

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

**NOTE**

*“origin” は特別なものではない*

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

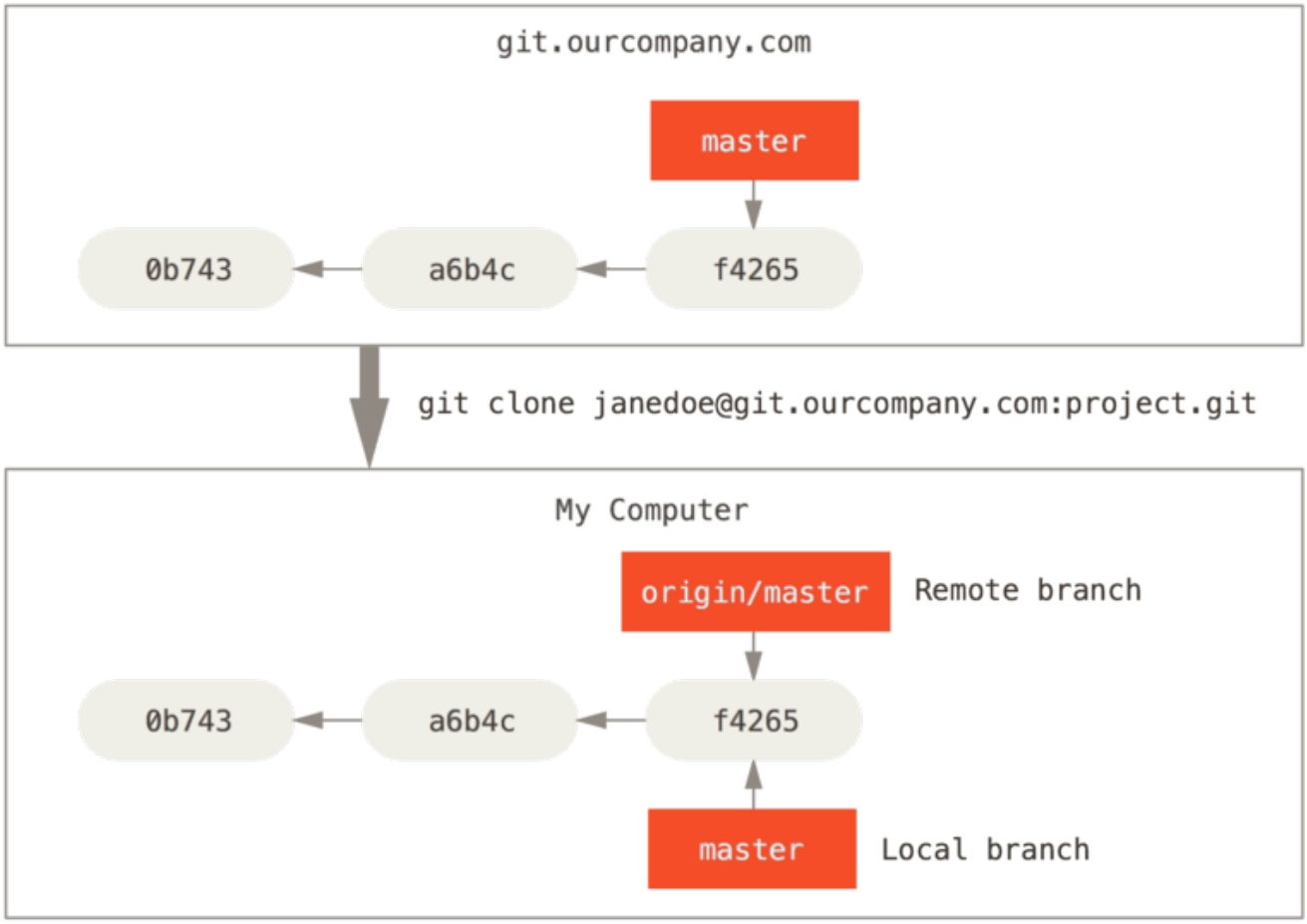



*Figure 30. クローン後のサーバーとローカルのリポジトリ*

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

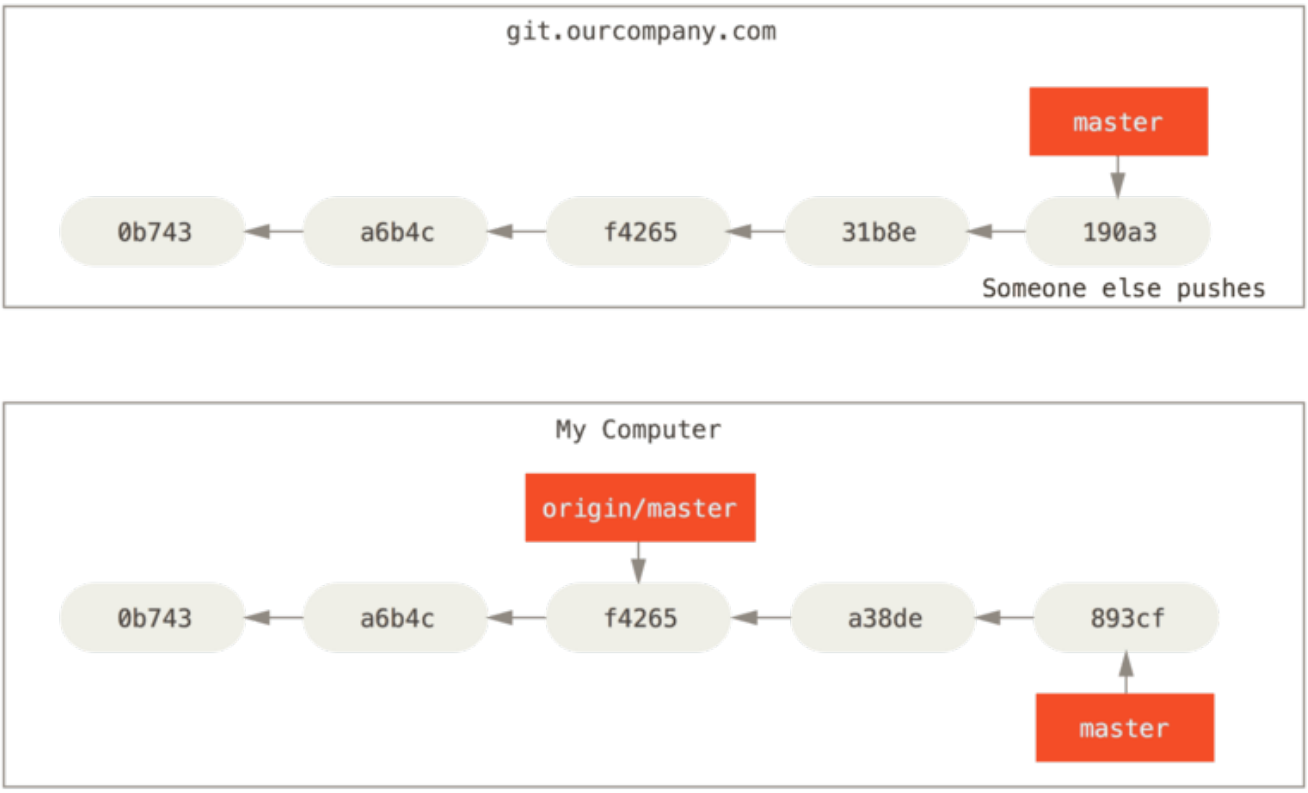



*Figure 31. ローカルとリモートの作業が枝分かれすることがある*

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

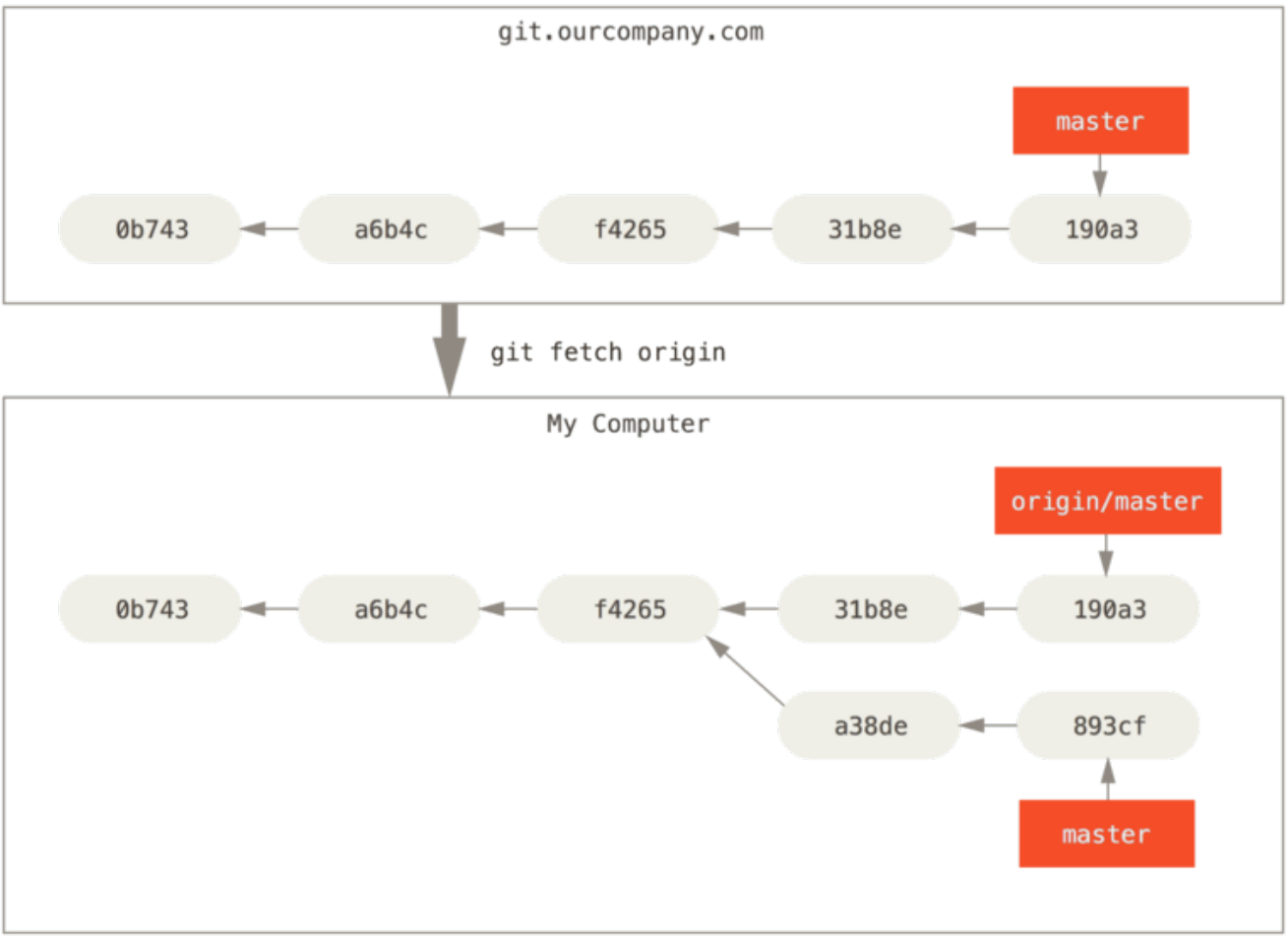



*Figure 32. `git fetch` によるリモートへの参照の更新*

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

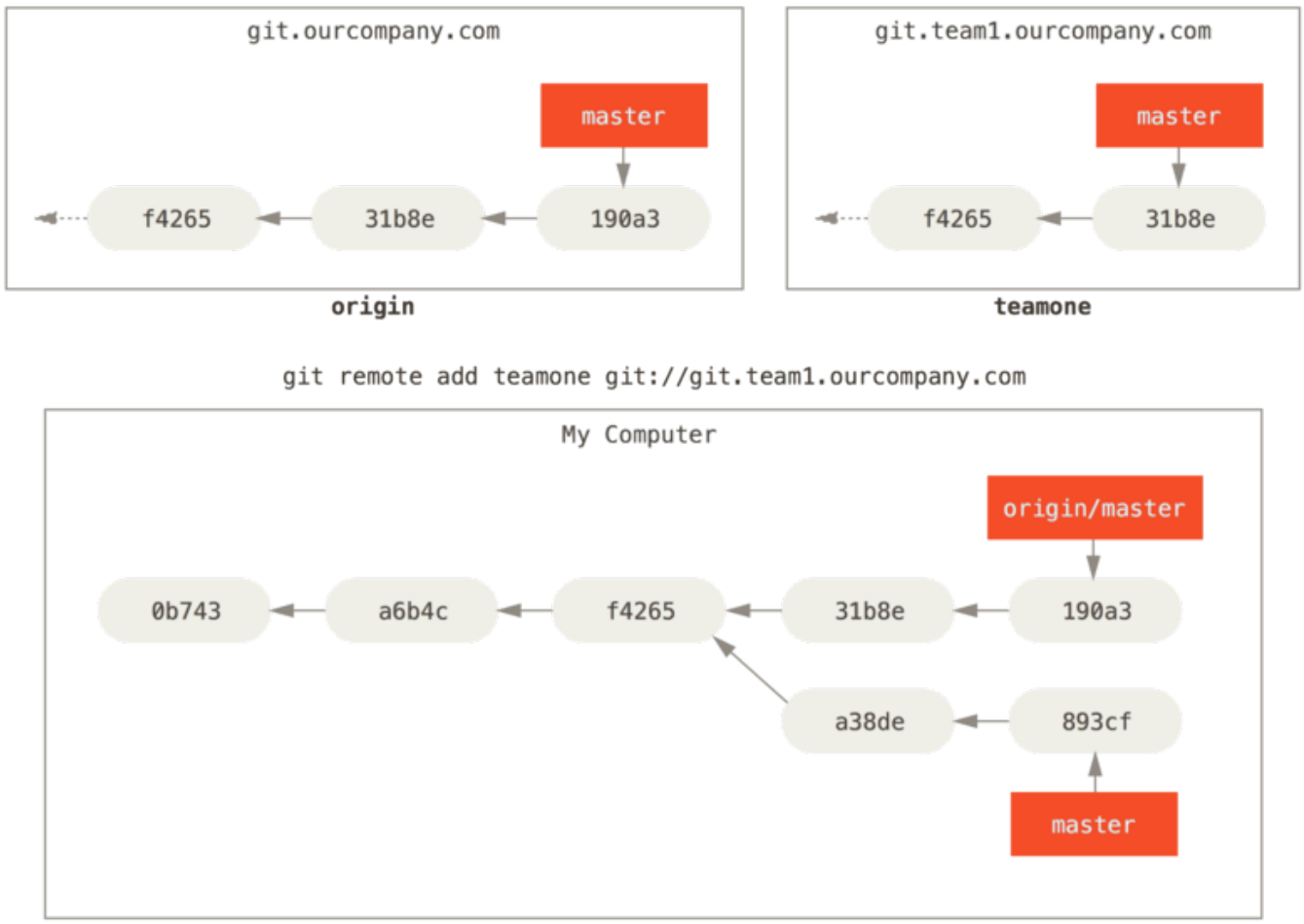



*Figure 33. 別のサーバーをリモートとして追加*

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

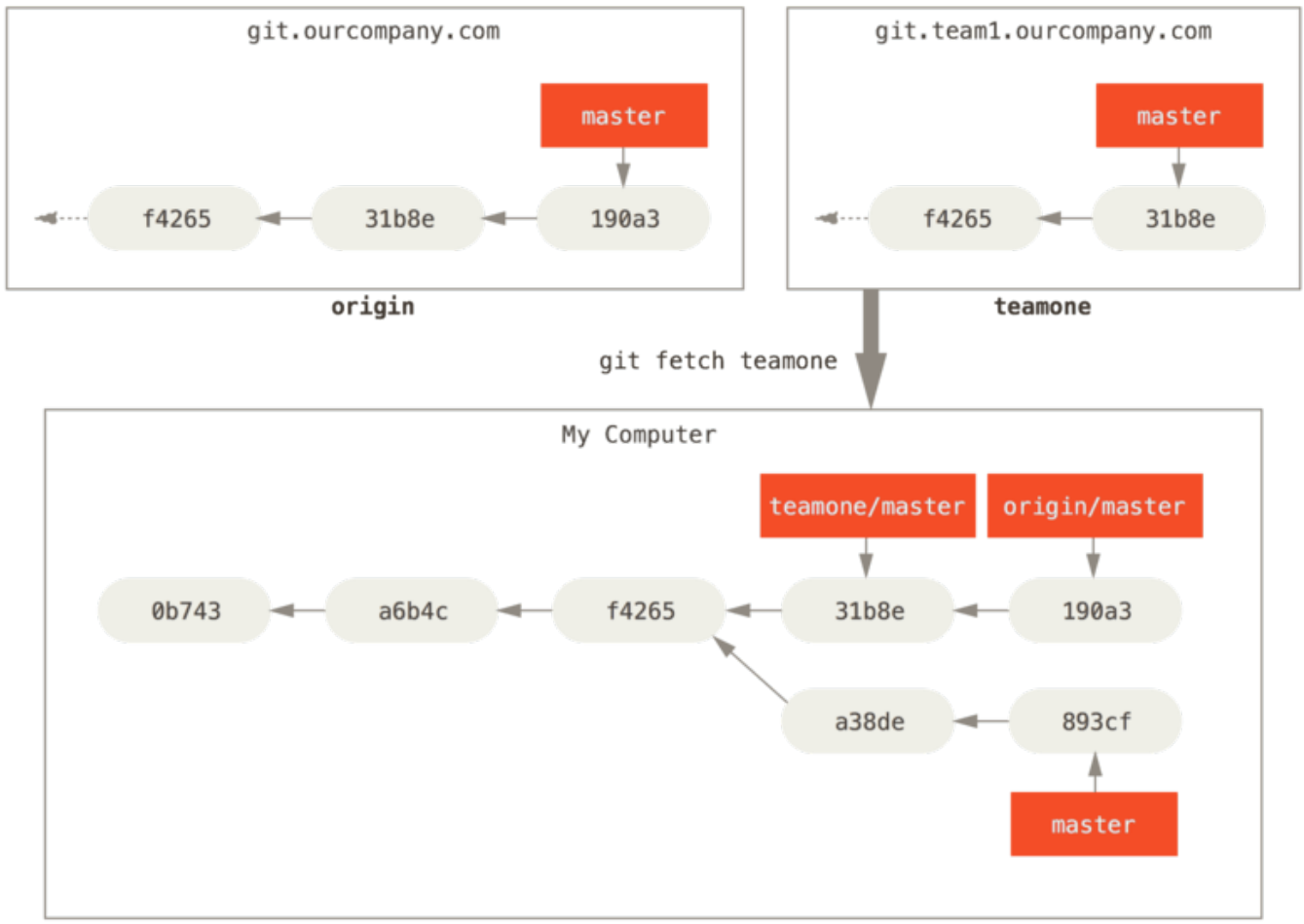



*Figure 34. リモート `teamone/master` を追跡するブランチ*

## プッシュ

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

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

```
$ git push origin serverfix
Counting objects: 24, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (15/15), done.
Writing objects: 100% (24/24), 1.91 KiB | 0 bytes/s, done.
Total 24 (delta 2), reused 0 (delta 0)
To https://github.com/schacon/simplegit
 * [new branch]      serverfix -> serverfix
```

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

**NOTE**

*パスワードを毎回入力したくない*

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

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

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

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

```
$ git fetch origin
remote: Counting objects: 7, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 3 (delta 0)
Unpacking objects: 100% (3/3), done.
From https://github.com/schacon/simplegit
 * [new branch]      serverfix    -> origin/serverfix
```

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

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

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

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

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

### 追跡ブランチ

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

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

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

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

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

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

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

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

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

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

**NOTE**

*上流の短縮記法*

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

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

```
$ git branch -vv
  iss53     7e424c3 [origin/iss53: ahead 2] forgot the brackets
  master    1ae2a45 [origin/master] deploying index fix
* serverfix f8674d9 [teamone/server-fix-good: ahead 3, behind 1] this
should do it
  testing   5ea463a trying something new
```

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

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

### プル

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

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

### リモートブランチの削除

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

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

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

## リベース

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

### リベースの基本

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

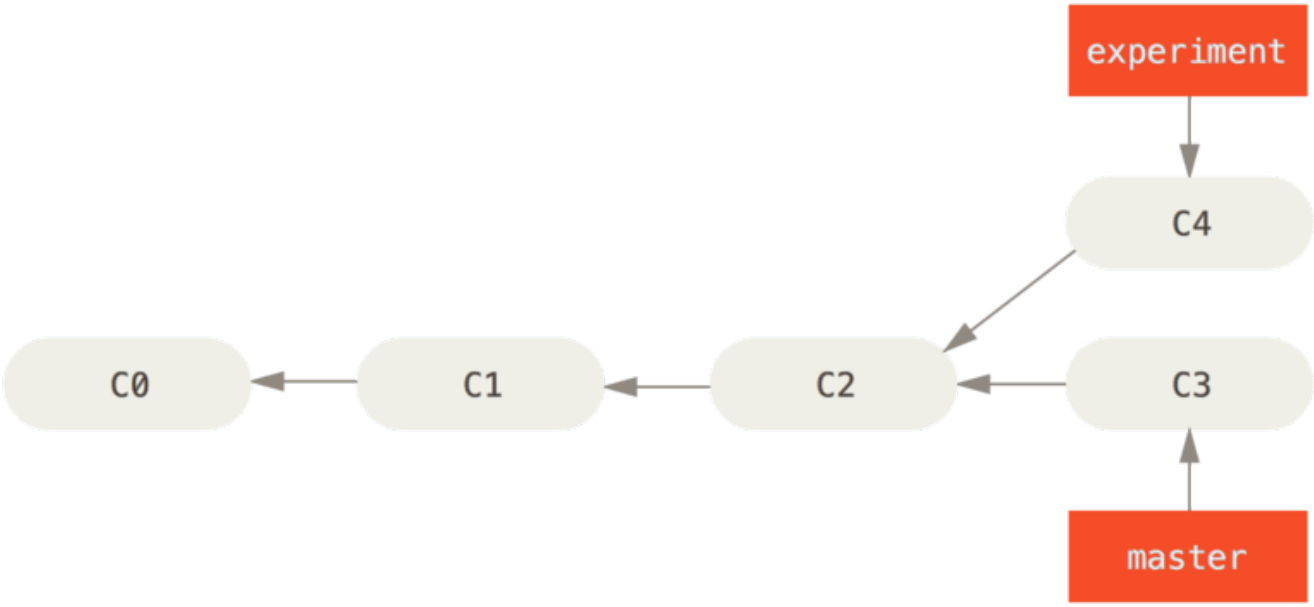



*Figure 35. シンプルな、分岐した歴史*

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

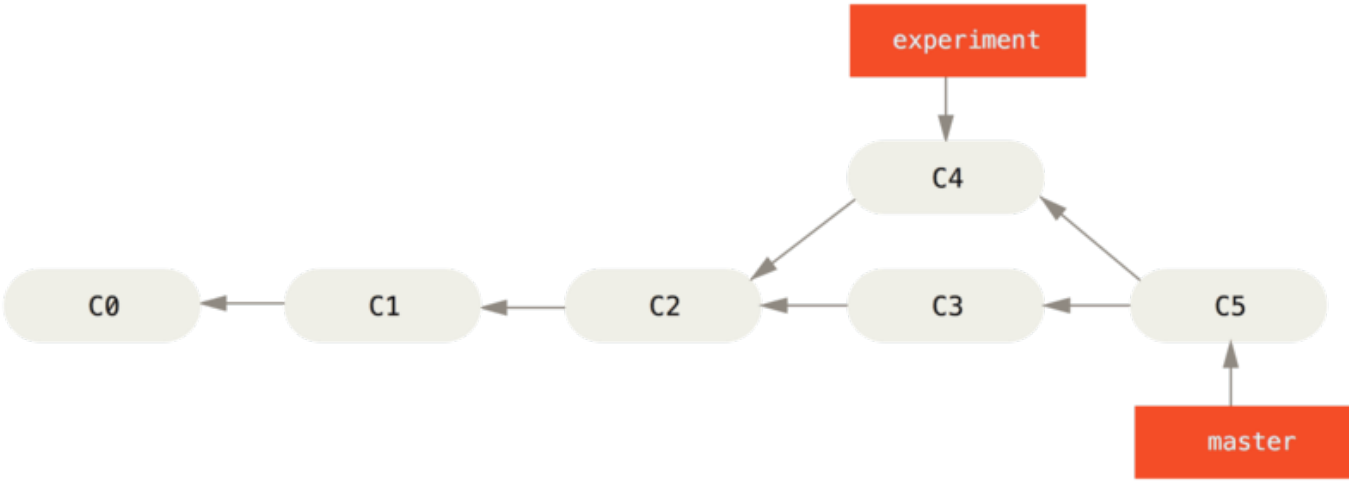



*Figure 36. 分岐した作業履歴をひとつに統合する*

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

今回の例では、次のように実行します。

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

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

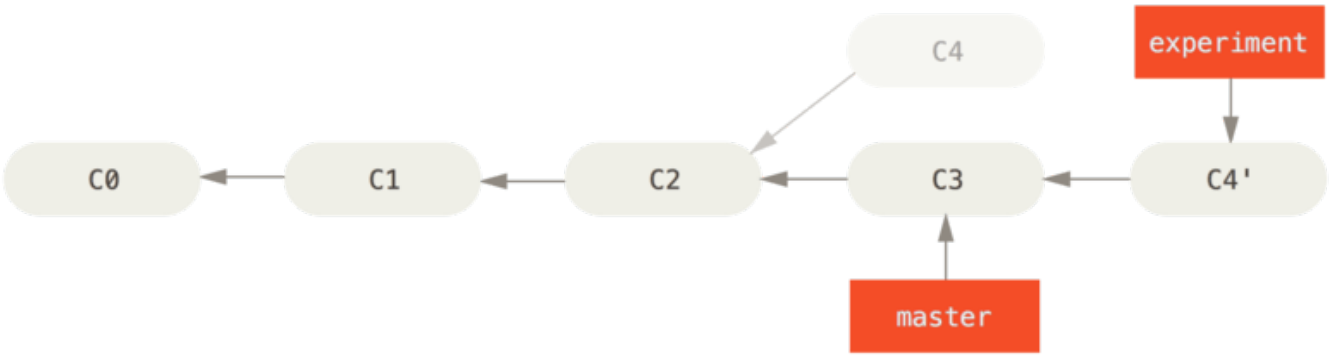



*Figure 37. C4 の変更を C3 にリベース*

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

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

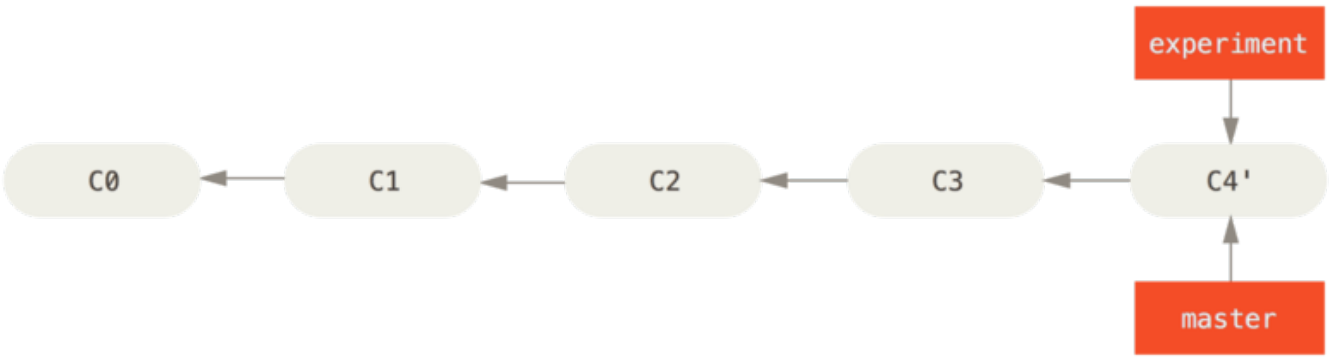



*Figure 38. master ブランチの Fast-forward*

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

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

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

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

### さらに興味深いリベース

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

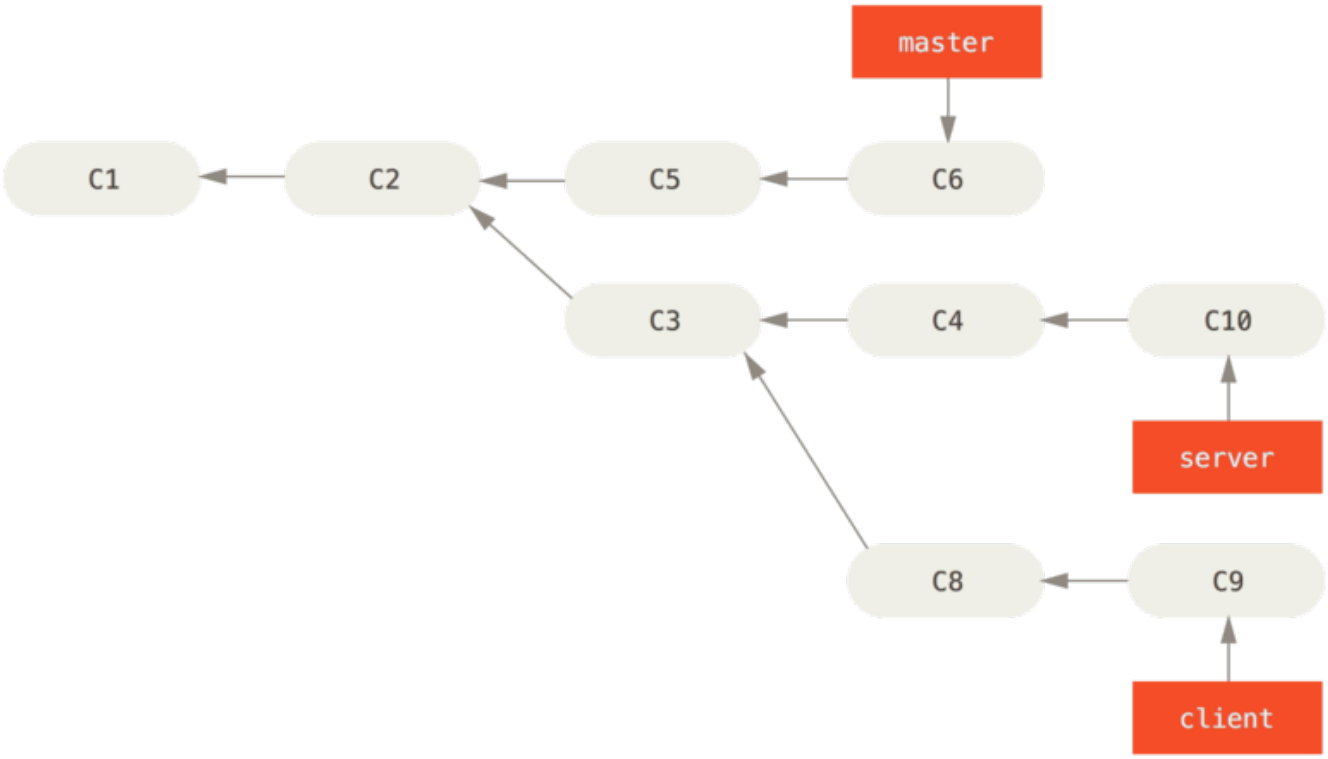



*Figure 39. トピックブランチからさらにトピックブランチを作成した歴史*

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

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

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

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

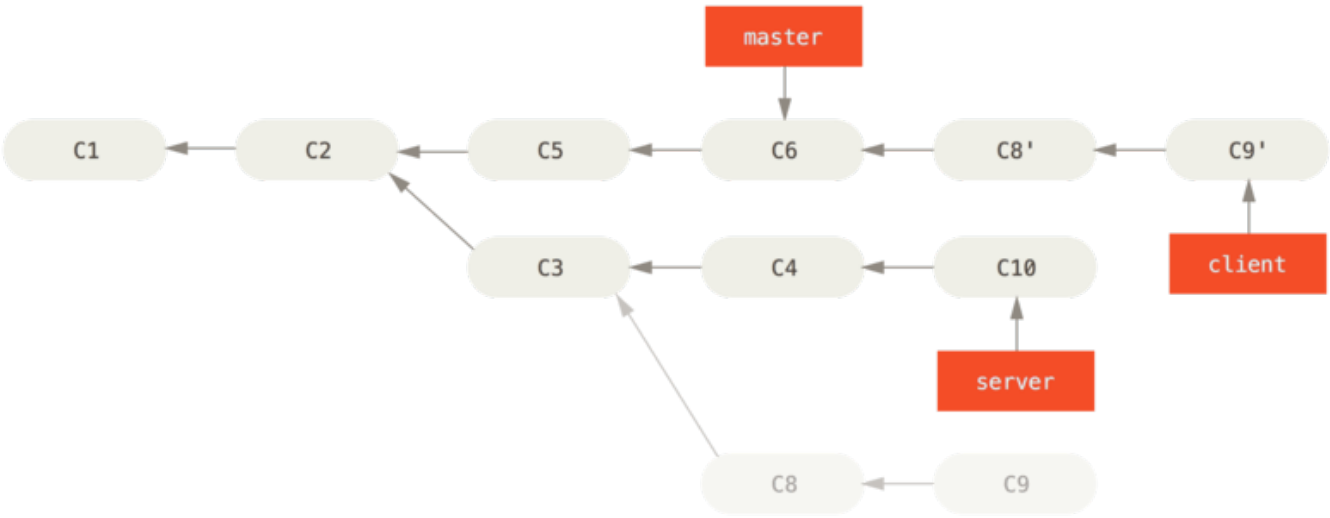



*Figure 40. 別のトピックブランチから派生したトピックブランチのリベース*

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

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

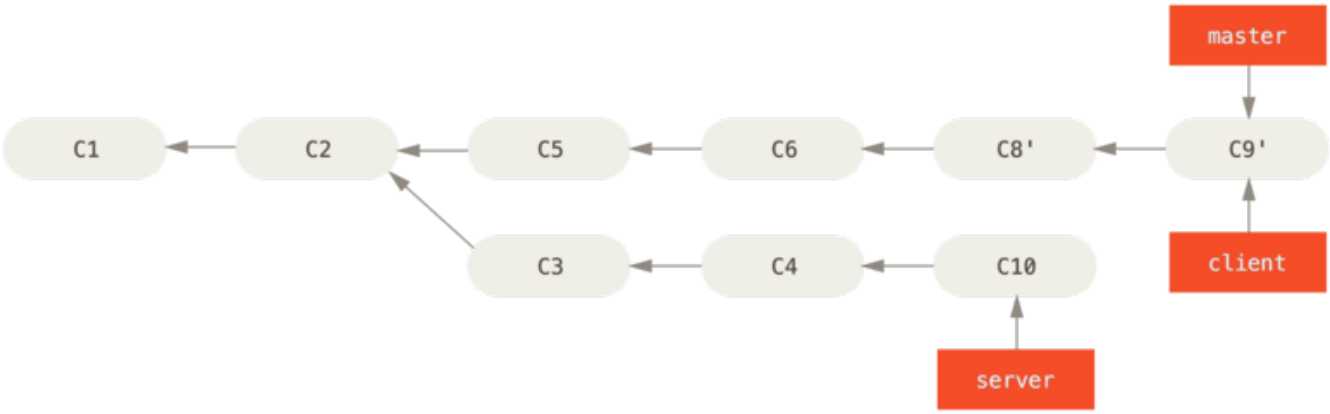



*Figure 41. master ブランチを fast-forward し、client ブランチの変更を含める*

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

```
$ git rebase master server
```

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

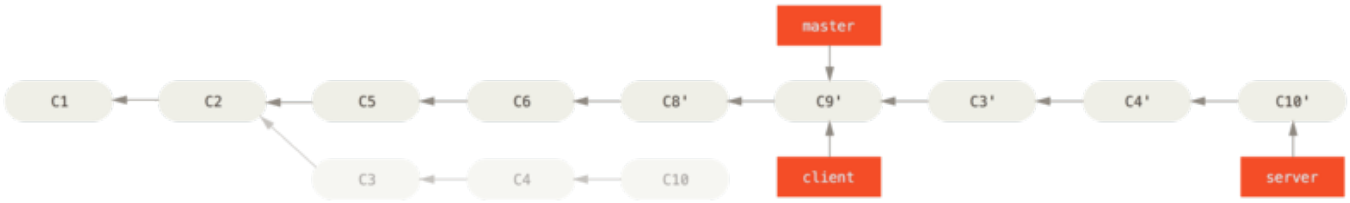



*Figure 42. server ブランチを master ブランチ上にリベースする*

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

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

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

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


*Figure 43. 最終的なコミット履歴*

## ほんとうは怖いリベース

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

**公開リポジトリにプッシュしたコミットをリベースしてはいけない**

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

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

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

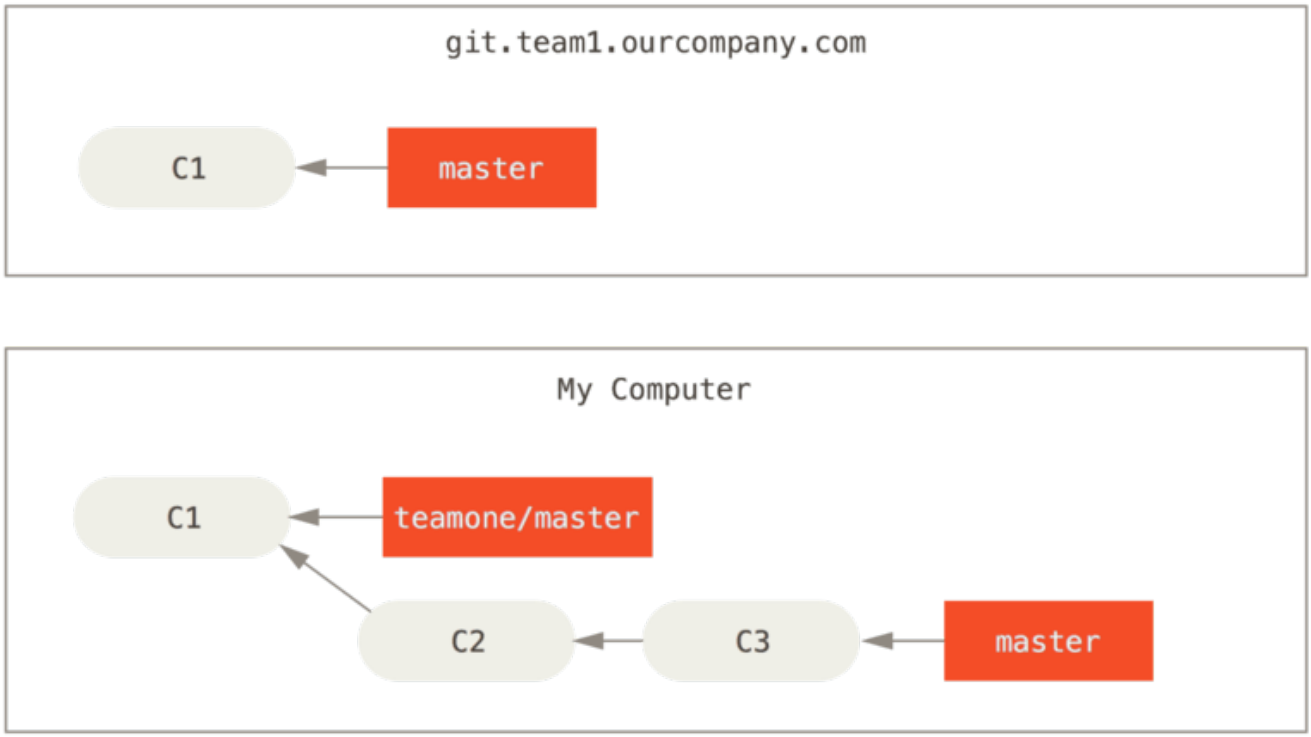



*Figure 44. リポジトリをクローンし、なんらかの作業をすませた状態*

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

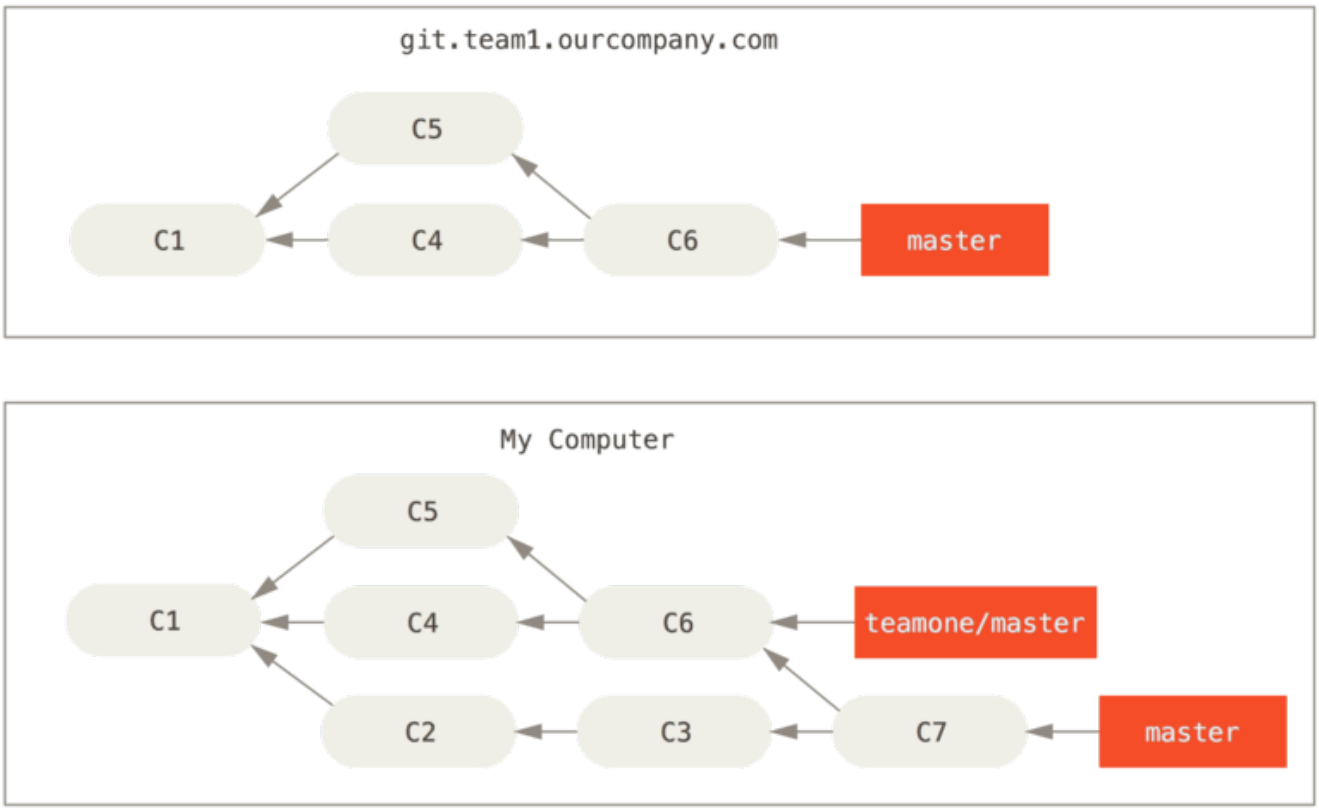



*Figure 45. さらなるコミットを取得し、作業環境にマージした状態*

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

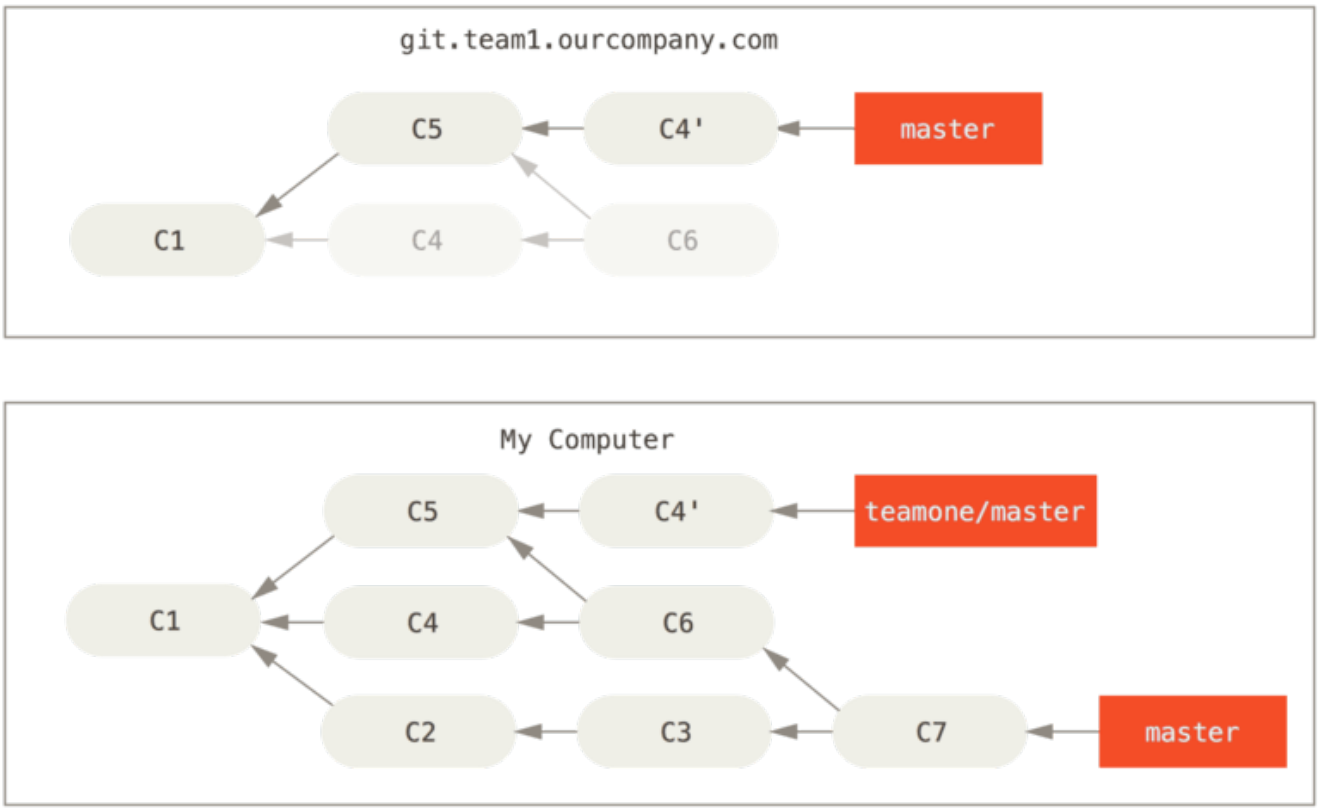



*Figure 46. 誰かがリベースしたコミットをプッシュし、あなたの作業環境の元になっているコミットが破棄された*

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

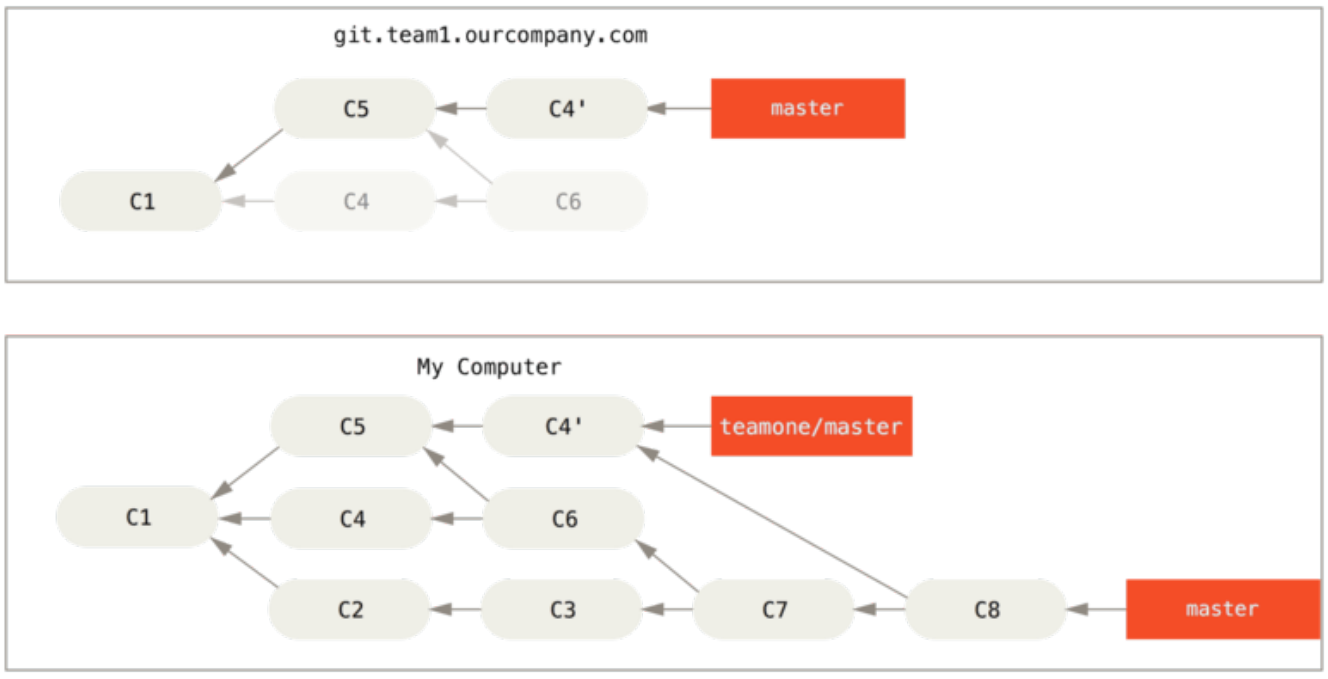



*Figure 47. 同じ作業を再びマージして新たなマージコミットを作成する*

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

## リベースした場合のリベース

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

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

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

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

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

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

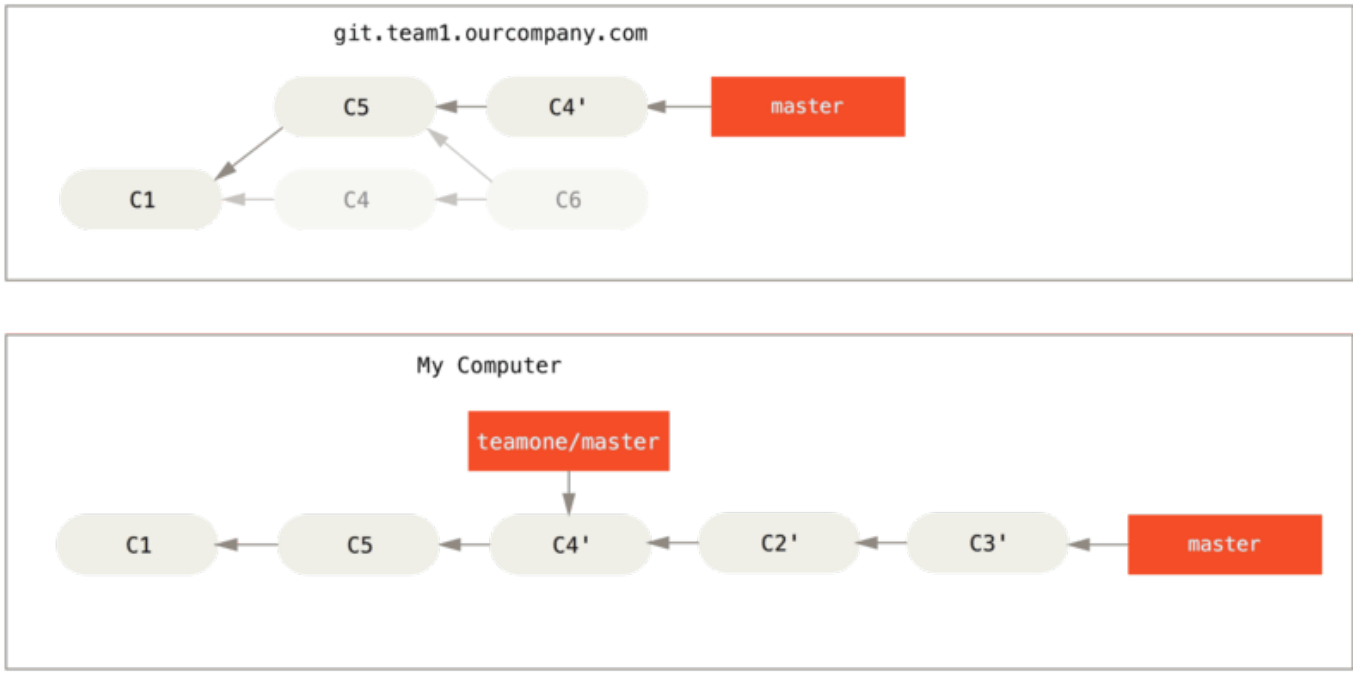



*Figure 48. リベース後、強制的にプッシュした作業へのリベース*

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

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

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

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

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

## リベースかマージか

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

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

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

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

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

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

# まとめ

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

# Gitサーバー

ここまで読んだみなさんは、ふだん Git を使う上で必要になるタスクのほとんどを身につけたことでしょう。しかし、Git で何らかの共同作業をしようと思えばリモートの Git リポジトリを持つ必要があります。 個人リポジトリとの間でのプッシュやプルも技術的には可能ですが、お勧めしません。よっぽど気をつけておかないと、ほかの人がどんな作業をしているのかをすぐに見失ってしまうからです。 さらに、自分のコンピューターがオフラインのときにもほかの人が自分のリポジトリにアクセスできるようにしたいとなると、共有リポジトリを持つほうがずっと便利です。 というわけで、他のメンバーとの共同作業をするときには、中間リポジトリをどこかに用意してみんながそこにアクセスできるようにし、プッシュやプルを行うようにすることをお勧めします。

Git サーバーを立ち上げるのは単純です。 まず、サーバーとの通信にどのプロトコルを使うのかを選択します。 この章の最初のセクションで、どんなプロトコルが使えるのかとそれぞれのプロトコルの利点・欠点を説明します。 その次のセクションでは、それぞれのプロトコルを使用したサーバーの設定方法とその動かし方を説明します。 最後に、ホスティングサービスについて紹介します。他人のサーバー上にコードを置くのが気にならない、そしてサーバーの設定だの保守だのといった面倒なことはやりたくないという人のためのものです。

自前でサーバーを立てることには興味がないという人は、この章は最後のセクションまで読み飛ばし、ホスティングサービスに関する情報だけを読めばよいでしょう。そして次の章に進み、分散ソース管理環境での作業について学びます。

リモートリポジトリは、一般的に *ベア（bare）リポジトリ* となります。これは、作業ディレクトリをもたない Git リポジトリのことです。 このリポジトリは共同作業の中継地点としてのみ用いられるので、ディスク上にスナップショットをチェックアウトする必要はありません。単に Git のデータがあればそれでよいのです。端的に言うと、ベアリポジトリとはそのプロジェクトの `.git` ディレクトリだけで構成されるもののことです。

## プロトコル

Git では、データ転送用のプロトコルとして Local、HTTP、Secure Shell (SSH)、Git の四つを使用できます。ここでは、それぞれがどんなものなのかとどんな場面で使うべきか (使うべきでないか) を説明します。

### Local プロトコル

一番基本的なプロトコルが *Local プロトコル* です。これは、リモートリポジトリをディスク上の別のディレクトリに置くものです。 これがよく使われるのは、たとえばチーム全員がアクセスできる共有ファイルシステム (NFS など) がある場合です。あるいは、あまりないでしょうが全員が同じコンピューターにログインしている場合にも使えます。 後者のパターンはあまりお勧めできません。すべてのコードリポジトリが同じコンピューター上に存在することになるので、何か事故が起こったときに何もかも失ってしまう可能性があります。

共有ファイルシステムをマウントしているのなら、それをローカルのファイルベースのリポジトリにクローン

したりお互いの間でプッシュやプルをしたりすることができます。
この手のリポジトリをクローンしたり既存のプロジェクトのリモートとして追加したりするには、リポジトリへのパスを URL に指定します。 たとえば、ローカルリポジトリにクローンするにはこのようなコマンドを実行します。

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

あるいは次のようにすることもできます。

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

URL の先頭に `file://` を明示するかどうかで、Git の動きは微妙に異なります。 `file://` を明示せずパスだけを指定した場合、Git は必要なオブジェクトにハードリンクを張るか、そのままコピーしようとします。 一方 `file://` を指定した場合は、Git がプロセスを立ち上げ、そのプロセスが (通常は) ネットワーク越しにデータを転送します。一般的に、直接のコピーに比べてこれは非常に非効率的です。 `file://` プレフィックスをつける最も大きな理由は、関係のない参照やオブジェクト(他のバージョン管理システムからインポートしたときなどにあらわれることが多いです。詳細は Gitの内側 を参照してください) を除いたクリーンなコピーがほしいということです。 本書では通常のパス表記を使用します。そのほうがたいていの場合に高速となるからです。

ローカルのリポジトリを既存の Git プロジェクトに追加するには、このようなコマンドを実行します。

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

そうすれば、このリモートとの間のプッシュやプルを、まるでネットワーク越しにあるのと同じようにすることができます。

**利点**

ファイルベースのリポジトリの利点は、シンプルであることと既存のファイルアクセス権やネットワークアクセスを流用できることです。 チーム全員がアクセスできる共有ファイルシステムがすでに存在するのなら、リポジトリを用意するのは非常に簡単です。 ベアリポジトリのコピーをみんながアクセスできるどこかの場所に置き、読み書き可能な権限を与えるという、ごく普通の共有ディレクトリ上での作業です。 この作業のために必要なベアリポジトリをエクスポートする方法については サーバー用の Git の取得 で説明します。

もうひとつ、ほかの誰かの作業ディレクトリの内容をすばやく取り込めるのも便利なところです。 同僚と作業しているプロジェクトで相手があなたに作業内容を確認してほしい言ってきたときなど、わざわざリモートのサーバーにプッシュしてもらってそれをプルするよりは単に `git pull /home/john/project` のようなコマンドを実行するほうがずっと簡単です。

**欠点**

この方式の欠点は、メンバーが別の場所にいるときに共有アクセスを設定するのは一般的に難しいということです。 自宅にいるときに自分のラップトップからプッシュしようとしたら、リモートディスクをマウントする必要があります。これはネットワーク越しのアクセスに比べて困難で遅くなるでしょう。

また、何らかの共有マウントを使用している場合は、必ずしもこの方式が最高速となるわけではありません。ローカルリポジトリが高速だというのは、単にデータに高速にアクセスできるからというだけの理由です。NFS 上に置いたリポジトリは、同じサーバーで稼動しているリポジトリに SSH でアクセスしたときよりも遅くなりがちです。SSH でアクセスしたときは、各システムのローカルディスクにアクセスすることになるからです。

もう1点、このプロトコルは「不慮の事故」を防ぐようにはできていない点も注意しておきましょう。 全ユーザーが接続先のディレクトリにシェルで自由にアクセスできるようになるため、Git 内部ファイルの変更・削除を防止することができないからです。仮にそういったことが起こると、リポジトリが破損してしまいます。

## HTTPプロトコル

HTTP を使って Git でやりとりをする場合、2つのモードが使えます。 以前のバージョンでは、単純で読み取り専用のモードしかありませんでした。 しかしバージョン 1.6.6 でより高機能なプロトコルが導入されました。これは、SSH の場合と同じように、HTTP でのデータのやりとりも Git が賢く処理できるようにするためのものでした。 ここ数年で、新しいほうの HTTP プロトコル はとても多く使われるようになりました。ユーザーからすればこちらのほうがシンプルですし、通信方法としても優れているからです。 新しいほうは “smart” HTTP プロトコルと呼ばれていて、古いほうは「ダム」（dumb）HTTP プロトコルと呼ばれています。 まずは “smart” HTTP プロトコルのほうから説明しましょう。

### Smart HTTP

“smart” HTTP プロトコルの動きは SSH や Git プロトコルと似ていますが、HTTP/S の標準ポートを使って通信します。また、HTTP 認証の仕組みをいくつも使うことができます。 よって、ユーザーにとっては SSH などよりも簡単であることが多いです。というのも、ユーザー名とパスワードを使ったベーシック認証を、SSH 鍵認証の代わりに使えるからです。

いまでは、Git で一番使われているのがこの方法だと思います。 というのも、 `git://` プロトコルが提供する匿名での読み込み機能と、SSH プロトコルが提供する認証・暗号化を経た書き込み機能の両方が、これひとつで実現できるからです。 これまでこういったことをするにはそれぞれに URL を用意する必要がありました。 いまでは、ひとつの URL で双方を実現できます。 プッシュしようとしたリポジトリで認証が必要であれば（通常であればそうすべきです）、サーバーはユーザー名とパスワードを要求することができます。 また、同じことが、読み込みについても言えます。

実際のところ、GitHub のようなサービスの場合、ブラウザでリポジトリを見るときに使う URL （“https://github.com/schacon/simplegit[]” など）と同じものを使って、リポジトリをクローンすることができます。書き込み権限があるなら、プッシュする場合も同じ URL が使えます。

### Dumb HTTP

Git の smart HTTP プロトコルにサーバーが応答しない場合、Git クライアントは簡易な “dumb” HTTP プロトコルへフォールバックします。 Dumb プロトコルでは、Git のベアリポジトリが通常のファイルと同じようにウェブサーバーから配信されます。 これのいいところは、セットアップがとても簡単な点です。 ベースとして必要になるのは、ベアリポジトリを HTTP ドキュメントのルートに配置することと、特定の `post-update` フックを設定することだけです（詳しくは Git フック を参照）。 それができれば、リポジトリを配置した

サーバーにアクセスできる人なら誰でも、そのリポジトリをクローンできます。 HTTP を使ったリポジトリへのアクセスは、以下のようにすると許可できます。

```
$ cd /var/www/htdocs/
$ git clone --bare /path/to/git_project gitproject.git
$ cd gitproject.git
$ mv hooks/post-update.sample hooks/post-update
$ chmod a+x hooks/post-update
```

これだけです。 Git に標準でついてくる `post-update` フックは、適切なコマンド (`git update-server-info`) を実行して HTTP でのフェッチとクローンをうまく動くようにします。 このコマンドが実行されるのは、このリポジトリに対して SSH 越しでのプッシュがあったときです。その他の人たちがクローンする際には次のようにします。

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

今回の例ではたまたま `/var/www/htdocs` (一般的な Apache の標準設定) を使用しましたが、別にそれに限らず任意のウェブサーバーを使うことができます。 単にベアリポジトリをそのパスに置けばよいだけです。 Git のデータは、普通の静的ファイルとして扱われます (実際のところどのようになっているかの詳細は Gitの内側 を参照ください)。

なお、構築するサーバーを選択する際は、読み書きが可能な Smart HTTP か、読み込み専用でファイルを配信する Dumb HTTP かのいずれかのサーバーを選ぶことになるでしょう。両方を混ぜあわせたものを構築することはめったにありません。

**利点**

ここでは、Smart HTTP プロトコルに限った範囲で利点について説明します。

ひとつの URL で全アクセスを処理できること、サーバーが入力を求めてくるのは認証が必要なときだけ、という2点があれば、エンドユーザーは Git をとても簡単に使えるようになります。 また、ユーザー名とパスワードを使って認証を受けられるのも、SSH と比べた場合大きな利点です。それができれば、SSH 鍵を生成したり公開鍵をサーバーへアップロードしなくても、サーバーとやりとりできるからです。 中〜上級者以外、または SSH があまり使われていないシステムのユーザーにとって、これはユーザビリティの点で大きな利点になります。 それに、HTTP プロトコルは SSH のようにとても早くて効率もよいです。

HTTPS で読み込み専用のリポジトリを公開することもできます。 これで、転送されるコンテンツを暗号化したりクライアント側で特定の署名つき SSL 証明書を使わせたりすることができるようになります。

もうひとつの利点としてあげられるのは、HTTP が非常に一般的なプロトコルであるということです。たいていの企業のファイアウォールはこのポートを通すように設定されています。

**欠点**

HTTP/S で Git を使えるようサーバーを設定するのは、SSH とは違ってやっかいなケースがあります。 それを除けば、他のプロトコルが “Smart” HTTP プロトコルより Git 用として優れてる点はほとんどありません。

上述のやっかいなケースとは、HTTP
を認証が必要なプッシュに用いるケースです。その場合、認証情報を入力するのは SSH 鍵を用いるより複雑になりがちです。 とはいえ、認証情報をキャッシュしてくれるツール（OSX の Keychain や Windows の資格情報マネージャーなど）を使えば、それも問題ではなくなります。認証情報の保存 を読めば、HTTP パスワードキャッシュを各システムで有効にする方法がわかるでしょう。

## SSH プロトコル

Git サーバーを自分でホスティングしているなら、転送プロトコルのうち一般的なのは SSH です。 SSH によるサーバーへのアクセスは、ほとんどの場面で既に用意されているからです。仮にまだ用意されていなかったとしても、導入するのは容易なことです。 SSH は認証付きのネットワークプロトコルでもあります。あらゆるところで用いられているので、環境を準備するのも容易です。

Git リポジトリを SSH 越しにクローンするには、次のように ssh:// URL を指定します。

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

あるいは、SCPコマンドのような省略形を使うこともできます。

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

ユーザー名も省略することもできます。その場合、Git は現在ログインしているユーザーでの接続を試みます。

### 利点

SSH を使う利点は多数あります。 まず、一般的に SSH 環境の準備は容易です。SSH デーモンはごくありふれたツールなので、ネットワーク管理者の多くはその使用経験があります。また、多くの OS に標準で組み込まれており、管理用ツールが付属しているものもあります。 さらに、SSH 越しのアクセスは安全です。すべての転送データは暗号化され、信頼できるものとなります。最後に、HTTP/S、Git、Local プロトコルと同程度に効率的です。転送するデータを可能な限りコンパクトにすることができます。

### 欠点

SSH の欠点は、リポジトリへの匿名アクセスを許可できないということです。 たとえ読み込み専用であっても、リポジトリにアクセスするには SSH 越しでのマシンへのアクセス権限が必要となります。つまり、オープンソースのプロジェクトにとっては SSH はあまりうれしくありません。 特定の企業内でのみ使用するのなら、SSH はおそらく唯一の選択肢となるでしょう。 あなたのプロジェクトに読み込み専用の匿名アクセスを許可しつつ自分は SSH を使いたい場合は、リポジトリへのプッシュ用に SSH を用意するのとは別にプル用の環境として別のプロトコルを提供する必要があります。

## Git プロトコル

次は Git プロトコルです。 これは Git に標準で付属する特別なデーモンです。専用のポート (9418) をリスンし、SSH プロトコルと同様のサービスを提供しますが、認証は行いません。 Git プロトコルを提供するリポジトリを準備するには、`git-daemon-export-ok` というファイルを作らなければなりません (このファイルがなければデーモンはサービスを提供しません)。ただ、このままでは一切セキュリティはありません。 Git リポジトリをすべての人に開放し、クローンさせることができます。 しかし、一般に、このプロトコルでプッシュさせることはありません。 プッシュアクセスを認めることは可能です。しかし認証がないということは、その URL を知ってさえいればインターネット上の誰もがプロジェクトにプッシュできるということになります。 これはありえない話だと言っても差し支えないでしょう。

**利点**

Git プロトコルは、もっとも高速なネットワーク転送プロトコルであることが多いです。 公開プロジェクトで大量のトラフィックをさばいている場合、あるいは巨大なプロジェクトで読み込みアクセス時のユーザー認証が不要な場合は、Git デーモンを用いてリポジトリを公開するとよいでしょう。 このプロトコルは SSH プロトコルと同様のデータ転送メカニズムを使いますが、暗号化と認証のオーバーヘッドがないのでより高速です。

**欠点**

Git プロトコルの弱点は、認証の仕組みがないことです。 Git プロトコルだけでしかプロジェクトにアクセスできないという状況は、一般的に望ましくありません。 SSH や HTTP と組み合わせ、プッシュ (書き込み) 権限を持つ一部の開発者には SSH を使わせてそれ以外の人には `git://` での読み込み専用アクセスを用意することになるでしょう。 また、Git プロトコルは準備するのがもっとも難しいプロトコルでもあります。 まず、独自のデーモンを起動しなければなりません。そのためには `xinetd` やそれに類するものの設定も必要になりますが、これはそんなにお手軽にできるものではありません。 また、ファイアウォールでポート 9418 のアクセスを許可する必要もあります。これは標準のポートではないので、企業のファイアウォールでは許可されていないかもしれません。 大企業のファイアウォールでは、こういったよくわからないポートは普通ブロックされています。

# サーバー用の Git の取得

さて、これまでに説明してきたプロトコルを使って Git サーバーを構築する方法を見ていきましょう。

**NOTE** ここで提示するコマンドや手順は、標準的な構成を Linux サーバーにインストールする場合のものです。また、これらは Mac や Windows のサーバーにも応用できます。 ただし、サーバーをプロダクション用にセットアップするときには、セキュリティの観点、OS のツール類などで違いが出るのは当然です。とはいえ、この節を読めば必要なものについて概ね把握できるでしょう。

Git サーバーを立ち上げるには、既存のリポジトリをエクスポートして新たなベアリポジトリ (作業ディレクトリを持たないリポジトリ) を作らなければなりません。 これは簡単にできます。 リポジトリをクローンして新たにベアリポジトリを作成するには、clone コマンドでオプション `--bare` を指定します。 慣例により、ベアリポジトリのディレクトリ名の最後は `.git` とすることになっています。

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

そうすると、Git ディレクトリのデータを `my_project.git` ディレクトリにコピーできます。

これは、おおざっぱに言うと次の操作と同じようなことです。

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

設定ファイルにはちょっとした違いもありますが、ほぼこんなものです。 作業ディレクトリなしで Git リポジトリを受け取り、それ単体のディレクトリを作成しました。

### ベアリポジトリのサーバー上への設置

ベアリポジトリを取得できたので、あとはそれをサーバー上においてプロトコルを準備するだけです。 ここでは、`git.example.com` というサーバーがあってそこに SSH でアクセスできるものと仮定しましょう。Git リポジトリはサーバー上の `/opt/git` ディレクトリに置く予定です。 `/opt/git` ディレクトリが作成済みであれば、新しいリポジトリを作成するには、ベアリポジトリを次のようにコピーします。

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

この時点で、同じサーバーに SSH でアクセスできてかつ `/opt/git` ディレクトリへの読み込みアクセス権限がある人なら、次のようにしてこのリポジトリをクローンできるようになりました。

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

ユーザーが SSH でアクセスでき、かつ `/opt/git/my_project.git` ディレクトリへの書き込みアクセス権限があれば、すでにプッシュもできる状態になっています。

`git init` コマンドで `--shared` オプションを指定すると、リポジトリに対するグループ書き込みパーミッションを自動的に追加することができます。

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

既存の Git リポジトリからベアリポジトリを作成し、メンバーが SSH でアクセスできるサーバーにそれを配置するだけ。簡単ですね。 これで、そのプロジェクトでの共同作業ができるようになりました。

複数名が使用する Git サーバーをたったこれだけの作業で用意できるというのは特筆すべきことです。 サーバーにSSHでアクセス可能なアカウントを作成し、ベアリポジトリをサーバーのどこかに置き、そこに読み書

き可能なアクセス権を設定する。 これで準備OK。他には何もいりません。

次のいくつかのセクションでは、より洗練された環境を作るための方法を説明します。いちいちユーザーごとにアカウントを作らなくて済む方法、一般向けにリポジトリへの読み込みアクセスを開放する方法、ウェブ UI の設定などです。しかし、数名のメンバーで閉じたプロジェクトでの作業なら、SSH サーバーとベアリポジトリ さえ あれば十分なことは覚えておきましょう。

### ちょっとしたセットアップ

小規模なグループ、あるいは数名の開発者しかいない組織で Git を使うなら、すべてはシンプルに進められます。 Git サーバーを準備する上でもっとも複雑なことのひとつは、ユーザー管理です。 同一リポジトリに対して「このユーザーは読み込みのみが可能、あのユーザーは読み書きともに可能」などと設定したければ、アクセス権とパーミッションの設定は、設定しない場合と比べて少しですが難しくなります。

#### SSH アクセス

開発者全員が SSH でアクセスできるサーバーがすでにあるのなら、リポジトリを用意するのは簡単です。先ほど説明したように、ほとんど何もする必要はないでしょう。 より複雑なアクセス制御をリポジトリ上で行いたい場合は、そのサーバーの OS 上でファイルシステムのパーミッションを設定するとよいでしょう。

リポジトリに対する書き込みアクセスをさせたいメンバーの中にサーバーのアカウントを持っていない人がいる場合は、新たに SSH アカウントを作成しなければなりません。 あなたがサーバーにアクセスできているということは、すでに SSH サーバーはインストールされているということです。

その状態で、チームの全員にアクセス権限を与えるにはいくつかの方法があります。 ひとつは全員分のアカウントを作成すること。直感的ですがすこし面倒です。 ひとりひとりに対して `adduser` を実行して初期パスワードを設定するという作業をしなければなりません。

もうひとつの方法は、*git* ユーザーをサーバー上に作成し、書き込みアクセスが必要なユーザーには SSH 公開鍵を用意してもらってそれを *git* ユーザーの `~/.ssh/authorized_keys` に追加します。 これで、全員が *git* ユーザーでそのマシンにアクセスできるようになりました。これがコミットデータに影響を及ぼすことはありません。 SSH で接続したときのユーザーとコミットするときに記録されるユーザーとは別のものだからです。

あるいは、SSH サーバーの認証を LDAP サーバーやその他の中央管理形式の仕組みなど既に用意されているものにするとこもできます。 各ユーザーがサーバー上でシェルへのアクセスができさえすれば、どんな仕組みの SSH 認証であっても動作します。

## SSH 公開鍵の作成

多くの Git サーバーでは、SSH の公開鍵認証を使用しています。 この方式を使用するには、各ユーザーが自分の公開鍵を作成しなければなりません。 公開鍵のつくりかたは、OS が何であってもほぼ同じです。 まず、自分がすでに公開鍵を持っていないかどうか確認します。 デフォルトでは、各ユーザーの SSH 鍵はそのユーザーの `~/.ssh` ディレクトリに置かれています。 自分が鍵を持っているかどうかを確認するには、このディレクトリに行ってその中身を調べます。

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

そして、id_dsa あるいは id_rsa というファイルと、同名で .pub という拡張子を持つファイルの組み合わせを探します。もし見つかったら、.pub がついているほうのファイルがあなたの公開鍵で、もう一方があなたの秘密鍵です。 そのようなファイルがない (あるいはそもそも .ssh ディレクトリがない) 場合は、ssh-keygen というプログラムを実行してそれを作成します。このプログラムは Linux/Mac なら SSH パッケージに含まれており、Windows では Git for Windows に含まれています。

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

まず、鍵の保存先 (.ssh/id_rsa) を指定し、それからパスフレーズを二回入力するよう求められます。鍵を使うときにパスフレーズを入力したくない場合は、パスフレーズを空のままにしておきます。

さて、次に各ユーザーは自分の公開鍵をあなた (あるいは Git サーバーの管理者である誰か) に送らなければなりません (ここでは、すでに公開鍵認証を使用するように SSH サーバーが設定済みであると仮定します)。 公開鍵を送るには、.pub ファイルの中身をコピーしてメールで送ります。 公開鍵は、このようなファイルになります。

```
$ cat ~/.ssh/id_rsa.pub
ssh-rsa AAAAB3NzaC1yc2EAAAABIwAAAQEAklOUpkDHrfHY17SbrmTIpNLTGK9Tjom/BWDSU
GPl+nafzlHDTYW7hdI4yZ5ew18JH4JW9jbhUFrviQzM7xlELEVf4h9lFX5QVkbPppSwg0cda3
Pbv7kOdJ/MTyBlWXFCR+HAo3FXRitBqxiX1nKhXpHAZsMciLq8V6RjsNAQwdsdMFvSlVK/7XA
t3FaoJoAsncM1Q9x5+3V0Ww68/eIFmb1zuUFljQJKprrX88XypNDvjYNby6vw/Pb0rwert/En
mZ+AW4OZPnTPI89ZPmVMLuayrD2cE86Z/il8b+gw3r3+1nKatmIkjn2so1d01QraTlMqVSsbx
NrRFi9wrf+M7Q== schacon@mylaptop.local
```

各種 OS 上での SSH 鍵の作り方については、GitHub の https://help.github.com/articles/generating-ssh-keys に詳しく説明されています。

## サーバーのセットアップ

それでは、サーバー側での SSH アクセスの設定について順を追って見ていきましょう。 この例では

authorized_keys 方式でユーザーの認証を行います。 また、Ubuntu のような標準的な Linux ディストリビューションを動かしているものと仮定します。 まずは git ユーザーを作成し、そのユーザーの .ssh ディレクトリを作りましょう。

```
$ sudo adduser git
$ su git
$ cd
$ mkdir .ssh && chmod 700 .ssh
$ touch .ssh/authorized_keys && chmod 600 .ssh/authorized_keys
```

次に、開発者たちの SSH 公開鍵を git ユーザーの authorized_keys に追加していきましょう。 信頼できる公開鍵が一時ファイルとしていくつか保存されているものとします。 先ほどもごらんいただいたとおり、公開鍵の中身はこのような感じになっています。

```
$ cat /tmp/id_rsa.john.pub
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQCB007n/ww+ouN4gSLKssMxXnBOvf9LGt4L
ojG6rs6hPB09j9R/T17/x4lhJA0F3FR1rP6kYBRsWj2aThGw6HXLm9/5zytK6Ztg3RPKK+4k
Yjh6541NYsnEAZuXz0jTTyAUfrtU3Z5E003C4oxOj6H0rfIF1kKI9MAQLMdpGW1GYEIgS9Ez
Sdfd8AcCIicTDWbqLAcU4UpkaX8KyGlLwsNuuGztobF8m72ALC/nLF6JLtPofwFBlgc+myiv
O7TCUSBdLQlgMVOFq1I2uPWQOkOWQAHukEOmfjy2jctxSDBQ220ymjaNsHT4kgtZg2AYYgPq
dAv8JggJICUvax2T9va5 gsg-keypair
```

これを、git ユーザーの .ssh ディレクトリにある authorized_keys に追加していきましょう。

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

さて、彼らが使うための空のリポジトリを作成しましょう。git init に --bare オプションを指定して実行すると、作業ディレクトリのない空のリポジトリを初期化します。

```
$ cd /opt/git
$ mkdir project.git
$ cd project.git
$ git init --bare
Initialized empty Git repository in /opt/git/project.git/
```

これで、John と Josie そして Jessica はプロジェクトの最初のバージョンをプッシュできるようになりました。このリポジトリをリモートとして追加し、ブランチをプッシュすればいいのです。 何か新しいプロジェクトを追加しようと思ったら、そのたびに誰かがサーバーにログインし、ベアリポジトリを作らなければならないことに注意しましょう。 git ユーザーとリポジトリを作ったサーバーのホスト名を gitserver としておきましょう。 gitserver がそのサーバーを指すように DNS を設定しておけば、このようなコマンドを使えます（ここでは、myproject というディレクトリがあってファイルも保存されているものとします）。

```
# on John's computer
$ cd myproject
$ git init
$ git add .
$ git commit -m 'initial commit'
$ git remote add origin git@gitserver:/opt/git/project.git
$ git push origin master
```

これで、他のメンバーがリポジトリをクローンして変更内容を書き戻せるようになりました。

```
$ git clone git@gitserver:/opt/git/project.git
$ cd project
$ vim README
$ git commit -am 'fix for the README file'
$ git push origin master
```

この方法を使えば、小規模なチーム用の読み書き可能な Git サーバーをすばやく立ち上げることができます。

この時点では、公開鍵を追加してもらったユーザー全員が `git` ユーザーとしてサーバーにログインしてシェルが使える状態であることに注意しましょう。そこを制限したいのなら、シェルを変更するために `passwd` ファイルを編集する必要があります。

`git` ユーザー権限の制限は簡単です。Git に付属している `git-shell` というツールを使えば、Git 関連の行動しかとれないようになります。 そして、これを `git` ユーザーのログインシェルにしてしまえば、サーバー上で `git` ユーザーは通常の行動がとれなくなります。 ユーザーのログインシェルを bash や csh から `git-shell` に変更すれば、制限がかかります。 それには、前もって `git-shell` を `/etc/shells` に追加しておく必要があります。

```
$ cat /etc/shells   # see if `git-shell` is already in there.  If not...
$ which git-shell   # make sure git-shell is installed on your system.
$ sudo vim /etc/shells  # and add the path to git-shell from last command
```

ユーザーのシェルを変更するには `chsh <username>` を実行します。

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

これで、`git` ユーザーは Git リポジトリへのプッシュやプル以外のシェル操作ができなくなりました。それ以外の操作をしようとすると、このように拒否されます。

```
$ ssh git@gitserver
fatal: Interactive git shell is not enabled.
hint: ~/git-shell-commands should exist and have read and execute access.
Connection to gitserver closed.
```

この状態でも Git のネットワーク関連のコマンドは機能しますが、通常のシェルアクセスはできなくなっています。 また、コマンド出力にもあるように、`git` ユーザーのホームディレクトリ配下にディレクトリを作って、`git-shell`をカスタマイズすることもできます。 具体的には、サーバー上で実行可能な Git コマンドの制限や、ユーザーが SSH でどこかに接続しようとしたときに表示するメッセージを変更できます。 `git help shell` を実行すると、シェルのカスタマイズについての詳細が確認できます。

## Git デーモン

続いて、“Git” プロトコルを使ってリポジトリを配信するデーモンをセットアップしてみましょう。 Git リポジトリへの認証なしの高速アクセスが必要な場合、一般的な選択肢になります。 ただし、これは認証なしのサービスで、配信されるデータは原則として公開されてしまうので注意してください。

ファイアウォールの外にサーバーがあるのなら、一般に公開しているプロジェクトにのみ使うようにしましょう。ファイアウォール内で使うのなら、たとえば大量のメンバーやコンピューター (継続的インテグレーションのビルドサーバーなど) に対して SSH の鍵なしで読み取り専用アクセスを許可するという使い方もあるでしょう。

いずれにせよ、Git プロトコルは比較的容易にセットアップすることができます。 デーモン化するためには、このようなコマンドを実行します。

```
$ git daemon --reuseaddr --base-path=/opt/git/ /opt/git/
```

`--reuseaddr` は、前の接続がタイムアウトするのを待たずにサーバーを再起動させるオプションです。 `--base-path` オプションを指定すると、フルパスを指定しなくてもプロジェクトをクローンできるようになります。そして最後に指定したパスは、Git デーモンに公開させるリポジトリの場所です。 ファイアウォールを使っているのなら、ポート 9418 に穴を開けなければなりません。

プロセスをデーモンにする方法は、OS によってさまざまです。 Ubuntu の場合は Upstart スクリプトを使います。

```
/etc/init/local-git-daemon.conf
```

のようなファイルを用意して、このようなスクリプトを書きます。

```
start on startup
stop on shutdown
exec /usr/bin/git daemon \
    --user=git --group=git \
    --reuseaddr \
    --detach \
    --base-path=/opt/git/ \
    /opt/git/
respawn
```

セキュリティを考慮して、リポジトリに対する読み込み権限しかないユーザーでこのデーモンを実行させるようにしましょう。新しいユーザー *git-ro* を作り、このユーザーでデーモンを実行させるとよいでしょう。 ここでは、説明を簡単にするために `git-shell` と同じユーザー *git* で実行させることにします。

マシンを再起動すれば Git デーモンが自動的に立ち上がり、終了させても再び起動するようになります。 再起動せずに実行させるには、次のコマンドを実行します。

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

その他のシステムでは、`xinetd` や `sysvinit` システムのスクリプトなど、コマンドをデーモン化して監視できる仕組みを使います。

次に、どのプロジェクトに対して Git プロトコルでの認証なしアクセスを許可するのかを Git に設定します。許可したいリポジトリに `git-deamon-export-ok` ファイルを作成すれば設定できます。

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

このファイルが存在するプロジェクトについては、Git は認証なしで公開してもよいものとみなします。

# Smart HTTP

これまでの説明で、SSH を使った認証ありのプロトコルと `git://` を使った認証なしのプロトコルについてわかったと思います。続いて、それら両方を実現してしまうプロトコルについて説明しましょう。 Smart HTTP のセットアップは、単に CGI スクリプトをひとつ、Git サーバー上で有効にすればおしまいです。Git に同梱されている `git-http-backend` というスクリプトを使います。 この CGI は、パスやヘッダー情報（`git fetch` や `git push` で特定の HTTP URL 宛に送られてきたデータ）を読み込み、クライアントが HTTP を使ってやりとりできるかどうか判断します（バージョン 1.6.6 以降の Git クライアントであれば対応しています）。 そして、CGI の判断が「このクライアントは Smart HTTP に対応している」だった場合は Smart HTTP が使われ、そうでなかった場合はリードオンリー（“dumb”）にフォールバックします（後方互換という意味では、読み込みについては古いクライアントにも対応しています）。

では、標準的なセットアップ方法について説明しましょう。ここでは、Apache を CGI サーバーとして使います。Apache がインストールされていない場合は、Linux サーバー上で以下のようなコマンドを実行してください。

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

そうすれば、`mod_cgi`、`mod_alias`、`mod_env`、`mod_rewrite` も有効になります。いずれも、Smart HTTP の動作に必要なものです。

また、`/opt/git` ディレクトリのグループを `www-data` に変更しなければなりません。CGIスクリプトを実行するApacheのインスタンスはデフォルトではそのグループの1ユーザーとして実行されるからです。設定を変更しておけば、ウェブサーバーは自由にリポジトリを読み書きできるようになります。

```
$ chgrp -R www-data /opt/git
```

次に、Apache の設定をします。`git-http-backend` をハンドラにして、ウェブサーバーの `/git` パスにアクセスがあった場合にそれに処理させるための設定です。

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

環境変数 `GIT_HTTP_EXPORT_ALL` を設定しない場合、クライアントからのアクセスは読み込み専用になり、読み込めるのは `git-daemon-export-ok` ファイルが保存されたリポジトリだけになります。Git デーモンと同様の挙動です。

最後に、Apacheの設定を2つ変更します。 `git-http-backend` へのアクセスを許可する設定と、書き込みを認証するための設定です。Auth ブロックを使う場合、以下のようにして設定できます。

```
RewriteEngine On
RewriteCond %{QUERY_STRING} service=git-receive-pack [OR]
RewriteCond %{REQUEST_URI} /git-receive-pack$
RewriteRule ^/git/ - [E=AUTHREQUIRED]

<Files "git-http-backend">
    AuthType Basic
    AuthName "Git Access"
    AuthUserFile /opt/git/.htpasswd
    Require valid-user
    Order deny,allow
    Deny from env=AUTHREQUIRED
    Satisfy any
</Files>
```

さらに、対象ユーザー全員のパスワードが記述された `.htaccess` ファイルが必要です。ユーザー “schacon” を追加したい場合は、このようなコマンドを実行します。

```
$ htpasswd -c /opt/git/.htpasswd schacon
```

ユーザー認証を Apache で実施する方法はたくさんあります。 ひとつ選んで設定してください。 ここでは、思いつく限り一番シンプルな方法を説明しました。 また、HTTP 通信が SSL 経由で行われるように設定しましょう。 そうすれば、データはすべて暗号化されます。

ここでは、Apache 設定の詳細についてはあえて立ち入らないようにしました。 Apache 以外の ウェブサーバーを使う場合もあるでしょうし、認証の要求も多様だからです。 覚えておいてほしいのは、Git には `git-http-backend` という CGI スクリプトが付属していることです。 それが実行されると、HTTP 経由でデータを送受信する際のネゴシエーションを処理してくれます。 このスクリプト自体は認証の仕組みを備えてはいませんが、ウェブサーバーの機能で認証は簡単に管理できます。 CGI に対応している ウェブサーバーであればどれも使っても構いません。一番使い慣れたものを使うのがよいでしょう。

**NOTE** Apacheを使った認証設定の詳細については、Apache の公式ドキュメント http://httpd.apache.org/docs/current/howto/auth.html を参照してください。

## GitWeb

これで、読み書き可能なアクセス方法と読み込み専用のアクセス方法を用意できるようになりました。次にほしくなるのは、ウェブベースでの閲覧方法でしょうか。 Git には標準で GitWeb という CGI スクリプトが付属しており、これを使うことができます。

自分のプロジェクトでためしに GitWeb を使ってみようという人のために、一時的なインスタンスを立ち上げるためのコマンドが Git に付属しています。これを実行するには `lighttpd` や `webrick` といった軽量なサーバーが必要です。 Linux マシンなら、たいてい `lighttpd` がインストールされています。これを実行するに

は、プロジェクトのディレクトリで `git instaweb` と打ち込みます。 Mac の場合なら、Leopard には Ruby がプレインストールされています。したがって `webrick` が一番よい選択肢でしょう。`instaweb` を lighttpd 以外で実行するには、`--httpd` オプションを指定します。

```
$ git instaweb --httpd=webrick
[2009-02-21 10:02:21] INFO  WEBrick 1.3.1
[2009-02-21 10:02:21] INFO  ruby 1.8.6 (2008-03-03) [universal-darwin9.0]
```

これは、HTTPD サーバーをポート 1234 で起動させ、自動的にウェブブラウザーを立ち上げてそのページを表示させます。 非常にお手軽です。 ひととおり見終えてサーバーを終了させたくなったら、同じコマンドに `--stop` オプションをつけて実行します。

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

ウェブインターフェイスをチーム内で常時立ち上げたりオープンソースプロジェクト用に公開したりする場合は、CGI スクリプトを設定して通常のウェブサーバーに配置しなければなりません。 Linux のディストリビューションの中には、`apt` や `yum` などで `gitweb` パッケージが用意されているものもあります。まずはそれを探してみるとよいでしょう。 手動での GitWeb のインストールについて、さっと流れを説明します。 まずは Git のソースコードを取得しましょう。その中に GitWeb が含まれており、CGI スクリプトを作ることができます。

```
$ git clone git://git.kernel.org/pub/scm/git/git.git
$ cd git/
$ make GITWEB_PROJECTROOT="/opt/git" prefix=/usr gitweb
    SUBDIR gitweb
    SUBDIR ../
make[2]: `GIT-VERSION-FILE' is up to date.
    GEN gitweb.cgi
    GEN static/gitweb.js
$ sudo cp -Rf gitweb /var/www/
```

コマンドを実行する際に、Git リポジトリの場所を `GITWEB_PROJECTROOT` 変数で指定しなければならないことに注意しましょう。 さて、次は Apache にこのスクリプトを処理させるようにしなければなりません。VirtualHost に次のように追加しましょう。

```
<VirtualHost *:80>
    ServerName gitserver
    DocumentRoot /var/www/gitweb
    <Directory /var/www/gitweb>
        Options ExecCGI +FollowSymLinks +SymLinksIfOwnerMatch
        AllowOverride All
        order allow,deny
        Allow from all
        AddHandler cgi-script cgi
        DirectoryIndex gitweb.cgi
    </Directory>
</VirtualHost>
```

GitWeb は、CGI か Perl に対応したウェブサーバーならどんなものを使っても動かすことができます。何か別のサーバーのほうがよいというのなら、そのサーバーで動かすのもたやすいことでしょう。これで、`http://gitserver/` にアクセスすればリポジトリをオンラインで見られるようになりました。

# GitLab

前節で紹介した GitWeb はとてもシンプルでした。 もし、もっとモダンで高機能な Git サーバーが必要な場合には、他のオープンソース・ソフトウェアが選択肢になるでしょう。 それらのなかでも GitLab はよく使われていますので、一例として紹介します。 GitWeb よりも複雑でメンテナンスも必要ではありますが、GitLab はより高機能な選択肢です。

## インストール

GitLab はデータベースを使用する Web アプリケーションです。そのため、インストール方法は他の Git サーバーより複雑になってしまいます。 とはいえ、幸いなことにドキュメントは充実していて、かつ手順は簡素化されています。

GitLab は数種類の方法でインストールできます。 とりあえず動かしてみるには、仮想マシンのイメージ、もしくはワンクリックインストーラーを使います。 <a href="https://bitnami.com/stack/gitlab" class="bare">https://bitnami.com/stack/gitlab</a> からそれらのファイルをダウンロード・インストールし、使用する環境に応じて設定を変更しましょう。 <a name="__indexterm-47168695884600" type="indexterm"></a> この方法では、Bitnami が気を利かせてログイン画面を使えるようにしてくれています（alt-→ と入力すると表示されます）。インストールした GitLab 用の IP アドレス・ユーザー名・パスワードを表示してくれる便利な画面です。


*Figure 49. Bitnami GitLab 仮想マシンのログイン画面*

その他の方法については、GitLab Community Edition の README を参照してください。 https://gitlab.com/gitlab-org/gitlab-ce/tree/master で確認できます。 そこで紹介されている GitLab のインストール方法には、Chef のレシピを使う方法、Digital Ocean で仮想マシンを用いる方法、RPM や DEB のパッケージを用いる方法（執筆段階ではベータ）などがあります。 その他にも “非公式” のガイドとして、サポート外の OS やデータベースで GitLab を動かす方法、手動でインストールを行うためのスクリプトなど、多くのトピックが紹介されています。

## GitLab の管理

GitLab の管理画面はブラウザでアクセスします。 ブラウザを使って GitLab をインストールしたサーバーのホスト名か IP アドレスにアクセスし、管理者としてログインしましょう。 デフォルトのユーザー名は `admin@local.host` で、デフォルトのパスワードは `5iveL!fe` です（この組み合わせでログインすると、パスワードを変更することを最初に促されます）。 ログインしたら、 画面右上のメニューにある “Admin area” のアイコンをクリックしましょう。

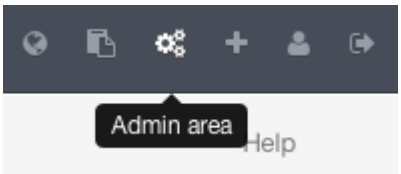



*Figure 50. GitLab メニューにある “Admin area” の項目*

### ユーザー

GitLab におけるユーザーは、使用者に紐付けられたアカウントのことを指します。 それは複雑なものではありません。メインはログイン情報ごとに登録された個人情報です。 また、ユーザーアカウントには **名前空間** が設定されていて、ユーザーが保持するプロジェクトの識別子として用いられます。 たとえば、ユーザー

`jane` が `project` という名前のプロジェクトを保持していた場合は、そのプロジェクトの URL は http://server/jane/project になります。

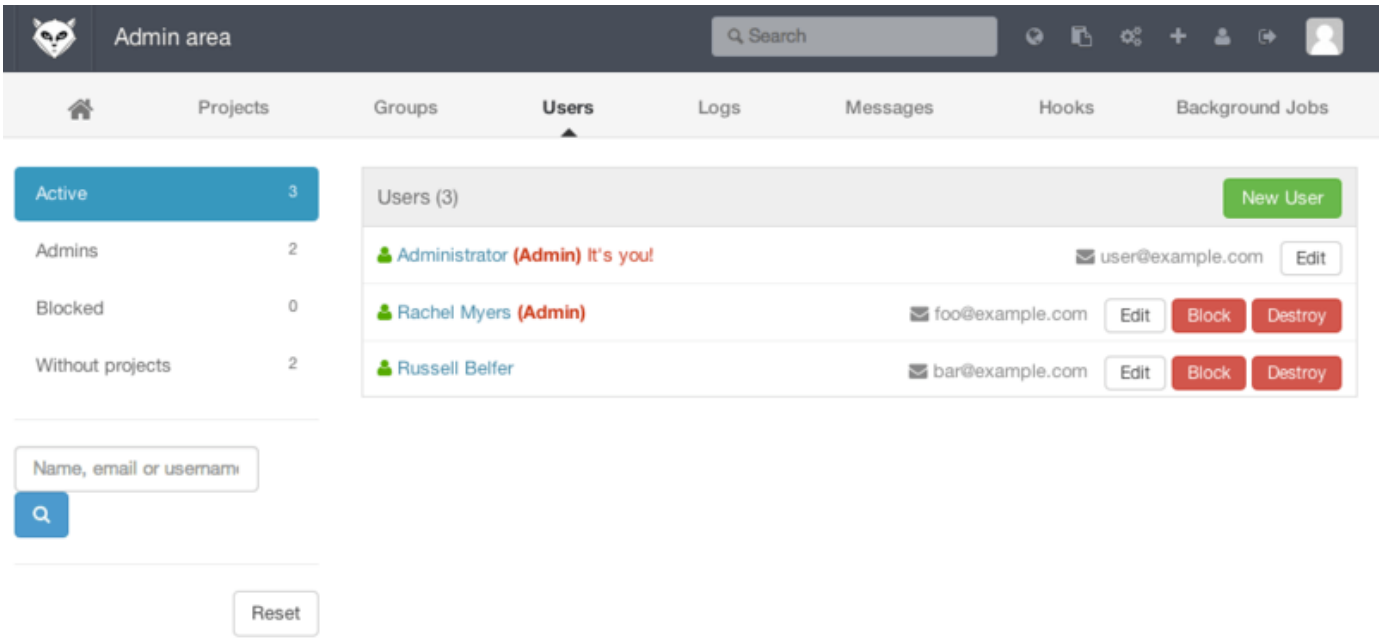



*Figure 51. GitLab ユーザー管理画面*

ユーザーを削除する場合、やり方は2つです。 まずはユーザーを「ブロック」する方法です。この場合、ブロックされたユーザーは GitLab にはログインできなくなります。一方、ユーザーの名前空間配下のデータは削除されず、ユーザーのメールアドレスで署名されたコミットとユーザープロファイルとの関連付けも有効なままになります。

もうひとつのやり方はユーザーを「破壊」する方法です。ユーザーを破壊すると、GitLab のデータベース、ファイルシステムから削除されます。 ユーザーの名前空間配下のデータ・プロジェクトも削除されますし、そのユーザーが作成したグループも同じように削除されます。 この方法は「ブロック」と比べはるかに恒久的でやり直しがきかないものです。よって、使われることはめったにありません。

### グループ

GitLab では、複数のプロジェクトをグループとしてまとめられます。そして、グループにはプロジェクトごとのユーザーのアクセス権情報も含まれています。 また、ユーザーの場合と同じように、それぞれのグループには名前空間があります。たとえば、 `training` グループに `materials` というプロジェクトがあった場合、URL は http://server/training/materials になります。

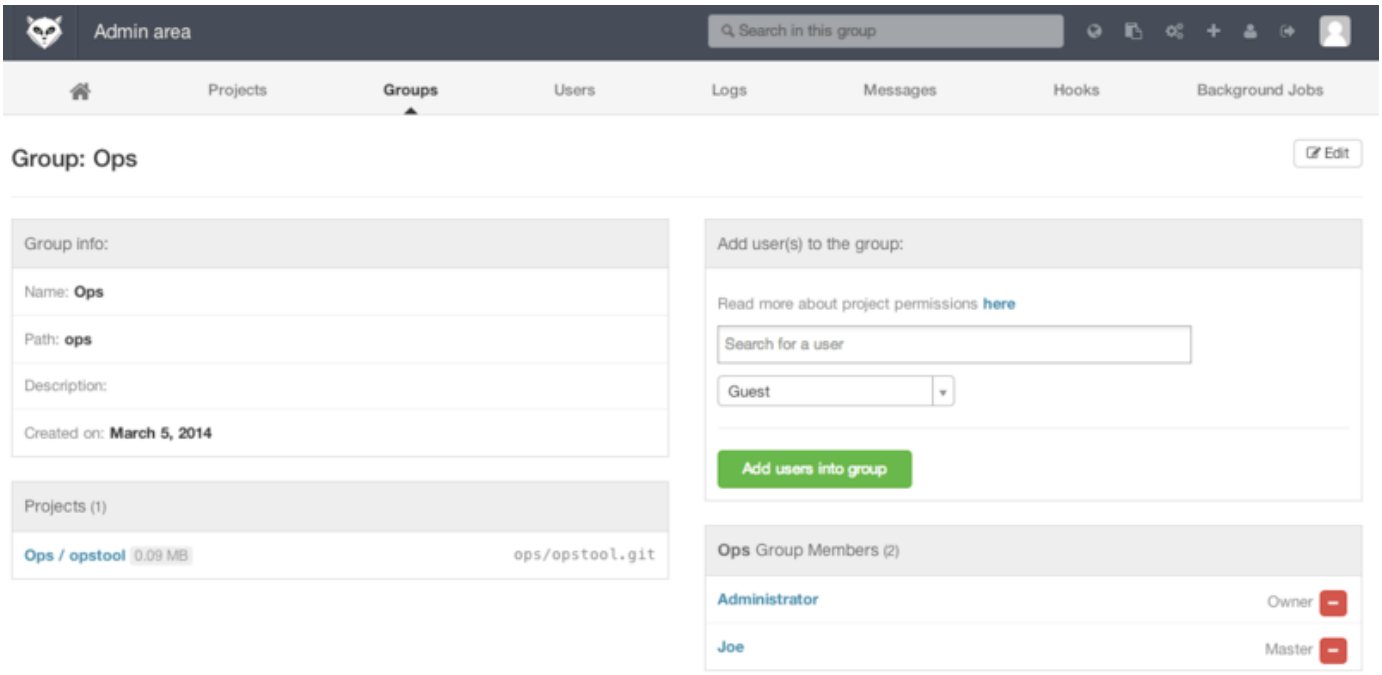



*Figure 52. GitLab グループ管理画面*

グループにはメンバーを追加できます。さらに、メンバーそれぞれにプロジェクトごとの権限と、グループそのものに対する権限が設定できます。 権限は “Guest” （issues とチャットのみ）から “Owner” （グループと配下のメンバー・プロジェクトの管理権限）までが用意されています。 各権限の詳細は膨大なため省略しますが、管理画面にあるリンクを辿ると確認できるようになっています。

### プロジェクト

GitLab のプロジェクトとは、大体の場合ひとつの Git のリポジトリを指します。 なんらかの名前空間（ユーザーかグループ）に、プロジェクトはすべて属しています。 プロジェクトがユーザーの名前空間に属していれば、そのオーナーはアクセス権をすべて管理できます。プロジェクトがグループに属していれば、グループ自体のアクセス権設定がプロジェクトにも引き継がれます。

また、プロジェクトには公開レベル設定があり、どのユーザーにプロジェクトのページとリポジトリの読み込み権限を与えるかを設定できます。 プロジェクトが *Private* の場合、アクセス権をユーザーごとに明示的に設定する必要があります。 プロジェクトが *Internal* の場合はログイン済みのユーザーであれば閲覧でき、*Public* なプロジェクトであれば誰でも閲覧できます。 なお、この設定で、Git の “fetch” コマンドと ウェブ画面の両方のアクセス権が設定されることに注意しましょう。

### フック

GitLab はフック（プロジェクト・システムの両方）に対応しています。 どちらのフックであれ、該当のイベントが発生した都度、GitLab のサーバーは JSON データを使って HTTP POST リクエストを発行します。 これは、Git リポジトリや GitLab を開発自動化の仕組みと連携させるときにとても便利です。特に CI サーバー・チャットサービス・デプロイ用のツールなどとの連携に役立つでしょう。

## 基本的な使い方

GitLab で最初にやるべきことは、新規プロジェクトの作成です。 ツールバーの “+” アイコンをクリックすると作成が始まります。 まず、プロジェクト名称・所属すべき名前空間・公開レベル設定を作成時に入力しま

す。 それらの大半は設定画面からあとから変更可能です。 次に “Create Project” をクリックすれば、プロジェクトが作成されます。

プロジェクトが作成されたら、まずは手元の Git リポジトリとそのプロジェクトを関連付けるとよいでしょう。 プロジェクトにアクセスするには HTTPS か SSH を使います。いずれも、Git のリモートとして設定可能なプロトコルです。 設定用の URL は、プロジェクトのページの最上部に表示されています。

```
$ git remote add gitlab https://server/namespace/project.git
```

手元に Git リポジトリがない場合は、以下のようにしてリモートからクローンしましょう。

```
$ git clone https://server/namespace/project.git
```

GitLab の ウェブ画面には、リポジトリの情報を確認する便利な画面がいくつもあります。 プロジェクトのトップページでは最近の行動履歴が確認できますし、画面上部にあるリンクをたどるとファイル一覧やコミットログを確認できます。

## 共同作業

GitLab でホストしているプロジェクトで共同作業を行うもっともシンプルな方法は、Git リポジトリへのプッシュアクセス権を相手に与えてしまうことです。 ユーザーをプロジェクトに追加するには、プロジェクトの設定画面にある “Members” のセクションを使います。新規ユーザーにアクセス権を設定するのにも同じセクションを使います（アクセス権の詳細については、グループ でも少し触れました）。 ユーザーに付与された権限が “Developer” かそれ以上であれば、リポジトリへコミットやブランチを問題なく直接プッシュできます。

もうひとつ、より疎結合な共同作業の方法があります。マージリクエストです。 この機能を使えば、任意のユーザー（プロジェクトを閲覧可能なユーザーに限られます）に所定の方法で共同作業に参加してもらえます。 まず、リポジトリに直接プッシュする権限のあるユーザーの場合は、ブランチを作ってコミットをプッシュしたうえで、そのブランチから `master` など希望するブランチに向けてマージリクエストを作成します。一方、プッシュ権限のないユーザーの場合、まずはリポジトリを「フォーク」（自分専用のコピーを作成）します。続いて *その* コピーにコミットをプッシュしたら、コピーから本家に向けてマージリクエストを作成しましょう。 この方法を使えば、どんな変更がいつリポジトリに追加されるかを管理者が管理しつつ、任意のユーザーに共同作業に参加してもらえます。

GitLab においては、マージリクエストや issue を使って議論を深めていきます。 マージリクエストは、変更内容について行ごとに議論すること（簡易的なコードレビュー機能としても使えます）にも使えますし、マージリクエスト全体についての議論のスレッドとしても使えます。 また、マージリクエストや issue には担当者を設定できますし、マイルストーンの一部にもなります。

この節では主に GitLab の Git 関連部分を見てきました。ただ、GitLab はとても完成度の高いシステムで、共同作業に役立つ機能は他にもたくさんあります。たとえば、プロジェクト用の wiki やシステム管理ツールなどです。 最後に GitLab の利点としてもう一点挙げておきましょう。GitLab は、一度セットアップが終わってサーバーが動き出せば、設定ファイルをいじったりサーバーに SSH でログインしたりする必要はほとんどありません。管理作業、そして通常利用の大半は、ブラウザ画面から操作できます。

## サードパーティによる Git ホスティング

色々と苦労してまで自分用 Git サーバーを立てようとは思わない、という場合は、Git 専用のホスティングサービスに Git のリポジトリを預けられます。 そうすれば、初期セットアップはすぐ終わり、簡単にプロジェクトに着手できます。また、サーバー保守や監視の必要もありません。 仮に内部用に自分用のサーバーを運用していたとしても、オープンソースのコードをホストするにはホスティングサービスの公開リポジトリを使うといいでしょう。そうすれば、リポジトリは見つかりやすく、オープンソースコミュニティの助けも得やすくなります。

最近では、数多くのホスティングサービスが存在していて、それぞれに長所・短所があります。 ホスティングサービス一覧の最新版は、Git wiki の GitHosting のページ https://git.wiki.kernel.org/index.php/GitHosting を確認してください。

なお、GitHub の使い方を GitHub で詳しく説明します。なぜなら、GitHub は最大のGit ホスティングサービスで、関わりあいを持とうとしたプロジェクトが GitHub にホストされていることも十分あり得るからです。とはいえ、Git サーバーを自らセットアップしたくないなら、選択肢はたくさんあります。

## まとめ

リモート Git リポジトリを用意するためのいくつかの方法を紹介し、他のメンバーとの共同作業ができるようになりました。

自前でサーバーを構築すれば、多くのことを制御できるようになり、ファイアウォールの内側でもサーバーを実行することができます。 しかし、サーバーを構築して運用するにはそれなりの手間がかかります。ホスティングサービスを使えば、サーバーの準備や保守は簡単になります。 しかし、他人のサーバー上に自分のコードを置き続けなければなりません。組織によってはそんなことを許可していないかもしれません。

どの方法 (あるいは複数の方法の組み合わせ) を使えばいいのか、自分や所属先の事情に合わせて考えましょう。

# Git での分散作業

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

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

## 分散作業の流れ

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

### 中央集権型のワークフロー

中央管理型のシステムでは共同作業の方式は一つだけです。それが中央集権型のワークフローです。 これは、中央にある一つのハブ (リポジトリ) がコードを受け入れ、他のメンバー全員がそこに作業内容を同期させるという流れです。 多数の開発者がハブにつながるノードとなり、作業を一か所に集約します。

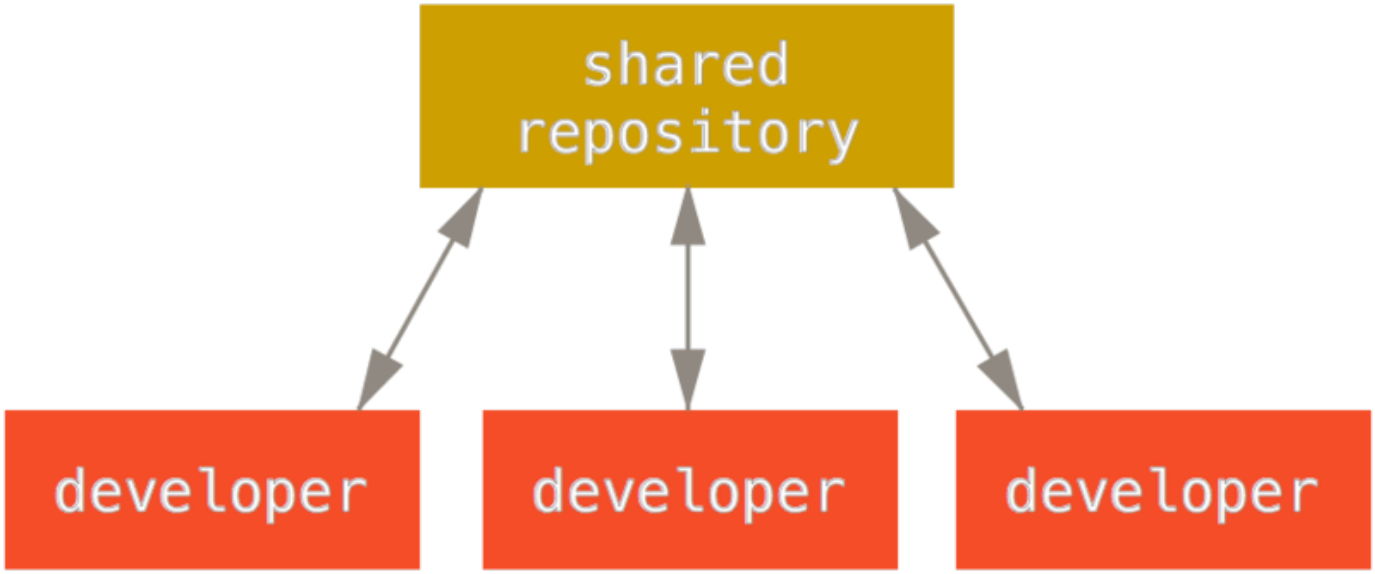



*Figure 53. 中央集権型のワークフロー*

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

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

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

## 統合マネージャー型のワークフロー

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

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

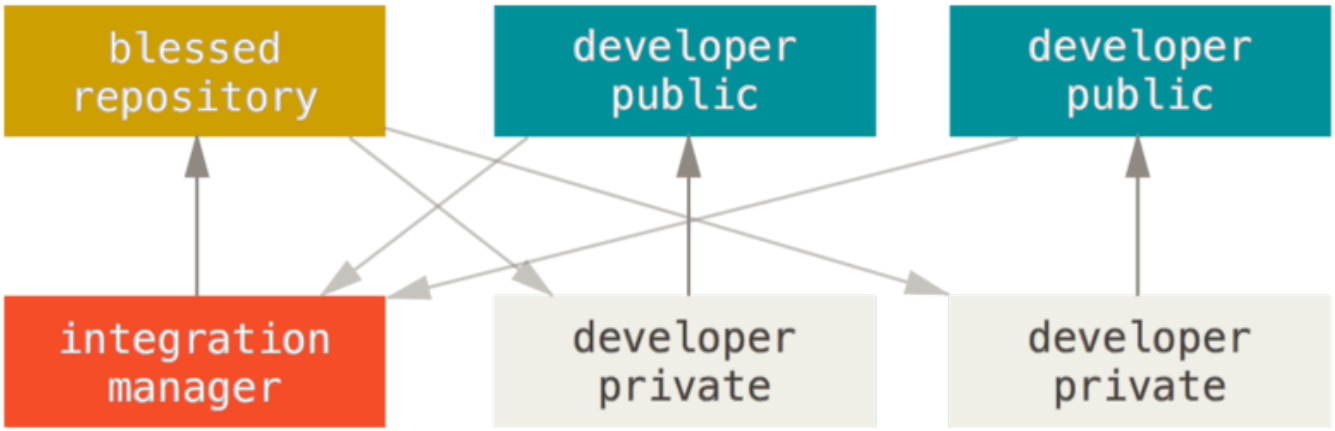



*Figure 54. 統合マネージャー型のワークフロー*

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

## 独裁者と副官型のワークフロー

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

1. 一般の開発者はトピックブランチ上で作業を進め、`master` の先頭にリベースする。独裁者の `master` ブランチがマスターとなる
2. 副官が各開発者のトピックブランチを自分の `master` ブランチにマージする
3. 独裁者が各副官の `master` ブランチを自分の `master` ブランチにマージする

4. 独裁者が自分の `master` をリポジトリにプッシュし、他のメンバーがリベースできるようにする

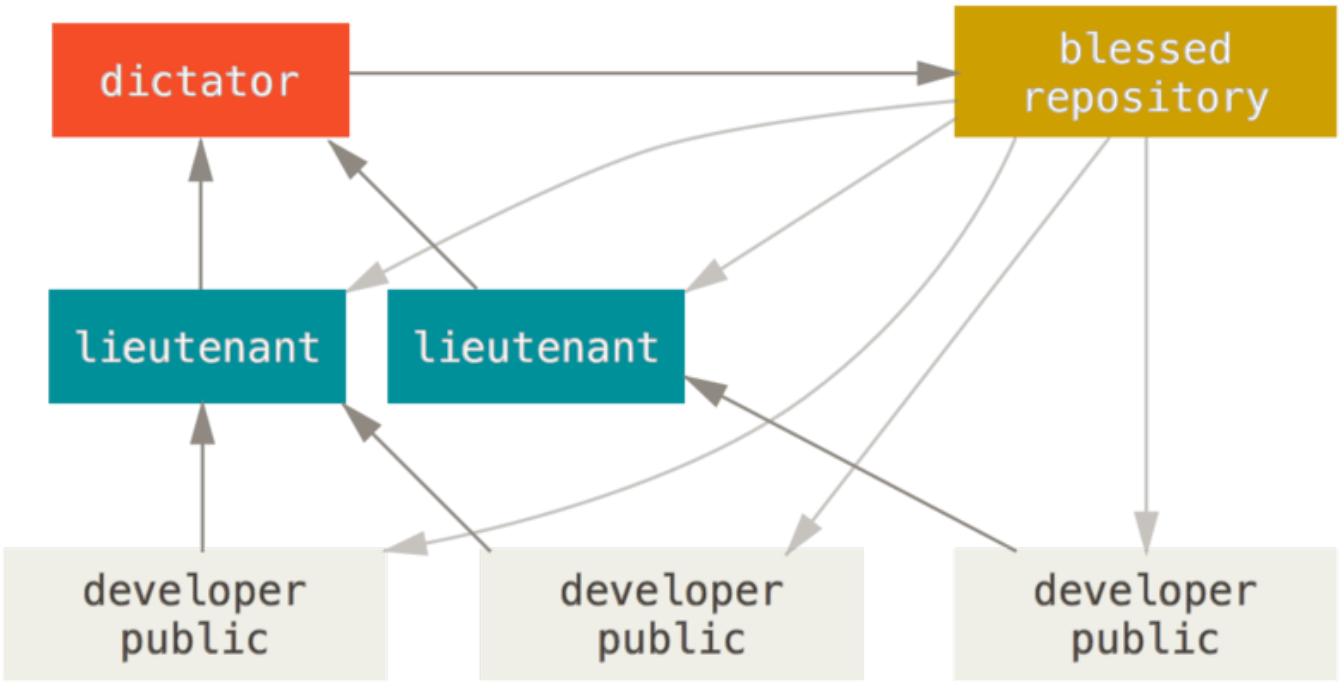



*Figure 55. 慈悲深い独裁者型のワークフロー*

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

### ワークフローのまとめ

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

## プロジェクトへの貢献

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

最初の要素はアクティブな貢献者の数です。そのプロジェクトに対してアクティブにコードを提供している開発者はどれくらいいるのか、そして彼らはどれくらいの頻度で提供しているのか。 よくあるのは、数名の開発者が一日数回のコミットを行うというものです。休眠状態のプロジェクトなら、もう少し頻度が低くなるでしょう。 企業やプロジェクトの規模が大きくなると、開発者の数が数千人になることもあります。数百から下手したら千を超えるようなコミットが毎日やってきます。 開発者の数が増えれば増えるほど、あなたのコードをきちんと適用したり他のコードをマージしたりするのが難しくなります。 あなたが手元で作業をし

ている間に他の変更が入って、手元で変更した内容が無意味になってしまったりあるいは他の変更を壊してしまう羽目になったり。そのせいで、手元の変更を適用してもらうための待ち時間が発生したり。 手元のコードを常に最新の状態にし、正しいコミットを作るにはどうしたらいいのでしょうか。

次に考えるのは、プロジェクトが採用しているワークフローです。 中央管理型で、すべての開発者がコードに対して同等の書き込みアクセス権を持っている状態? 特定のメンテナーや統合マネージャーがすべてのパッチをチェックしている? パッチを適用する前にピアレビューをしている? あなたはパッチをチェックしたりピアレビューに参加したりしている人? 副官型のワークフローを使っており、まず彼らにコードを渡さなければならない?

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

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

## コミットの指針

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

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

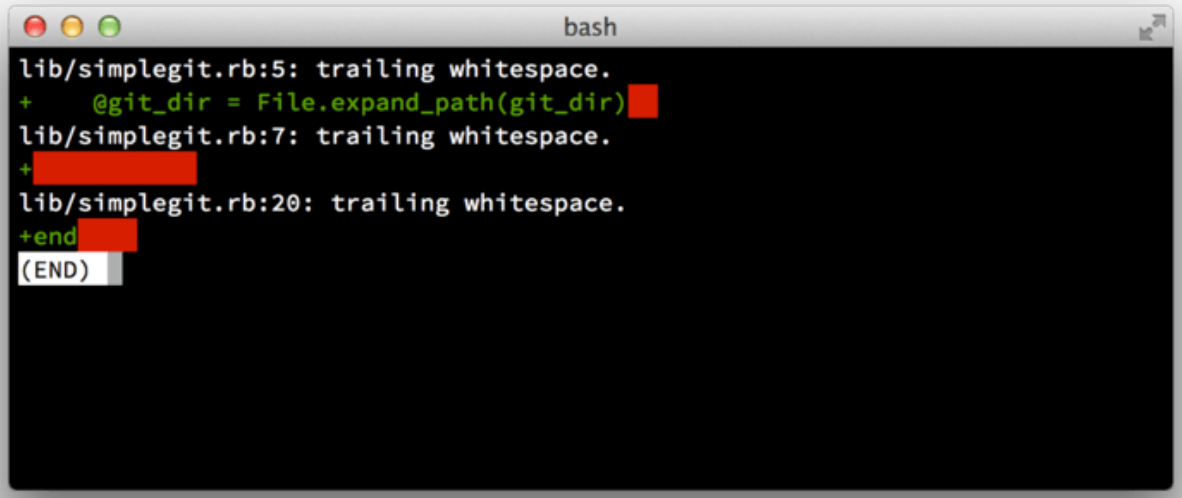



*Figure 56. `git diff --check` 実行結果*

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

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

最後に注意しておきたいのが、コミットメッセージです。 よりよいコミットメッセージを書く習慣を身に着けておくと、Git を使った共同作業をより簡単に行えるようになります。 一般的な規則として、メッセージの最初には変更の概要を一行 (50 文字以内) にまとめた説明をつけるようにします。その後に空行をひとつ置いてからより詳しい説明を続けます。 Git プロジェクトでは、その変更の動機やこれまでの実装との違いなどのできるだけ詳しい説明をつけることを推奨しています。参考にするとよいでしょう。 また、メッセージでは命令形、現在形を使うようにしています。 つまり “私は○○のテストを追加しました (I added tests for)” とか “○○のテストを追加します (Adding tests for,)” ではなく “○○のテストを追加 (Add tests for.)” 形式にするということです。 Tim Pope が書いたテンプレート (の日本語訳) を以下に示します。

```
短い（50 文字以下での）変更内容のまとめ

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

空行を置いて、さらに段落を続けることもできます。

- 箇条書きも可能

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

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

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

## 非公開な小規模のチーム

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

こういった環境では、今まで Subversion やその他の中央管理型システムを使っていたときとほぼ同じワークフローで作業を進めることができます。 オフラインでコミットできたりブランチやマージが楽だったりといった Git ならではの利点はいかせますが、作業の流れ自体は今までとほぼ同じです。最大の違いは、マージが (コミット時にサーバー側で行われるのではなく) クライアント側で行われるということです。 二人の開発者が共有リポジトリで開発を始めるときにどうなるかを見ていきましょう。 最初の開発者 John が、リポジトリをクローンして変更を加え、それをローカルでコミットします (これ以降のメッセージでは、プロトコル関連のメッセージを `...` で省略しています)。

```
# John のマシン
$ git clone john@githost:simplegit.git
Initialized empty Git repository in /home/john/simplegit/.git/
...
$ cd simplegit/
$ vim lib/simplegit.rb
$ git commit -am 'removed invalid default value'
[master 738ee87] removed invalid default value
 1 files changed, 1 insertions(+), 1 deletions(-)
```

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

```
# Jessica のマシン
$ git clone jessica@githost:simplegit.git
Initialized empty Git repository in /home/jessica/simplegit/.git/
...
$ cd simplegit/
$ vim TODO
$ git commit -am 'add reset task'
[master fbff5bc] add reset task
 1 files changed, 1 insertions(+), 0 deletions(-)
```

Jessica が作業内容をサーバーにプッシュします。

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

John も同様にプッシュしようとしました。

```
# John のマシン
$ git push origin master
To john@githost:simplegit.git
 ! [rejected]        master -> master (non-fast forward)
error: failed to push some refs to 'john@githost:simplegit.git'
```

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

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

この時点で、John のローカルリポジトリはこのようになっています。

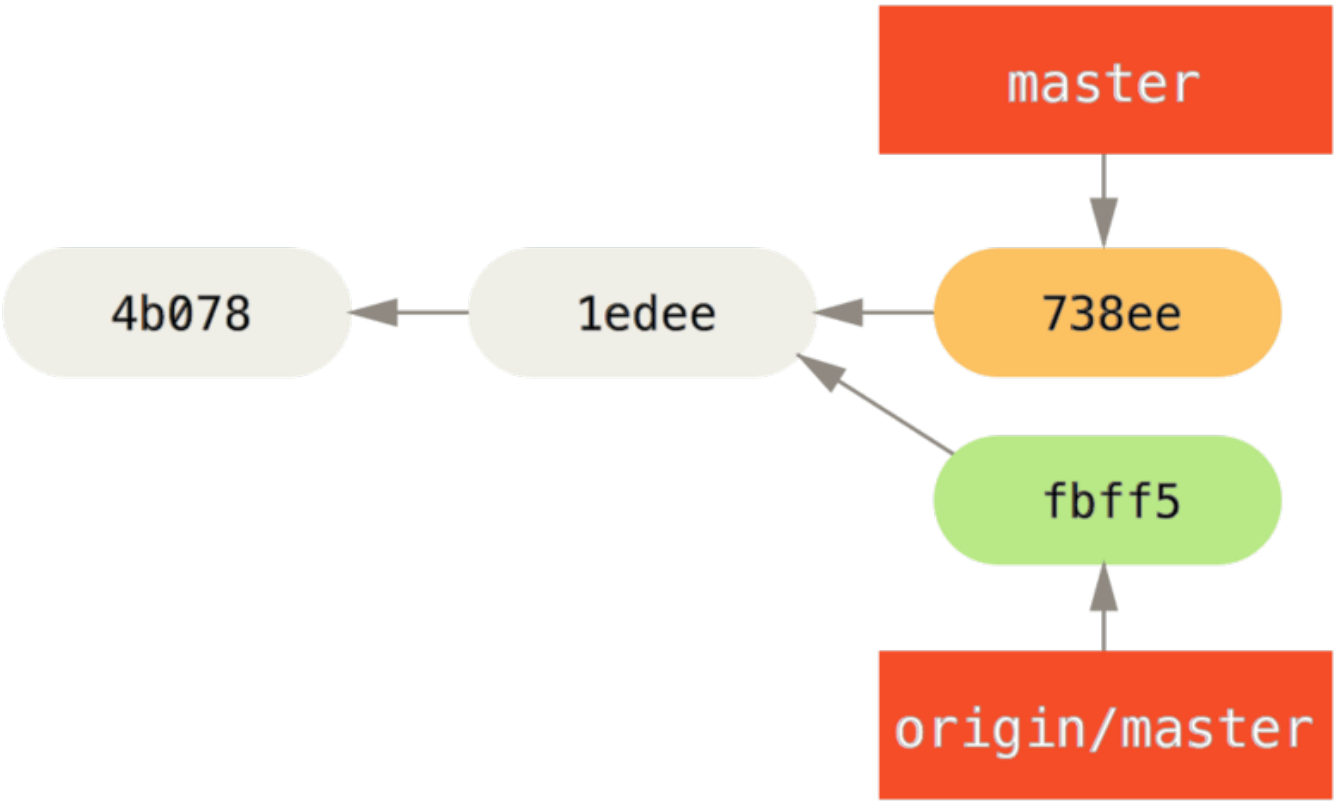



*Figure 57. John の分岐した歴史*

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

```
$ git merge origin/master
Merge made by recursive.
 TODO |    1 +
 1 files changed, 1 insertions(+), 0 deletions(-)
```

マージがうまくいきました。John のコミット履歴は次のようになります。

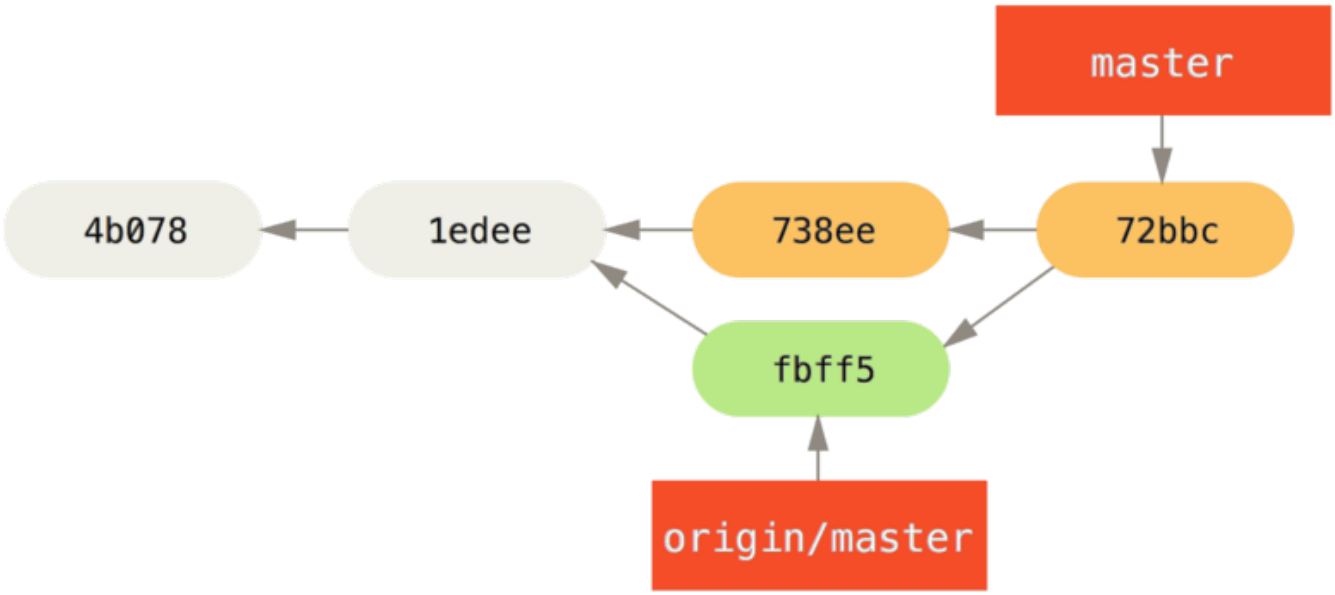



*Figure 58. `origin/master` をマージしたあとの John のリポジトリ*

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

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

最終的に、John のコミット履歴は以下のようになりました。

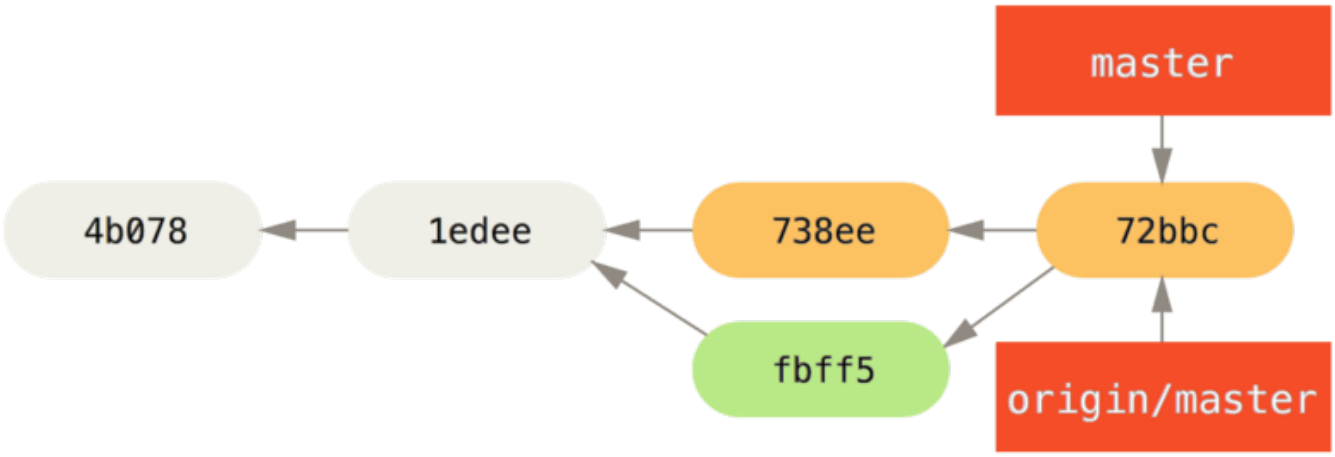



*Figure 59. origin サーバーにプッシュした後の John の履歴*

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

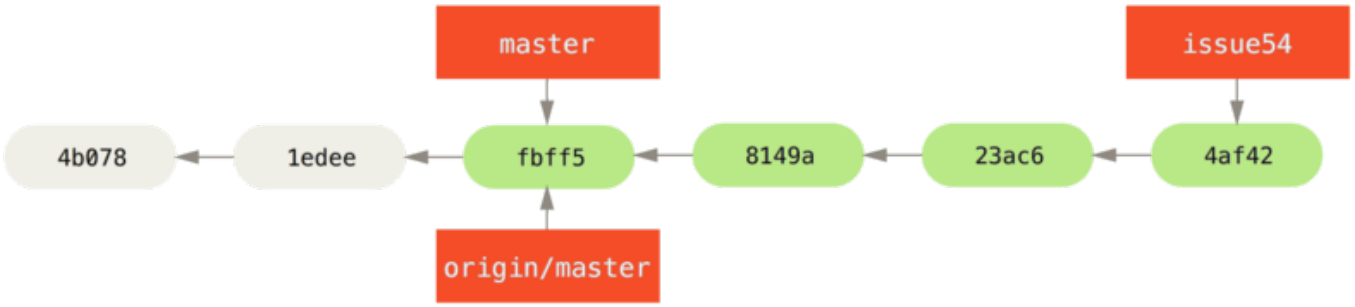



*Figure 60. Jessica のコミット履歴*

Jessica は John の作業を取り込もうとしました。

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

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

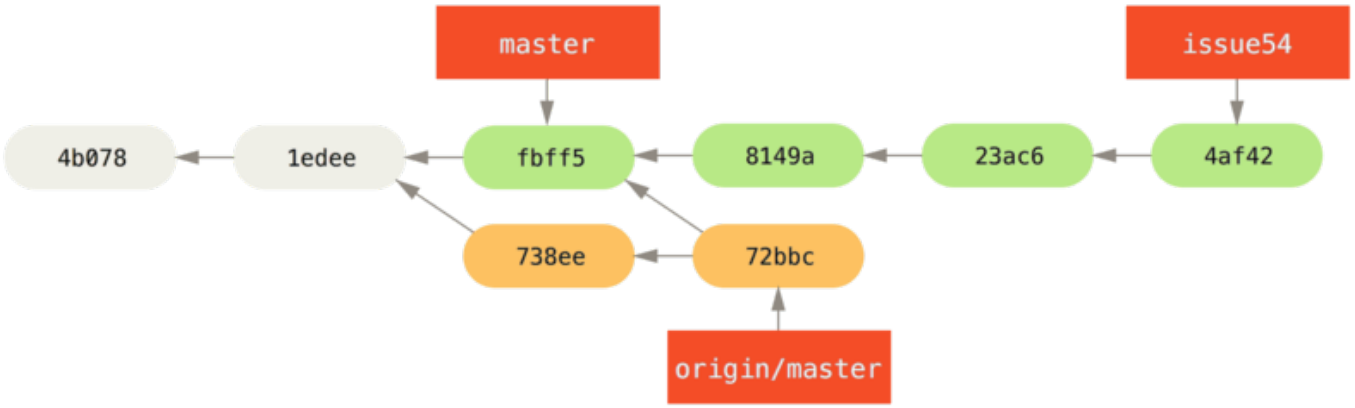



*Figure 61. John の変更を取り込んだ後の Jessica の履歴*

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

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

   removed invalid default value
```

`issue54..origin/master` はログのフィルター記法です。このように書くと、後者のブランチ（この例では `origin/master`）には含まれるが前者のブランチ（この例では `issue54`）には含まれないコミットのロ

グだけを表示します。この記法の詳細は コミットの範囲指定 で説明します。

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

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

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

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

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

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

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

こちらもうまく完了しました。Jessica の履歴はこのようになります。

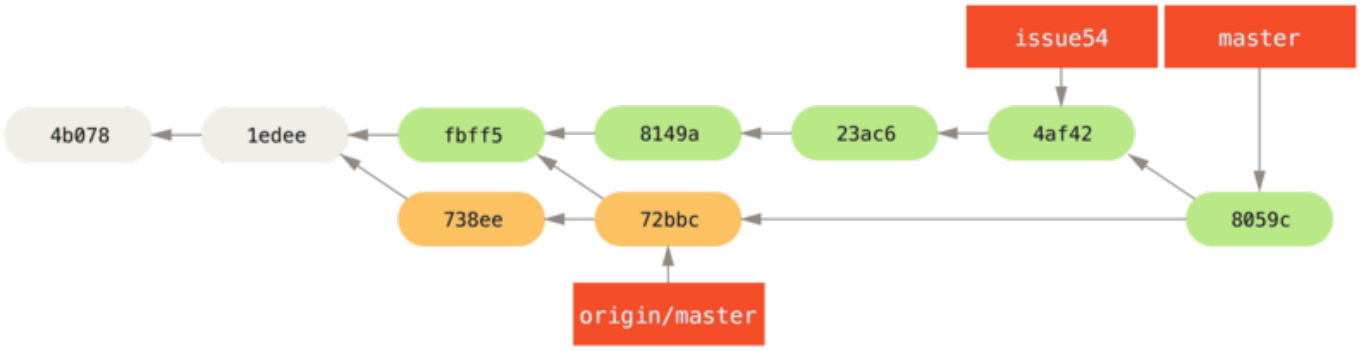



*Figure 62. John の変更をマージした後の Jessica の履歴*

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

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

各開発者が何度かコミットし、お互いの作業のマージも無事できました。

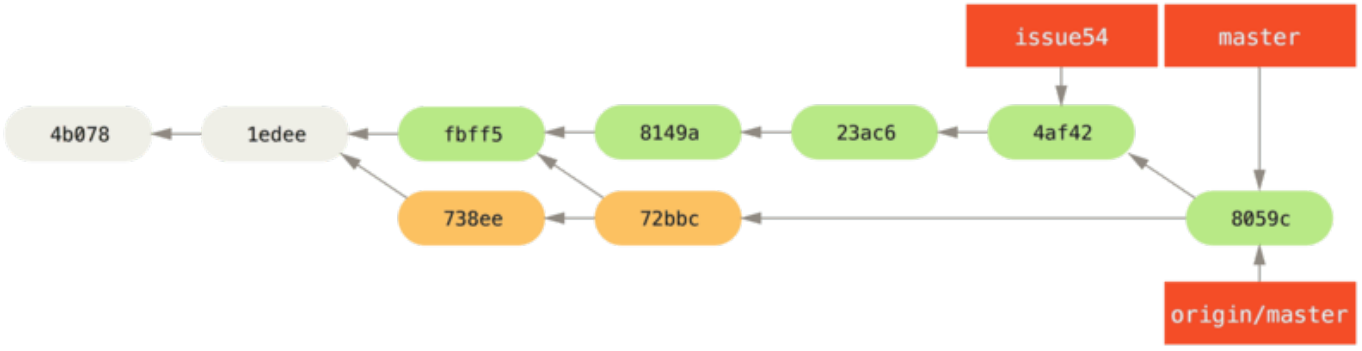



*Figure 63. すべての変更をサーバーに書き戻した後の Jessica の履歴*

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

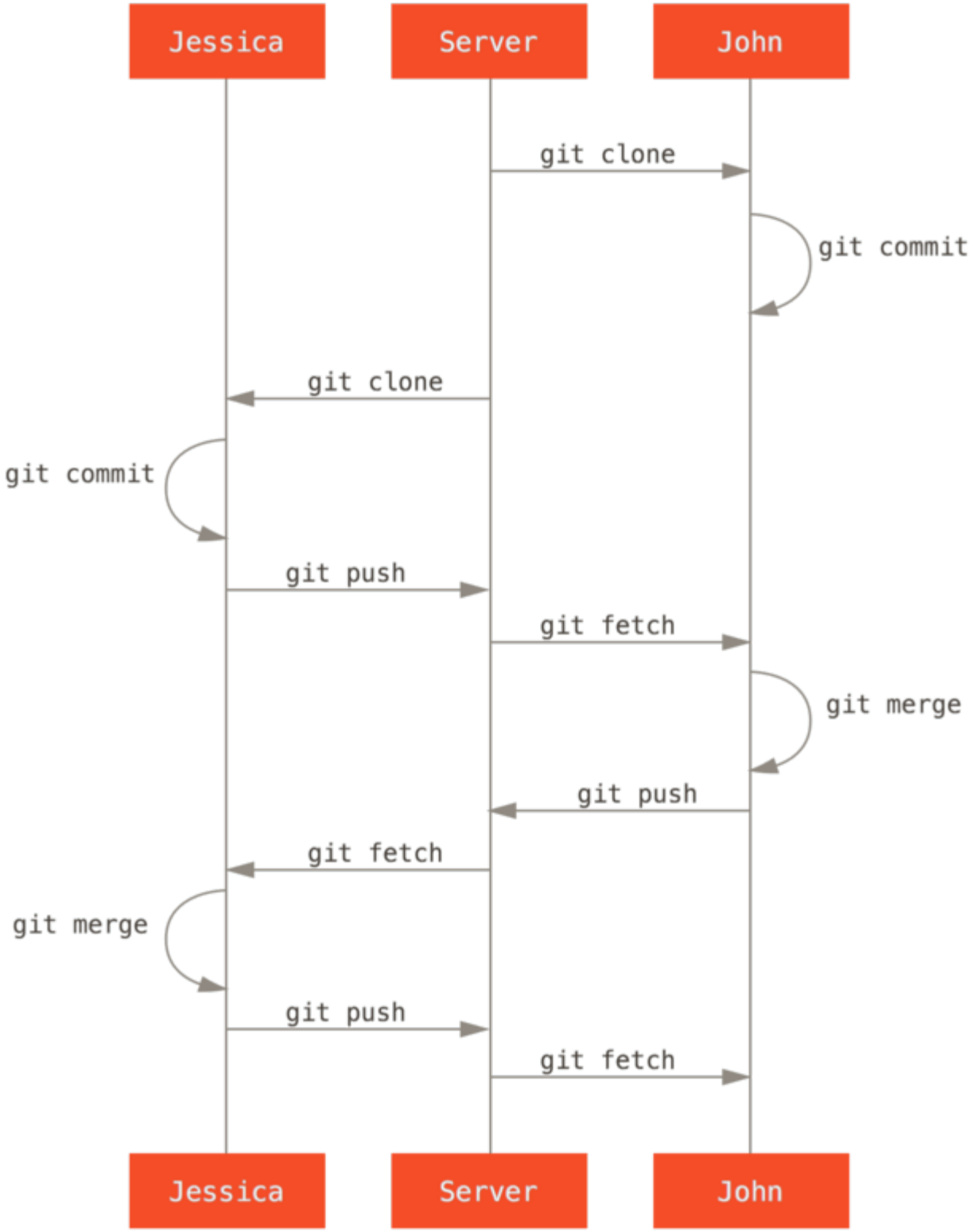



*Figure 64. 複数開発者での Git を使ったシンプルな開発作業のイベントシーケンス*

### 非公開で管理されているチーム

次に扱うシナリオは、大規模な非公開のグループに貢献するものです。 機能単位の小規模なグループで共同

作業した結果を別のグループと統合するような環境での作業の進め方を学びましょう。

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

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

```
# Jessica のマシン
$ git checkout -b featureA
Switched to a new branch 'featureA'
$ vim lib/simplegit.rb
$ git commit -am 'add limit to log function'
[featureA 3300904] add limit to log function
 1 files changed, 1 insertions(+), 1 deletions(-)
```

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

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

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

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

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

```
$ vim lib/simplegit.rb
$ git commit -am 'made the ls-tree function recursive'
[featureB e5b0fdc] made the ls-tree function recursive
 1 files changed, 1 insertions(+), 1 deletions(-)
$ vim lib/simplegit.rb
$ git commit -am 'add ls-files'
[featureB 8512791] add ls-files
 1 files changed, 5 insertions(+), 0 deletions(-)
```

Jessica のリポジトリはこのようになっています。

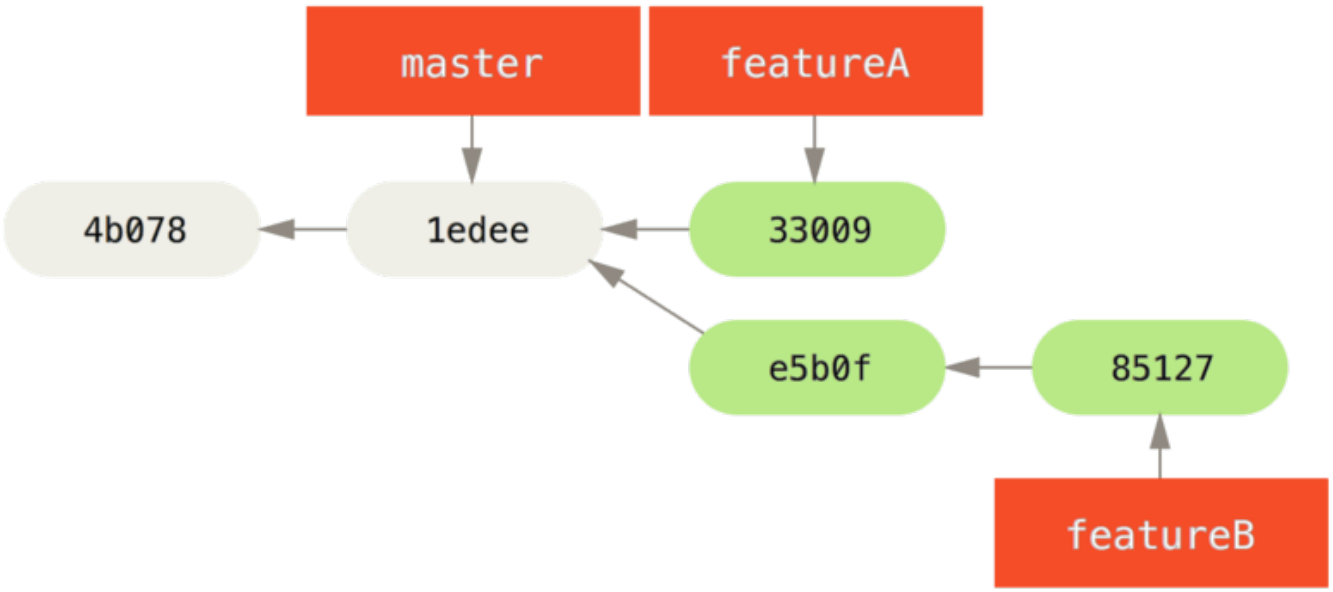



*Figure 65. Jessica のコミット履歴*

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

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

次に、`git merge` でこの内容を自分の作業にマージします。

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

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

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

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

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

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

そして、`git log` で何が変わったのかを確認します。

```
$ git log featureA..origin/featureA
commit aad881d154acdaeb2b6b18ea0e827ed8a6d671e6
Author: John Smith <jsmith@example.com>
Date:   Fri May 29 19:57:33 2009 -0700

    changed log output to 30 from 25
```

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

```
$ git checkout featureA
Switched to branch 'featureA'
$ git merge origin/featureA
Updating 3300904..aad881d
Fast forward
 lib/simplegit.rb |   10 +++++++++-
1 files changed, 9 insertions(+), 1 deletions(-)
```

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

```
$ git commit -am 'small tweak'
[featureA 774b3ed] small tweak
 1 files changed, 1 insertions(+), 1 deletions(-)
$ git push
...
To jessica@githost:simplegit.git
   3300904..774b3ed  featureA -> featureA
```

Jessica のコミット履歴は、この時点で以下のようになります。

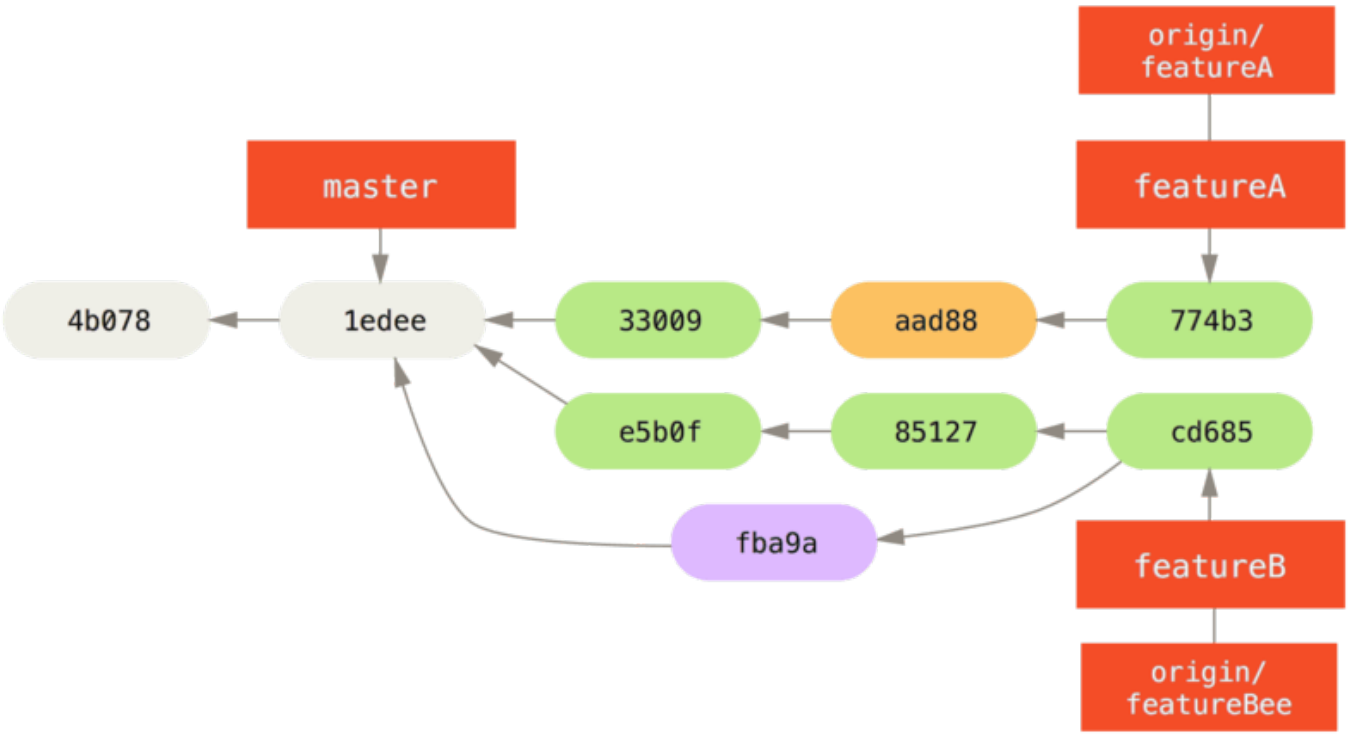



*Figure 66. Jessica がブランチにコミットした後のコミット履歴*

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

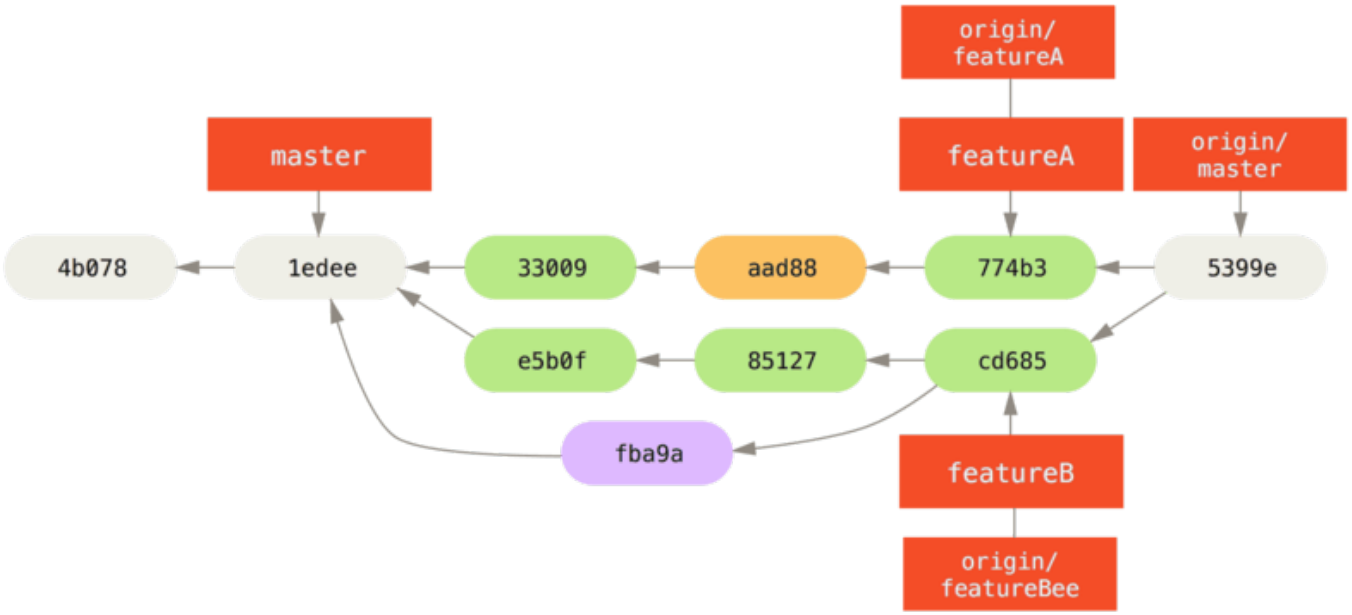



*Figure 67. Jessica が両方のトピックブランチをマージしたあとのコミット履歴*

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

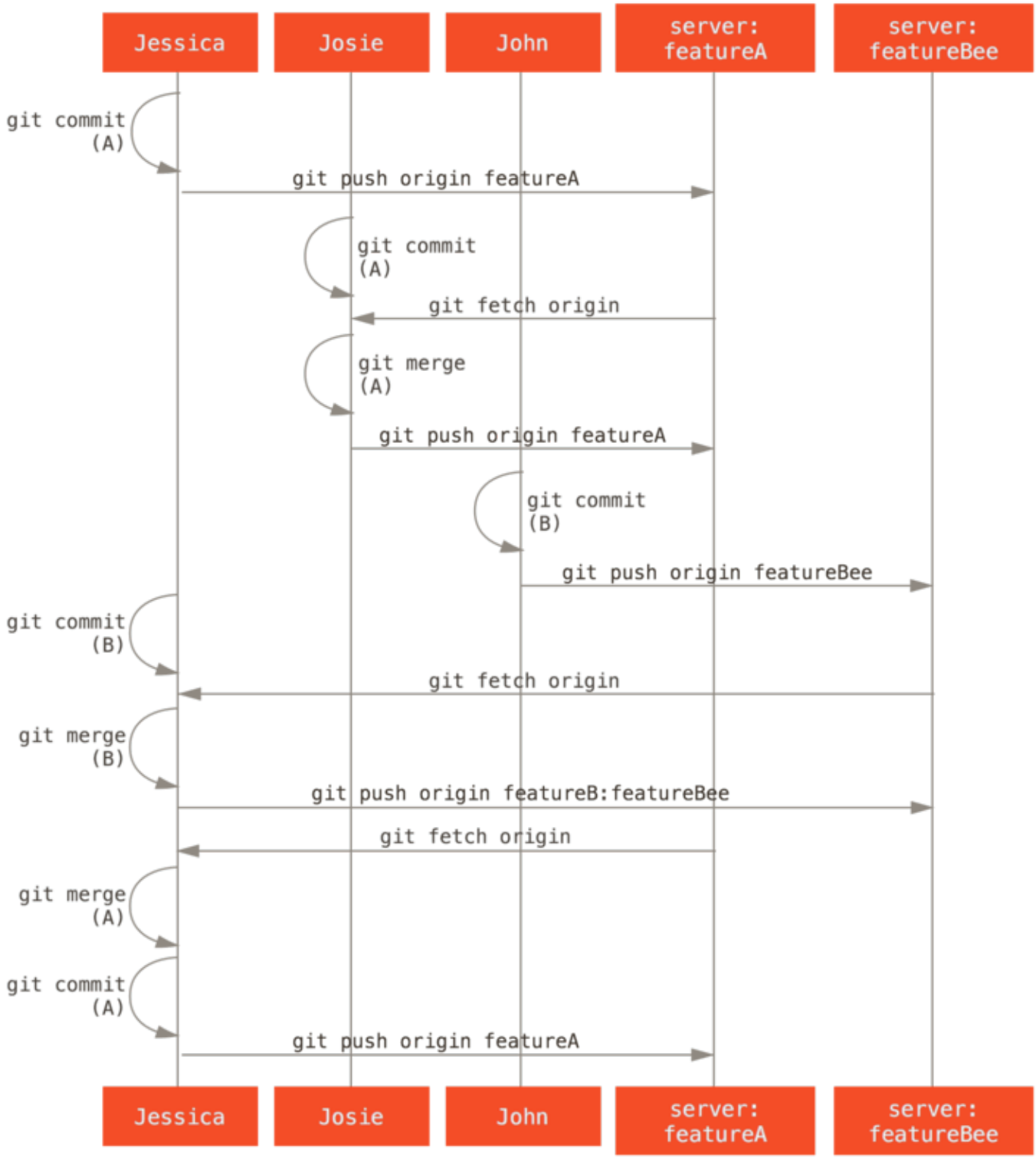



*Figure 68. 管理されたチームでのワークフローの基本的な流れ*

## フォークされた公開プロジェクト

公開プロジェクトに貢献するとなると、また少し話が変わってきます。 そのプロジェクトのブランチを直接更新できる権限はないでしょうから、何か別の方法でメンテナに接触する必要があります。 まずは、フォークをサポートしている Git ホスティングサービスでフォークを使って貢献する方法を説明します。 多くの Git

ホスティングサービス（GitHub、 BitBucket、 Google Code、 repo.or.cz など）
がフォークをサポートしており、メンテナの多くはこの方式での協力を期待しています。 そしてこの次のセクションでは、メールでパッチを送る形式での貢献について説明します。

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

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

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

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

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

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

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

自分用のフォークに作業内容をプッシュし終えたら、それをメンテナに伝えましょう。 これは、よく「プルリクエスト」と呼ばれるもので、ウェブサイトから実行する (GutHub には Pull request を行う独自の仕組みがあります。詳しくは GitHub で説明します) こともできれば、 `git request-pull` コマンドの出力をプロジェクトのメンテナにメールで送ることもできます。

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

```
$ git request-pull origin/master myfork
The following changes since commit
1edee6b1d61823a2de3b09c160d7080b8d1b3a40:
  John Smith (1):
        added a new function

are available in the git repository at:

  git://githost/simplegit.git featureA

Jessica Smith (2):
      add limit to log function
      change log output to 30 from 25

 lib/simplegit.rb |   10 +++++++++-
 1 files changed, 9 insertions(+), 1 deletions(-)
```

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

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

```
$ git checkout -b featureB origin/master
#  (作業)
$ git commit
$ git push myfork featureB
#  (メンテナにメールを送る)
$ git fetch origin
```

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

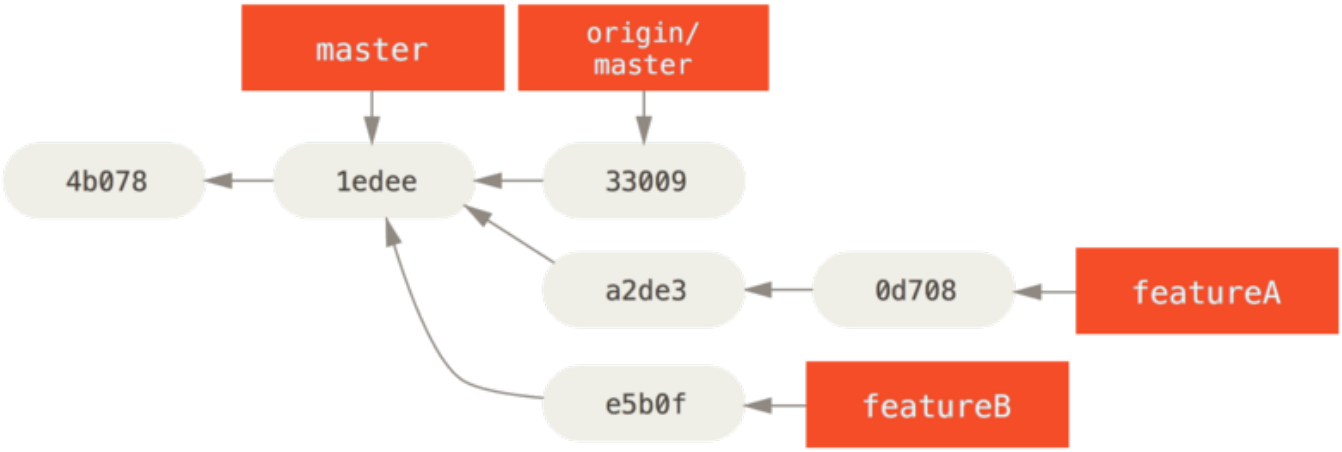



*Figure 69. featureB に関する作業のコミット履歴*

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

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

これで、あなたの歴史は featureA の作業を終えた後のコミット履歴 のように書き換えられました。

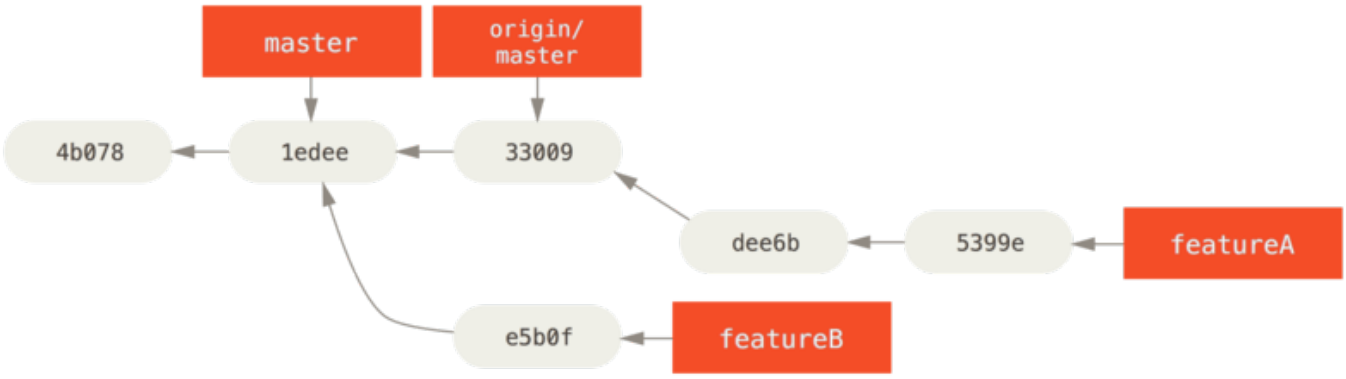



*Figure 70. featureA の作業を終えた後のコミット履歴*

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

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

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

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

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

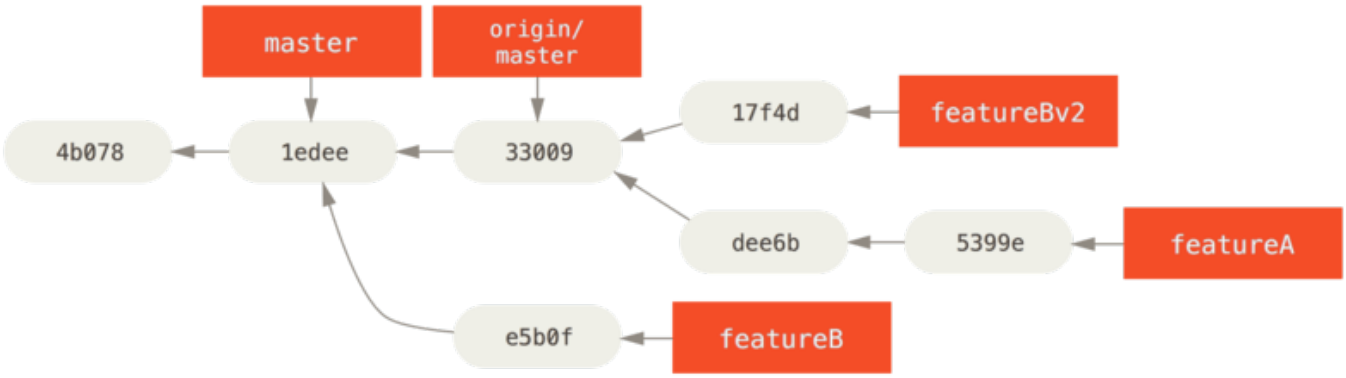



*Figure 71. `featureBv2` の作業を終えた後のコミット履歴*

## メールを使った公開プロジェクトへの貢献

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

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

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

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

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

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

```
$ cat 0001-add-limit-to-log-function.patch
From 330090432754092d704da8e76ca5c05c198e71a8 Mon Sep 17 00:00:00 2001
From: Jessica Smith <jessica@example.com>
Date: Sun, 6 Apr 2008 10:17:23 -0700
Subject: [PATCH 1/2] add limit to log function

Limit log functionality to the first 20

---
 lib/simplegit.rb |    2 +-
 1 files changed, 1 insertions(+), 1 deletions(-)

diff --git a/lib/simplegit.rb b/lib/simplegit.rb
index 76f47bc..f9815f1 100644
--- a/lib/simplegit.rb
+++ b/lib/simplegit.rb
@@ -14,7 +14,7 @@ class SimpleGit
   end

   def log(treeish = 'master')
-    command("git log #{treeish}")
+    command("git log -n 20 #{treeish}")
   end

   def ls_tree(treeish = 'master')
--
2.1.0
```

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

これをメーリングリストに投稿するには、メールソフトにファイルの内容を貼り付けるか、あるいはコマンドラインのプログラムを使います。 ファイルの内容をコピーして貼り付けると「かしこい」メールソフトが勝手に改行の位置を変えてしまうなどの問題が起こりがちです。 ありがたいことに Git には、きちんとしたフォーマットのパッチを IMAP で送ることを支援するツールが用意されています。これを使うと便利です。 ここでは、パッチを Gmail で送る方法を説明しましょう。というのも、一番よく知っているメールソフトが Gmail だからです。さまざまなメールソフトでの詳細なメール送信方法が、Git ソースコードにある `Documentation/SubmittingPatches` の最後に載っています。

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

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

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

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

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

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

```
[sendemail]
  smtpencryption = tls
  smtpserver = smtp.gmail.com
  smtpuser = user@gmail.com
  smtpserverport = 587
```

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

```
$ git send-email *.patch
0001-added-limit-to-log-function.patch
0002-changed-log-output-to-30-from-25.patch
Who should the emails appear to be from? [Jessica Smith
<jessica@example.com>]
Emails will be sent from: Jessica Smith <jessica@example.com>
Who should the emails be sent to? jessica@example.com
Message-ID to be used as In-Reply-To for the first email? y
```

Git はその後、各パッチについてこのようなログ情報をはき出すはずです。

```
(mbox) Adding cc: Jessica Smith <jessica@example.com> from
  \line 'From: Jessica Smith <jessica@example.com>'
OK. Log says:
Sendmail: /usr/sbin/sendmail -i jessica@example.com
From: Jessica Smith <jessica@example.com>
To: jessica@example.com
Subject: [PATCH 1/2] added limit to log function
Date: Sat, 30 May 2009 13:29:15 -0700
Message-Id: <1243715356-61726-1-git-send-email-jessica@example.com>
X-Mailer: git-send-email 1.6.2.rc1.20.g8c5b.dirty
In-Reply-To: <y>
References: <y>

Result: OK
```

### まとめ

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

## プロジェクトの運営

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

### トピックブランチでの作業

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

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

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

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

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

## メールで受け取ったパッチの適用

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

### apply によるパッチの適用

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

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

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

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

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

何も出力されなければ、そのパッチはうまく適用できるということです。 このコマンドは、チェックに失敗

した場合にゼロ以外の値を返して終了します。スクリプト内でチェックしたい場合などにはこの返り値を使用します。

### am でのパッチの適用

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

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

```
From 330090432754092d704da8e76ca5c05c198e71a8 Mon Sep 17 00:00:00 2001
From: Jessica Smith <jessica@example.com>
Date: Sun, 6 Apr 2008 10:17:23 -0700
Subject: [PATCH 1/2] add limit to log function

Limit log functionality to the first 20
```

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

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

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

どんなパッチを適用したのかが表示され、コミットも自動的に作られます。作者の情報はメールの From ヘッダと Date ヘッダから取得し、コミットメッセージは Subject とメールの本文 (パッチより前の部分) から取得します。たとえば、先ほどごらんいただいた mbox の例にあるパッチを適用した場合は次のようなコミットとなります。

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

   add limit to log function

   Limit log functionality to the first 20
```

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

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

```
$ git am 0001-seeing-if-this-helps-the-gem.patch
Applying: seeing if this helps the gem
error: patch failed: ticgit.gemspec:1
error: ticgit.gemspec: patch does not apply
Patch failed at 0001.
When you have resolved this problem run "git am --resolved".
If you would prefer to skip this patch, instead run "git am --skip".
To restore the original branch and stop patching run "git am --abort".
```

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

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

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

```
$ git am -3 0001-seeing-if-this-helps-the-gem.patch
Applying: seeing if this helps the gem
error: patch failed: ticgit.gemspec:1
error: ticgit.gemspec: patch does not apply
Using index info to reconstruct a base tree...
Falling back to patching base and 3-way merge...
No changes -- Patch already applied.
```

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

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

```
$ git am -3 -i mbox
Commit Body is:
--------------------------
seeing if this helps the gem
--------------------------
Apply? [y]es/[n]o/[e]dit/[v]iew patch/[a]ccept all
```

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

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

## リモートブランチのチェックアウト

自前のリポジトリを持つ Git ユーザーが自分のリポジトリに変更をプッシュし、そのリポジトリの URL とリモートブランチ名だけをあなたにメールで連絡してきた場合のことを考えてみましょう。そのリポジトリをリモートとして登録し、それをローカルにマージすることになります。

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

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

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

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

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

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

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

## 何が変わるのかの把握

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

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

```
$ git log contrib --not master
commit 5b6235bd297351589efc4d73316f0a68d484f118
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri Oct 24 09:53:59 2008 -0700

    seeing if this helps the gem

commit 7482e0d16d04bea79d0dba8988cc78df655f16a0
Author: Scott Chacon <schacon@gmail.com>
Date:   Mon Oct 22 19:38:36 2008 -0700

    updated the gemspec to hopefully work better
```

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

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

```
$ git diff master
```

このコマンドで表示される diff は、誤解を招きかねないものです。 トピックブランチを切った時点からさらに `master` ブランチが先に進んでいたとすると、これは少し奇妙に見える結果を返します。 というのも、Git は現在のトピックブランチの最新のコミットのスナップショットと `master` ブランチの最新のコミットのスナップショットを直接比較するからです。 トピックブランチを切った後に `master` ブランチ上であるファイルに行を追加したとすると、スナップショットを比較した結果は「トピックブランチでその行を削除しようとしている」状態になります。

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

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

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

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

しかし、これでは不便です。そこで Git には、同じことをより手短にやるための手段としてトリプルドット構文が用意されています。 `diff` コマンドを実行するときにピリオドを三つ打った後に別のブランチを指定する

と、「現在いるブランチの最新のコミット」と「指定した二つのブランチの共通の先祖」とを比較するようになります。

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

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

## 提供された作業の取り込み

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

### マージのワークフロー

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

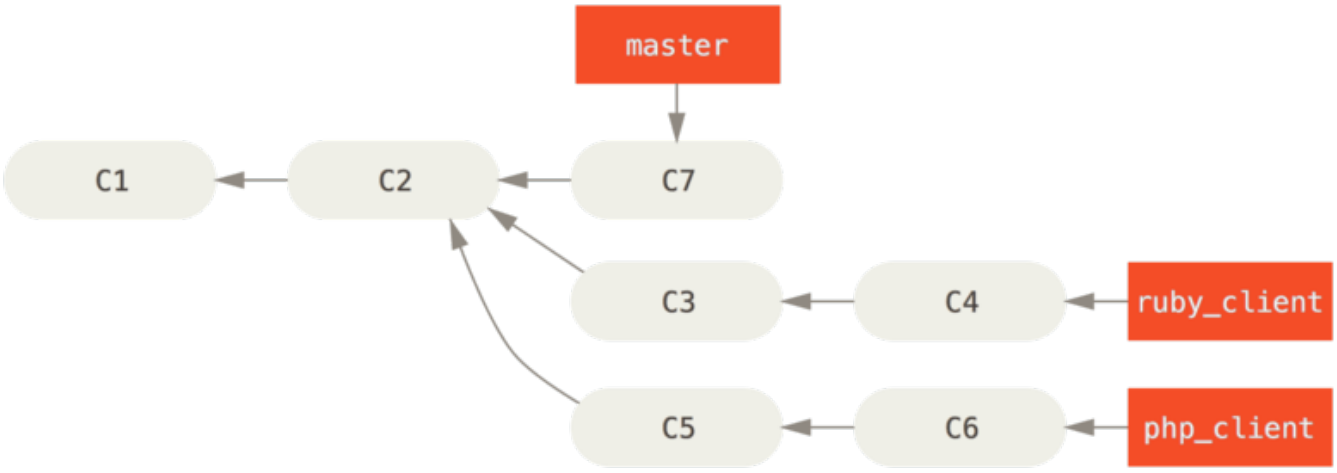



*Figure 72. いくつかのトピックブランチを含む履歴*

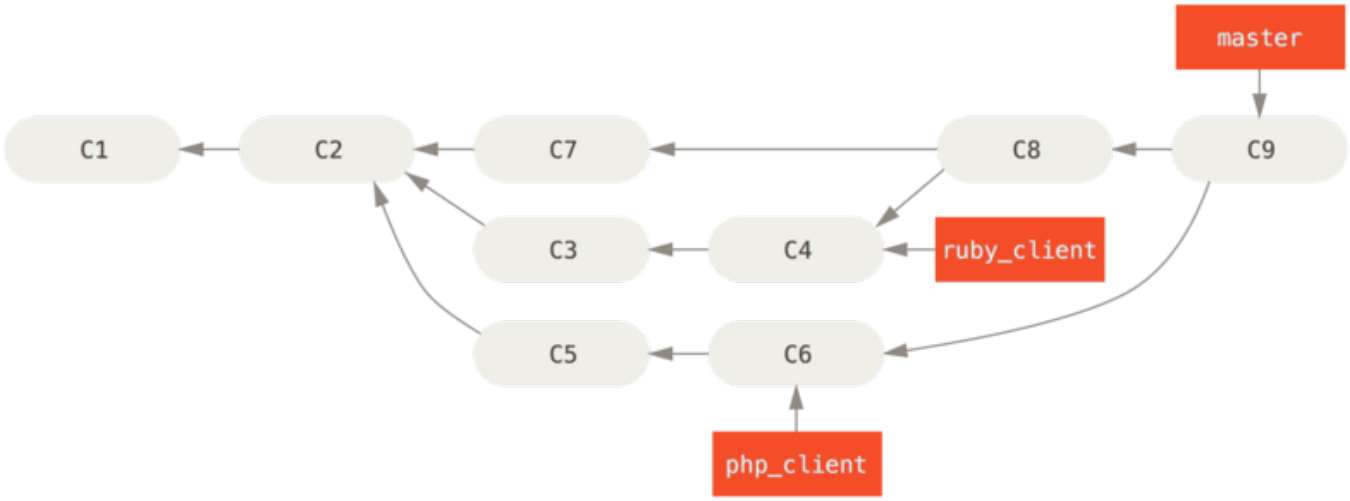



*Figure 73. トピックブランチをマージした後の状態*

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

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

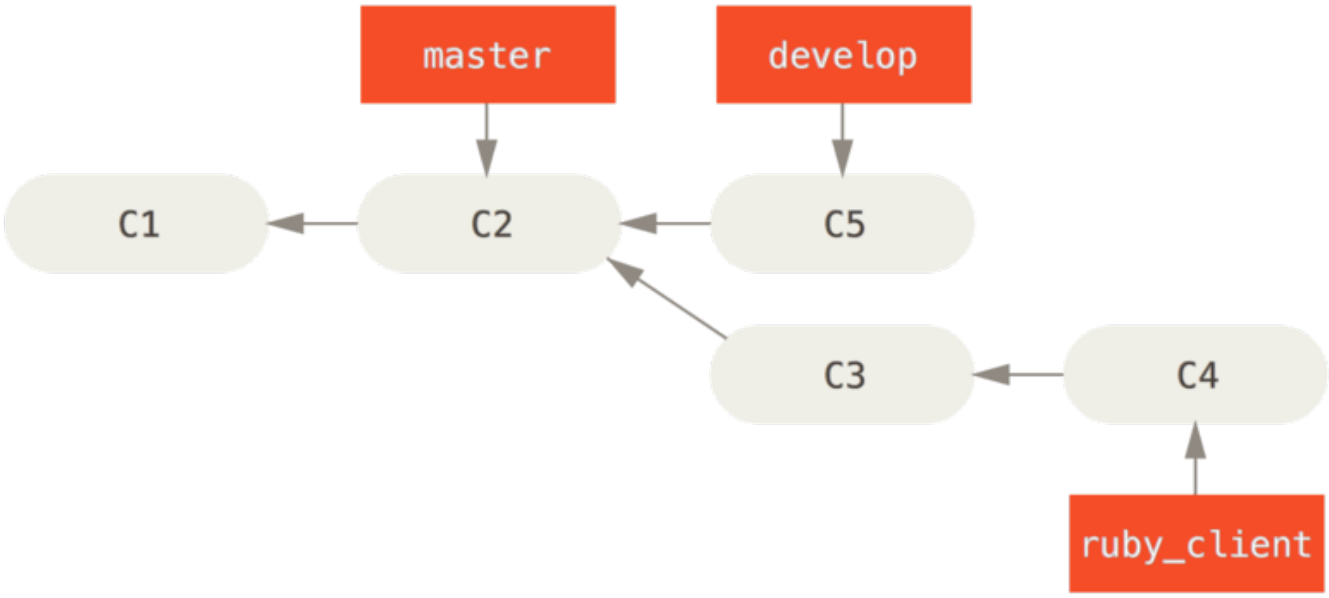



*Figure 74. トピックブランチのマージ前*

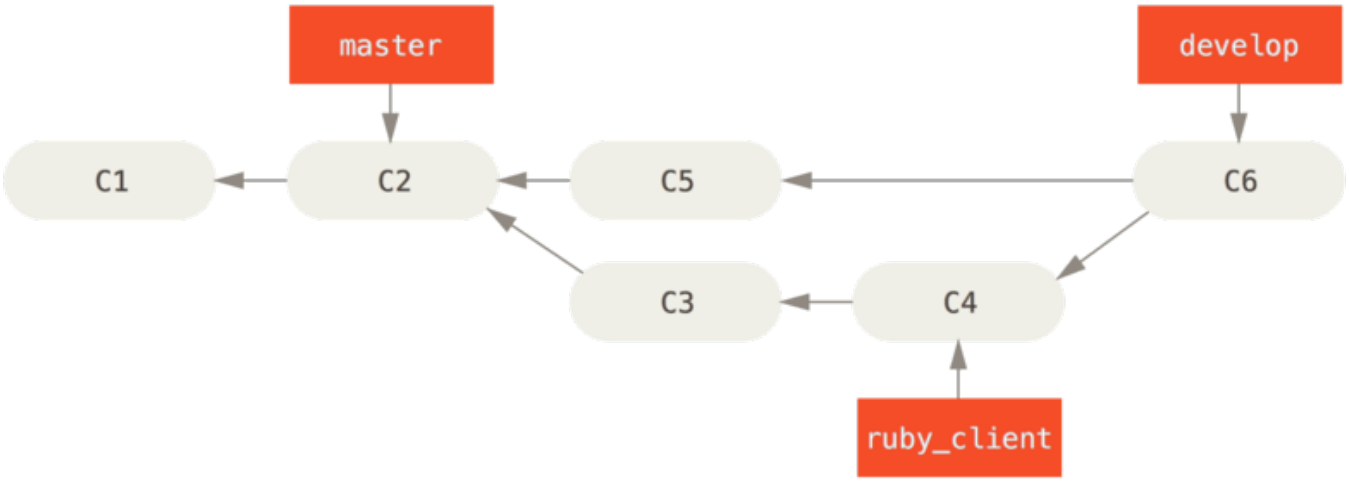



*Figure 75. トピックブランチのマージ後*

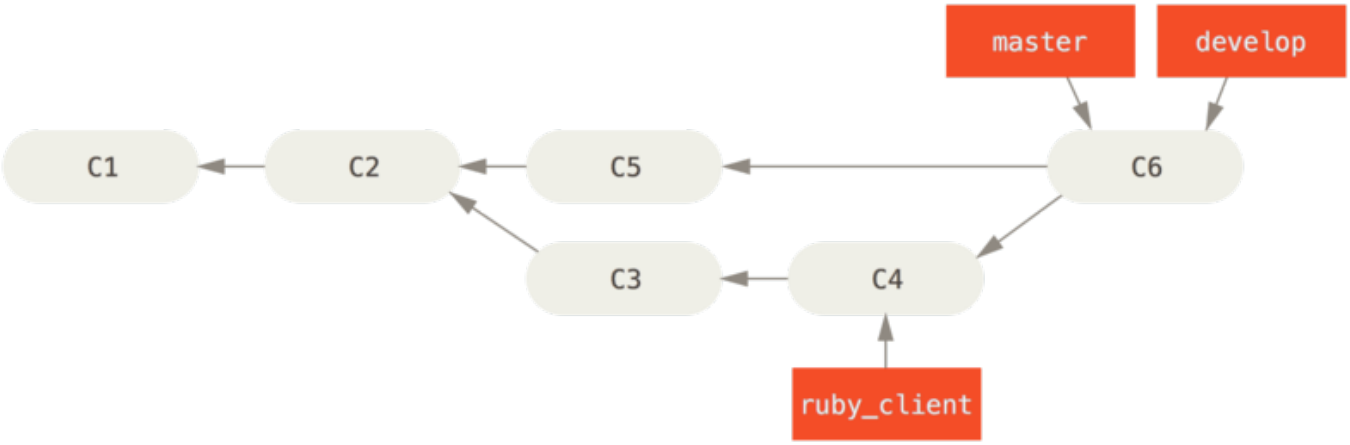



*Figure 76. プロジェクトのリリース後*

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

**大規模マージのワークフロー**

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

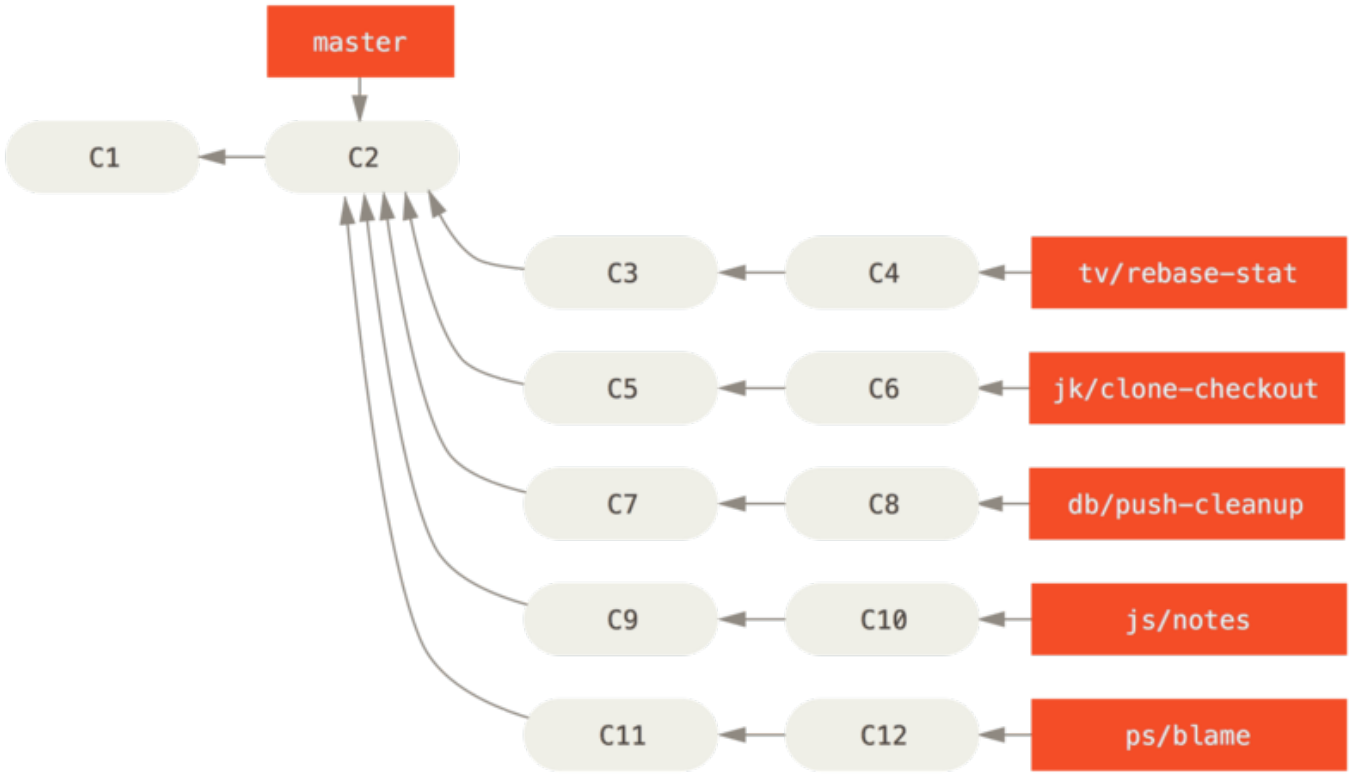



*Figure 77. 複数のトピックブランチの並行管理*

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

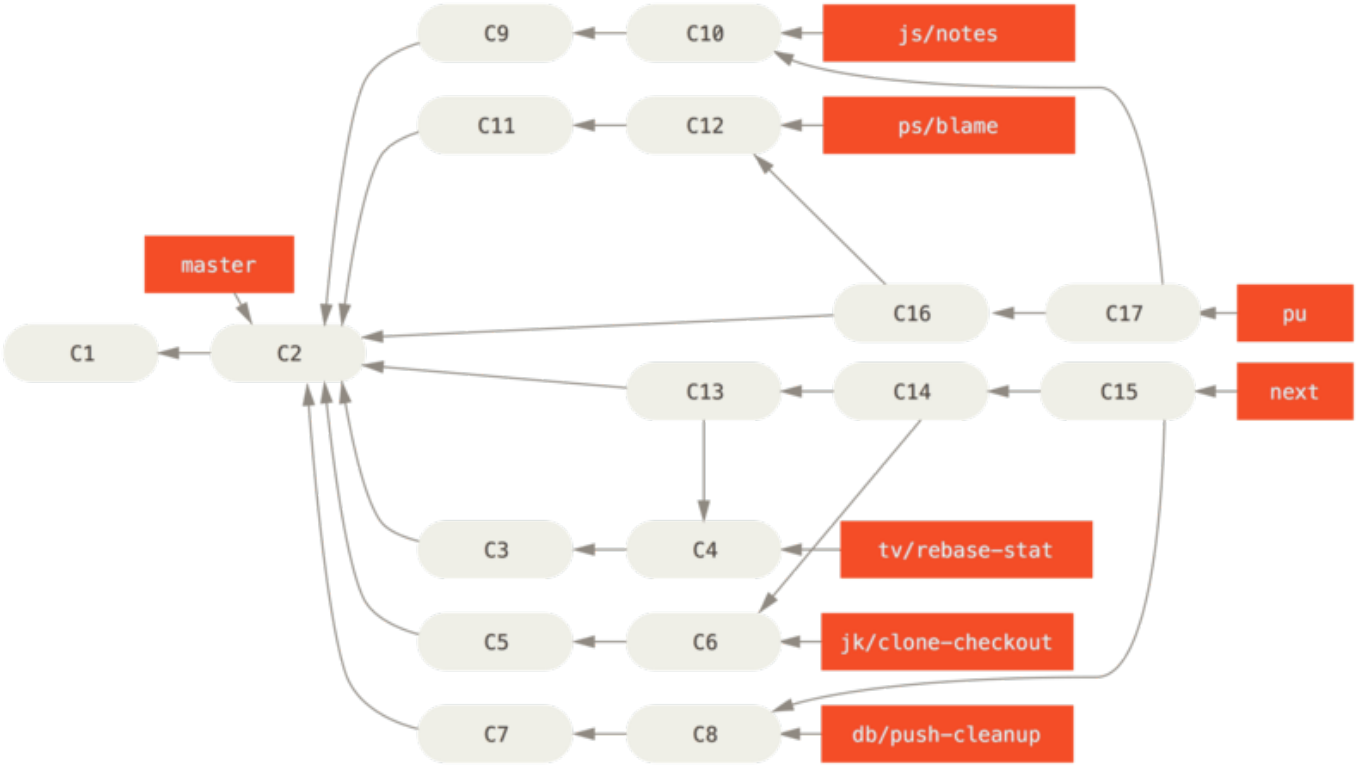



*Figure 78. 常時稼働する統合用ブランチへのトピックブランチのマージ*

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

**リベースとチェリーピックのワークフロー**

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

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

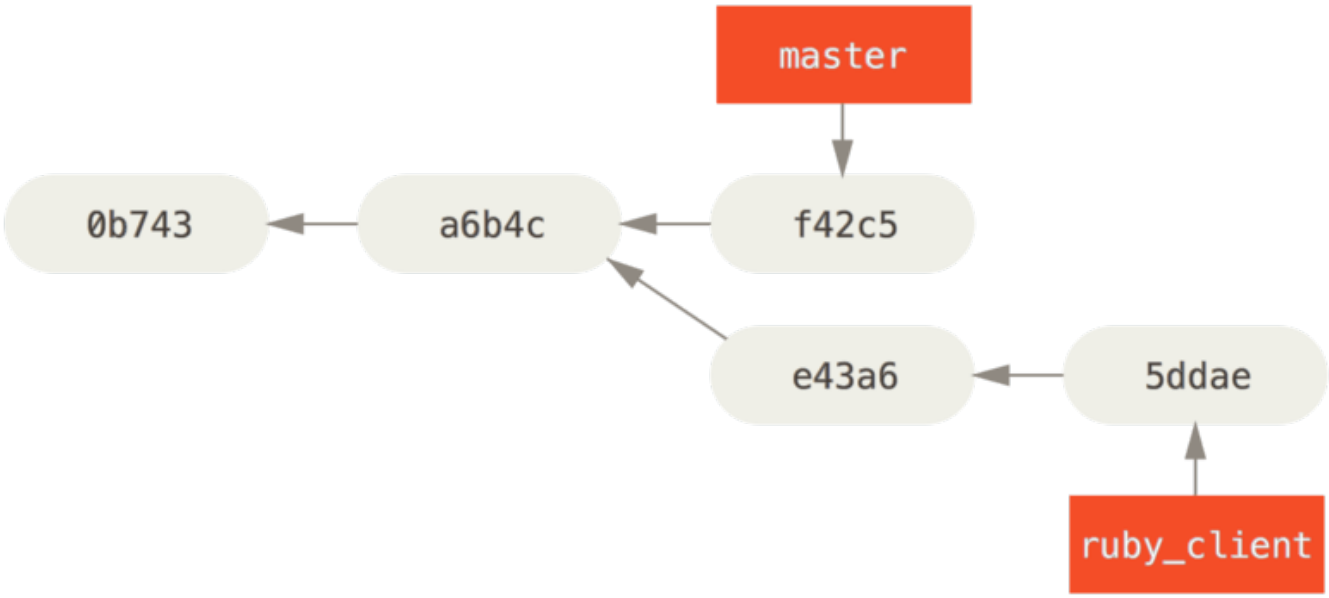



*Figure 79. チェリーピック前の歴史*

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

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

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

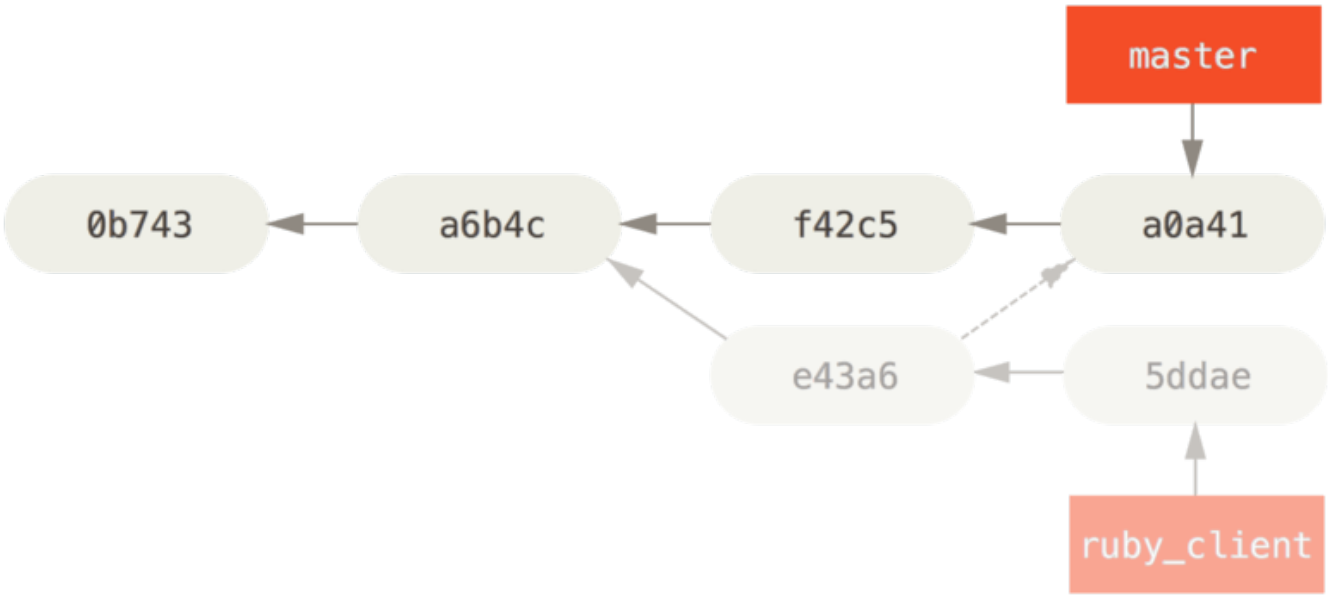



*Figure 80. トピックブランチのコミットをチェリーピックした後の歴史*

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

**Rerere**

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

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

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

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

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

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

### リリース用のタグ付け

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

```
$ git tag -s v1.5 -m 'my signed 1.5 tag'
You need a passphrase to unlock the secret key for
user: "Scott Chacon <schacon@gmail.com>"
1024-bit DSA key, ID F721C45A, created 2009-02-09
```

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

```
$ gpg --list-keys
/Users/schacon/.gnupg/pubring.gpg
---------------------------------
pub   1024D/F721C45A 2009-02-09 [expires: 2010-02-09]
uid                  Scott Chacon <schacon@gmail.com>
sub   2048g/45D02282 2009-02-09 [expires: 2010-02-09]
```

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

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

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

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

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

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

この鍵をインポートした人は、あなたが署名したすべてのタグを検証できるようになります。タグのメッセージに検証手順の説明を含めておけば、`git show <tag>` でエンドユーザー向けに詳しい検証手順を示すことができます。

## ビルド番号の生成

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

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

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

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

## リリースの準備

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

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

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

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

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

### 短いログ

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

```
$ git shortlog --no-merges master --not v1.0.1
Chris Wanstrath (8):
      Add support for annotated tags to Grit::Tag
      Add packed-refs annotated tag support.
      Add Grit::Commit#to_patch
      Update version and History.txt
      Remove stray `puts`
      Make ls_tree ignore nils

Tom Preston-Werner (4):
      fix dates in history
      dynamic version method
      Version bump to 1.0.2
      Regenerated gemspec for version 1.0.2
```

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

## まとめ

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

# GitHub

GitHub は世界最大の Git リポジトリホスティングサービスで、何百万もの開発者やプロジェクトが集う、共同作業の場になっています。 世の中の Git リポジトリの多くが GitHub に置かれており、多くのオープンソースプロジェクトが、 Git リポジトリ以外にも、課題追跡やコードレビューなどに GitHub を利用しています。 Git そのものとは直接関係ありませんが、Git を使っていれば、遅かれ早かれ GitHub を利用したくなる（あるいはそうせざるを得なくなる）でしょう。

本章では、GitHub を有効活用する方法を説明します。 アカウントの取得や管理、Git リポジトリの作成と利用、 プロジェクトに貢献したり自分のプロジェクトへの貢献を受け入れたりするときの一般的なワークフロー、 GitHubをプログラマティックに利用するためのインターフェイスなどのほかにも、 GitHubをうまく使っていくためのさまざまなヒントを紹介します。

GitHub に自分のプロジェクトを置いたり、GitHub にある他のプロジェクトで共同作業をしたりといったことに興味がないかたは、 本章を読み飛ばして Git のさまざまなツール に進んでもかまいません。

**WARNING**

*インターフェイスは変わるもの*

GitHub に限ったことではありませんが、本章のスクリーンショットで示している UI は、将来的に変わる可能性があります。 本章で紹介しようとしている考えかたはそれでも伝わるでしょうが、 最新版のスクリーンショットを見たい場合は、オンライン版を見たほうがいいでしょう。

## アカウントの準備と設定

まずやるべきことは、ユーザーアカウントの作成です。無料で作れます。 https://github.com を開いて、他の人が使っていないユーザー名を選び、メールアドレスとパスワードを入力したら、あとは “Sign up for GitHub” という大きな緑色のボタンを押すだけです。

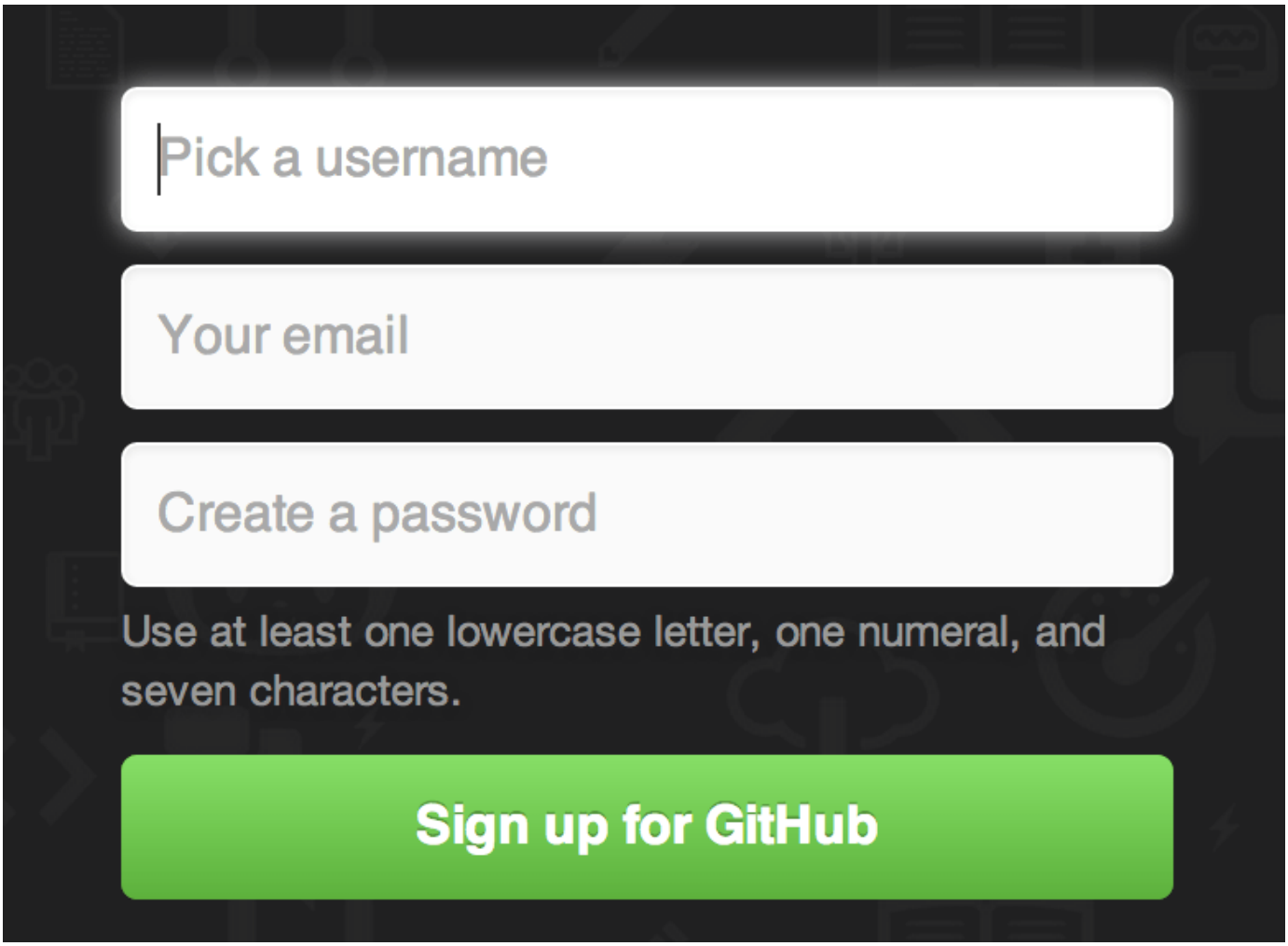



*Figure 81. GitHub のサインアップフォーム*

その次に出てくるのは、有償プランへのアップグレードについての説明です。 とりあえずは無視してもかまいません。 先ほど入力したメールアドレスが正しいことを確認するために、GitHubからのメールが届きます。 メールの指示に従ってください。後で見るとおり、これはとても重要です。

**NOTE** 無償版のアカウントで、GitHubのすべての機能が使えます。 ただし、すべてのプロジェクトを公開しなければいけない（誰でも読めるようにしなければいけない）という制約があります。 GitHubの有償プランは、プライベートなプロジェクトを所定の数だけ作れるようになっています。 しかし本書では、この機能については扱いません。

画面左上にある Octocat のロゴをクリックすると、ダッシュボードのページが開きます。 これで、GitHub を使う準備が整いました。

### SSH でのアクセス

この時点ですでに、`https://` プロトコルを使った Git リポジトリへの接続ができるようになっています。 接続する際には、先ほど指定したユーザー名とパスワードを利用します。 しかし、単に公開プロジェクトをクローンするだけなら、そもそもアカウントを取得する必要すらありません。 取得したアカウントが役立つのは、プロジェクトをフォークして、そのフォークにプッシュするときです。

SSH を使って接続したい場合は、公開鍵の設定が必要です （公開鍵をまだ持っていない場合は、 SSH 公開鍵の作成 を参照ください）。 画面右上のリンクから、アカウント設定のページを開きましょう。

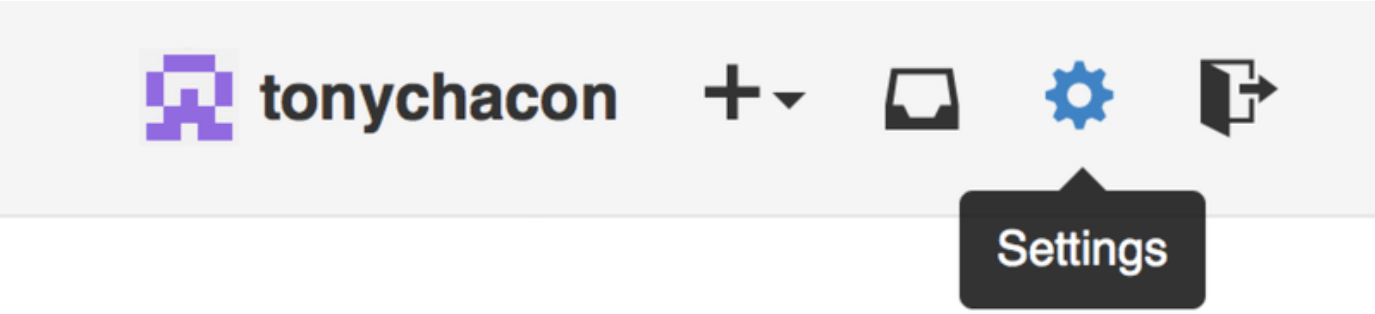



*Figure 82. “Account settings” のリンク*

そして、左側にある “SSH keys” を選択します。

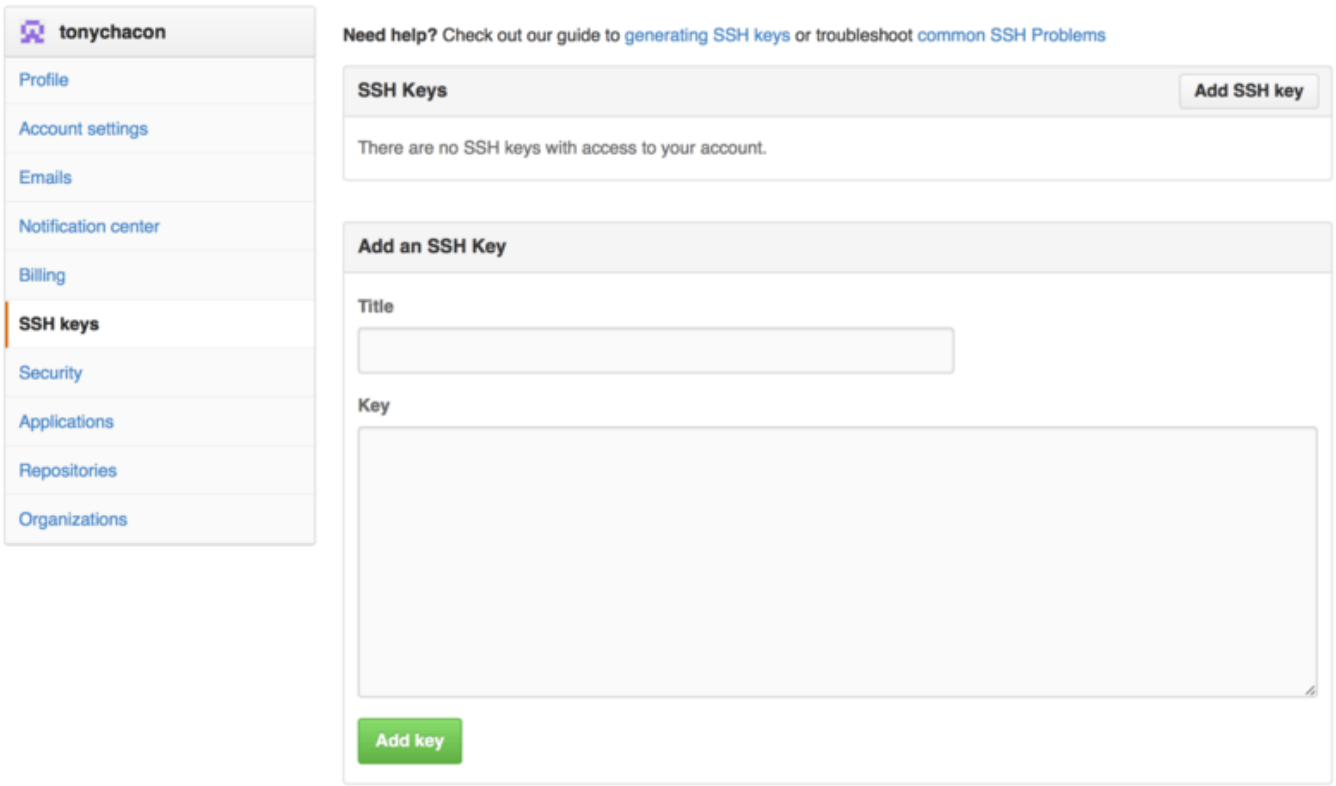



*Figure 83. “SSH keys” のリンク*

ここで "`Add an SSH key`" ボタンをクリックし、鍵の名前を入力し、自分の公開鍵ファイル (`~/.ssh/id_rsa.pub` あるいは、自分で設定したその他の名前) の内容をテキストエリアに貼り付けて、“Add key” をクリックします。

| | |
|---|---|
| **NOTE** | 鍵の名前は、自分で覚えやすいものにしておきましょう。鍵ごとに「ラップトップ」「仕事用」などの名前をつけておけば、 後で鍵を破棄することになったときに、どれを破棄すればいいのかがわかりやすくなります。 |

## アバター

自分のアカウント用のアバターとして、好きな画像を指定することもできます。 まずは、SSH key タブの上にある “Profile” タブを開き、“Upload new picture” をクリックしましょう。

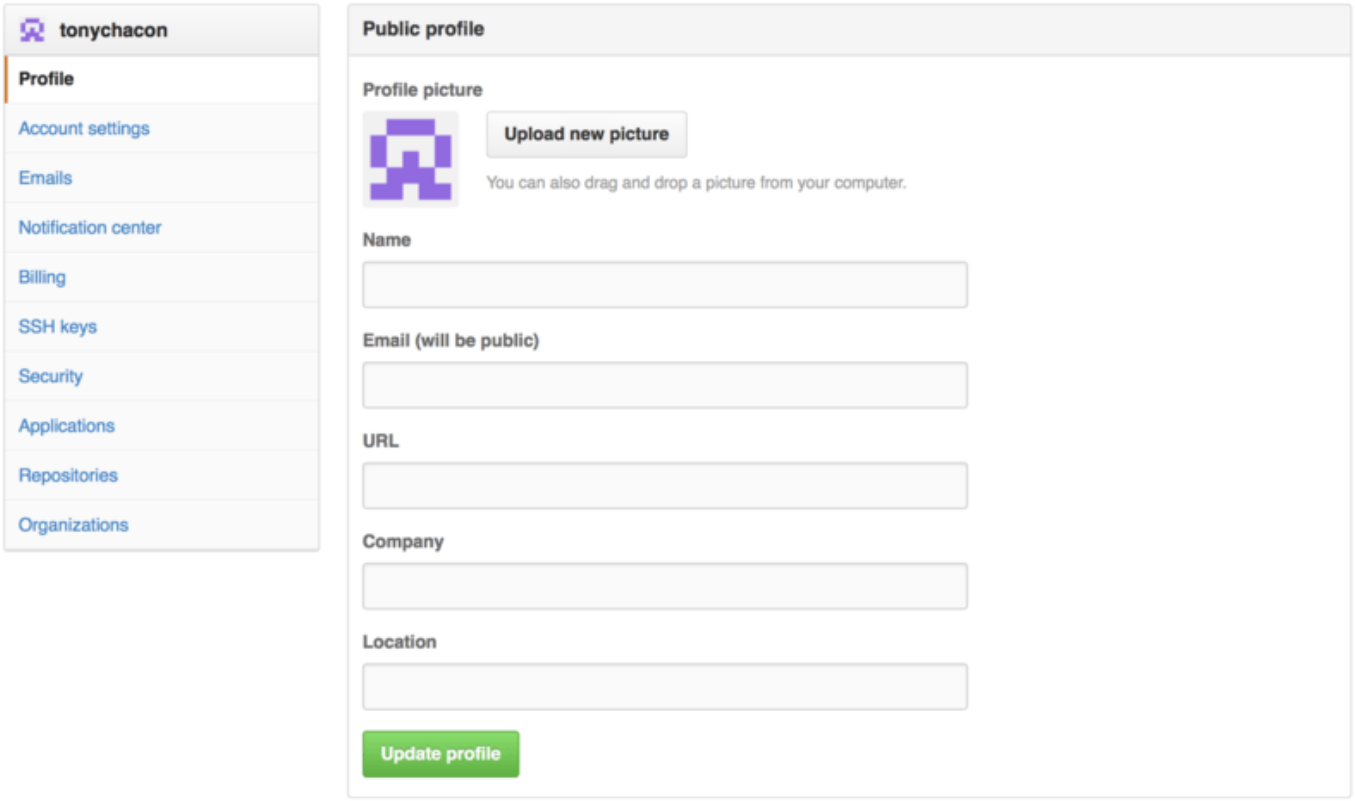



*Figure 84. “Profile” のリンク*

ハードディスク上にある Git のロゴを選ぶと、必要な部分だけを切り抜けるようになります。

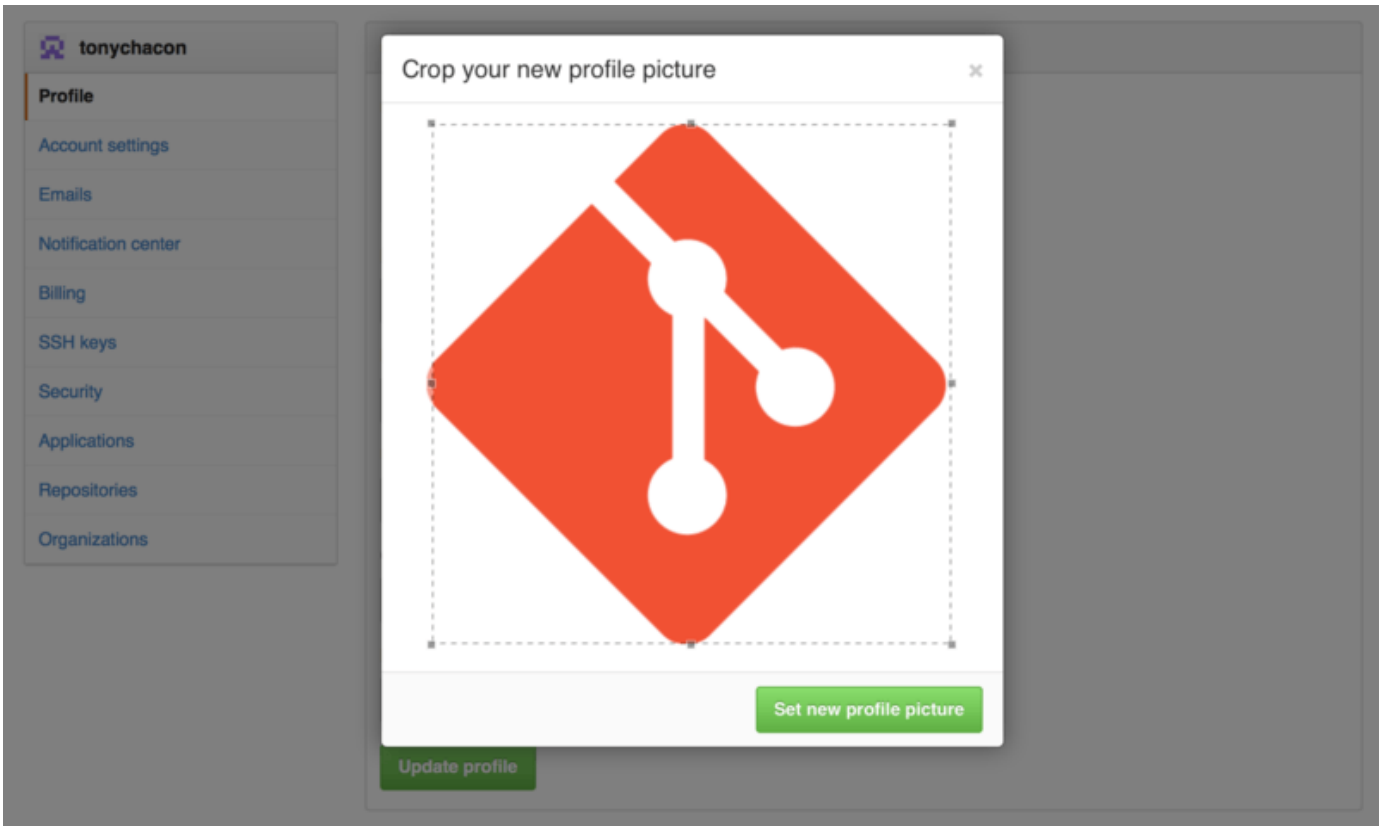



*Figure 85. アバターの切り抜き*

これで、GitHub 上で何かのアクションを起こしたときに、あなたのユーザー名の隣にその画像が表示されるようになりました。

もしすでに Gravatar にアバターを登録している場合 (Wordpress のアカウントを持っている人の多くが、Gravatar を使っています) は、デフォルトでそのアバターが使われるので、何もする必要がありません。

## メールアドレス

GitHub が Git のコミットとユーザーを紐付けるときに使うのが、メールアドレスです。 複数のメールアドレスを使い分けてコミットしているときに、それをあなたのアカウントに適切にリンクさせるためには、 すべてのメールアドレスを管理画面の Emails セクションで登録する必要があります。

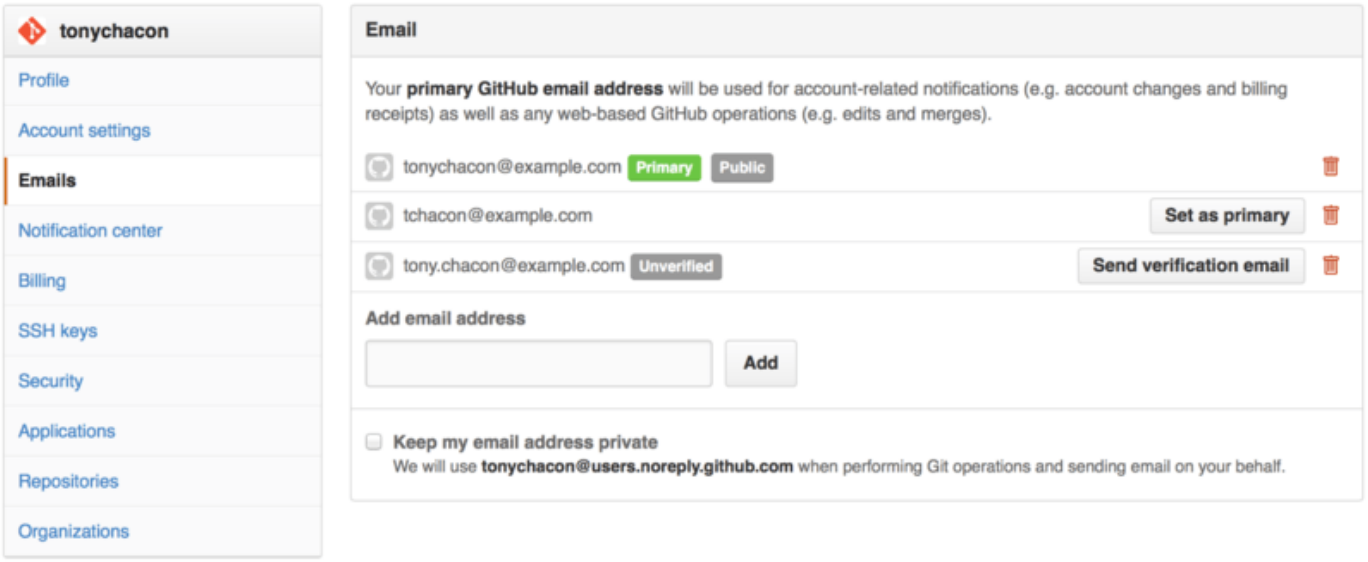



*Figure 86. メールアドレスの追加*

メールアドレスの追加 を見ると、メールアドレスにはさまざまな状態があることがわかります。 最初のアドレスは検証済みで、プライマリアドレスとして設定されています。つまり、各種の通知や有償プランの領収書などが、このアドレスに届くということです。 二番目のアドレスも検証済みです。もしプライマリアドレスをこちらに変更したい場合は、切り替えることができます。 最後のアドレスは未検証です。検証済みになるまでは、これをプライマリアドレスにすることはできません。 GitHub のサイト上にこれらのメールアドレスを含むコミットがあった場合、それがあなたのアカウントと関連づけられます。

## 二要素認証

最後に、セキュリティ高めるために、二要素認証（“2FA”）の設定をしておきましょう。 二要素認証とは、認証方式のひとつで、最近よく使われるようになりつつあります。 この方式を使うと、仮に何らかの方法でパスワードが盗まれてしまった場合でも、アカウントを乗っ取られるリスクを減らせます。 二要素認証を有効にすると、GitHub は複数の方法による認証を行うようになります。 仮にその一方の情報が盗まれたとしても、それだけでは、攻撃者があなたのアカウントにアクセスすることはできないのです。

二要素認証の設定は、アカウント設定画面の「Security」タブの中にあります。

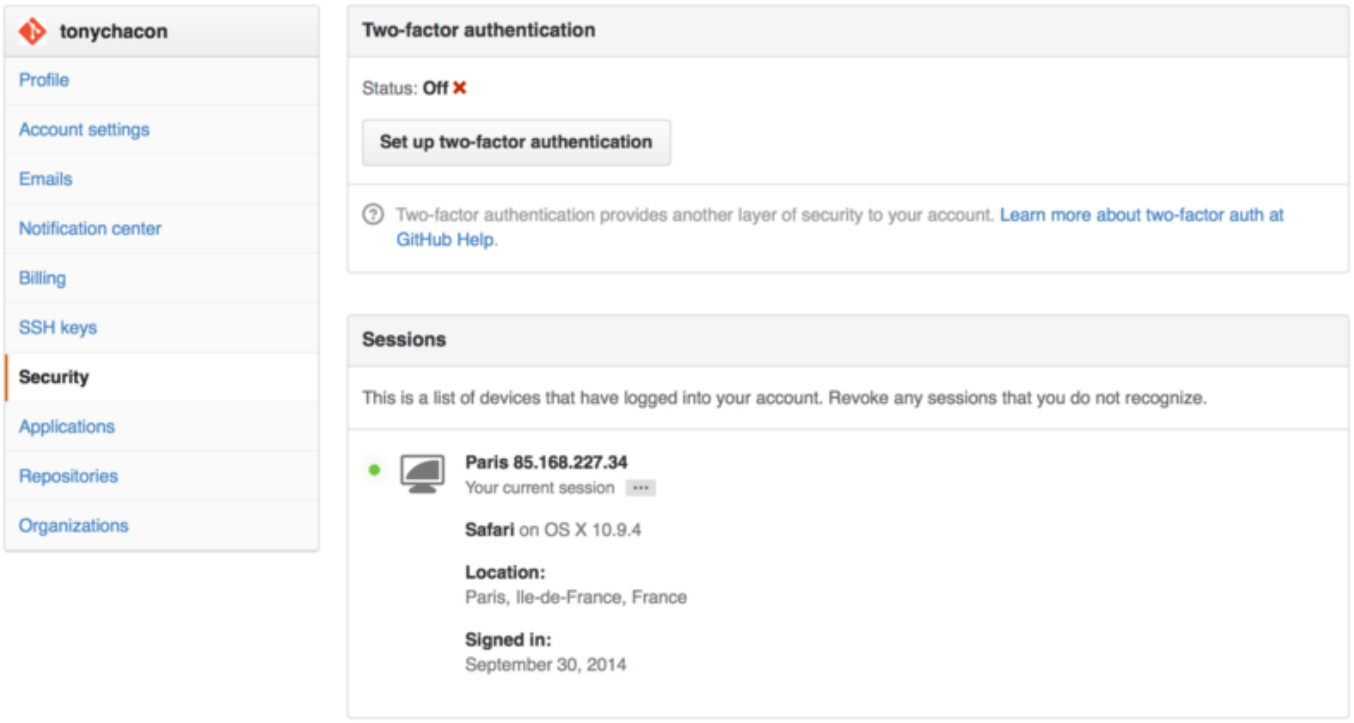



*Figure 87. Security タブの二要素認証*

“Set up two-factor authentication” ボタンをクリックすると、設定ページに移動します。 ここでは、スマホアプリを使ってセキュリティコード（“時刻ベースのワンタイムパスワード”）を設定するか、 ログインのたびに GitHub からの SMS でコードを受け取るようにするのかを選べます。

いずれかのお好みの方法を選び、指示に従って二要素認証を設定し終えたら、 あなたのアカウントは今までよりも少しだけ安全になります。 ただし、GitHub にログインするときには、パスワードだけでなくセキュリティコードも必要になります。

# プロジェクトへの貢献

これでアカウントが用意できたので、次は、既存のプロジェクトへの貢献にあたって役立つであろうことを説明していきましょう。

## プロジェクトのフォーク

既存のプロジェクトに貢献したいけれども、そのリポジトリにプッシュする権限がないという場合は、プロジェクトを「フォーク」できます。 「フォーク」するとは、GitHub があなた専用にそのプロジェクトのコピーを作るということです。あなた自身の名前空間に置かれるので、そこには自分でプッシュできます。

**NOTE**

歴史的に、この「フォーク」という用語はあまり好ましくない意味で使われてきました。 何かのオープンソースプロジェクトの方針を気に入らない人が、別の道を歩み出すこと （そして時には、競合するプロジェクトを作って、貢献者を引き抜いてしまうこと）を指していたのです。 GitHub における「フォーク」とは、単にあなたの配下に作られるコピー以外の何者でもありません。 自分自身による変更を公開の場でそのプロジェクトに適用でき、よりオープンなやりかたでプロジェクトに貢献できるようにするための手段なのです。

この方式なら、協力してくれる人たちにいちいちプッシュアクセス権を付与していく必要はありません。 それぞれがプロジェクトをフォークして、そこにプッシュして、その変更を元のリポジトリに提供したければ、いわゆる「プルリクエスト」を作ればいいのです。 プルリクエストについては、後ほど説明します。 プルリクエストを作ると、そこにコードレビューのスレッドが立ち上がります。 プロジェクトのオーナーとプルリクエストの作者は、そこで変更についての議論を重ねて、 オーナーが納得した時点で、それをマージすることができます。

プロジェクトをフォークするには、プロジェクトのページに行って、ページ右上にある`‘Fork’'ボタンを押します。

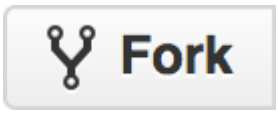



*Figure 88. “Fork” ボタン*

数秒後、新しいプロジェクトのページに自動的に移動します。これは、あなた自身が書き込み可能なコピーです。

## GitHub Flow

GitHub は、プルリクエストを中心としたコラボレーションのワークフローを想定して作られています。 ひとつのリポジトリを共有する密接に連携したチームでの作業であっても、世界中に広がる企業や個人が関わるプロジェクトで何十ものフォークがあるプロジェクトであっても、 このワークフローはうまく機能します。 その中心になるのが、Git のブランチ機能 でとりあげた トピックブランチ のワークフローです。

全体的な流れは、以下のようになります。

1. `master` からトピックブランチを作る。
2. そこに、プロジェクトの改良につながるコミットをする。
3. このブランチを、自分の GitHub プロジェクトにプッシュする。
4. GitHub 上でプルリクエストを作る。
5. 議論を重ね、必要ならさらにコミットをする。
6. プロジェクトのオーナーは、プルリクエストをマージする（あるいは、マージせずに閉じる）。

これは基本的に、統合マネージャー型のワークフロー でとりあげる、統合マネージャー型のワークフローです。 しかし、変更についてのやりとりやレビューをメールで行う代わりに、ここでは GitHub のウェブベースのツールを使います。

GitHub で公開しているオープンソースのプロジェクトに対して、このフローを使って変更を提案する例を見ていきましょう。

### プルリクエストの作成

自分のArduino上で実行するコードを探していたトニーは、GitHub 上にすばらしいプログラムがあることを発見しました。 それが https://github.com/schacon/blink です。

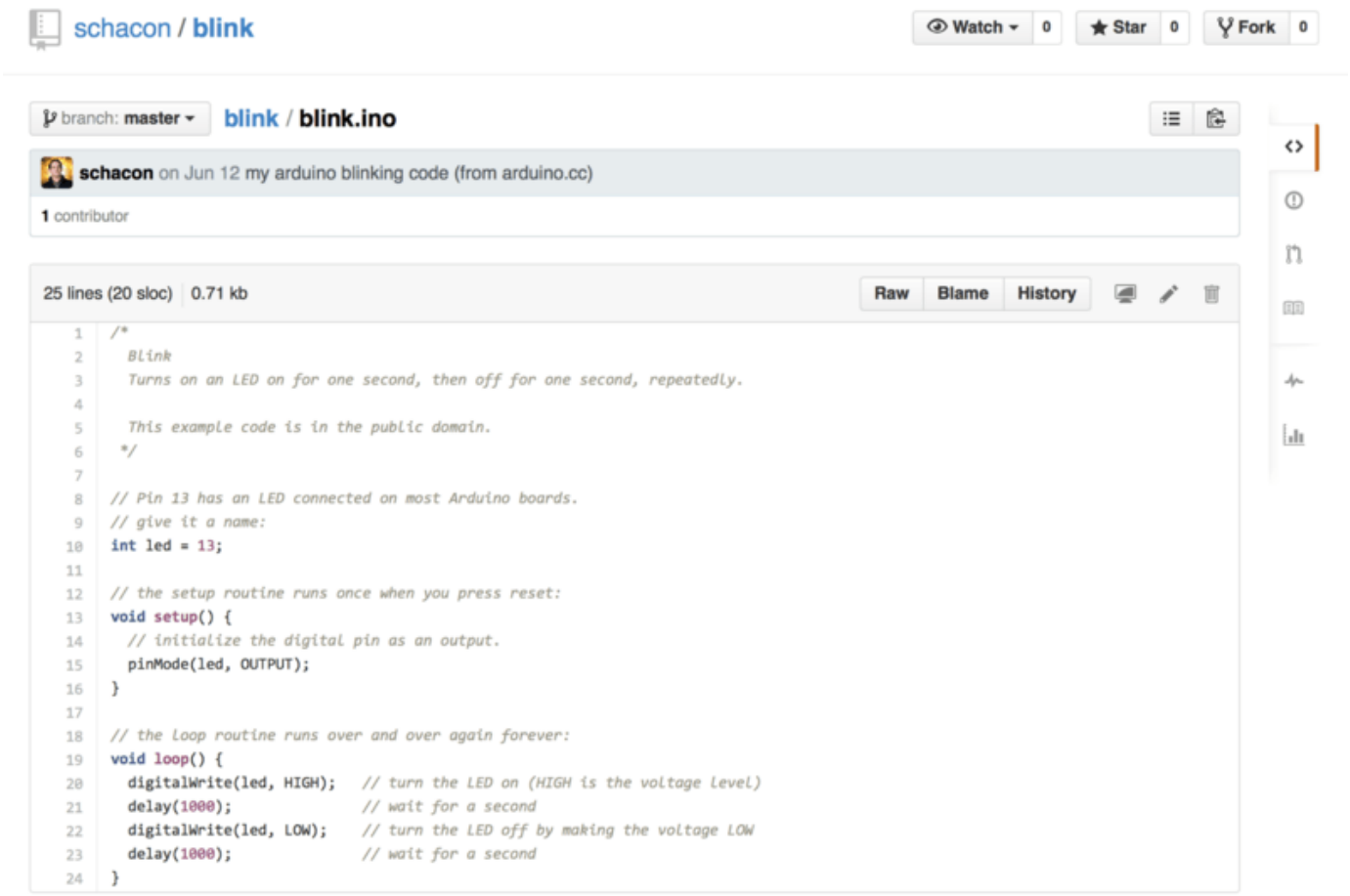



*Figure 89. 貢献したいプロジェクト*

ただ、ひとつ問題がありました。点滅の間隔が速すぎるのです。1 秒おきに状態を切り替えるのではなく、3 秒くらいは間を置きたいものです。 さて、このプログラムを改良して、その変更を提案してみましょう。

まずは、先ほど説明した *Fork* ボタンをクリックして、このプロジェクトのコピーを手に入れます。 この例で使うユーザー名は “tonychacon” とします。つまり、できあがったコピーは `https://github.com/tonychacon/blink` となり、ここからはこのプロジェクトを変更していきます。これをローカルにクローンして、トピックブランチを作り、コードを変更して、その変更を GitHub にプッシュしましょう。

```
$ git clone https://github.com/tonychacon/blink ①
Cloning into 'blink'...

$ cd blink
$ git checkout -b slow-blink ②
Switched to a new branch 'slow-blink'

$ sed -i '' 's/1000/3000/' blink.ino ③

$ git diff --word-diff ④
diff --git a/blink.ino b/blink.ino
index 15b9911..a6cc5a5 100644
--- a/blink.ino
+++ b/blink.ino
@@ -18,7 +18,7 @@ void setup() {
// the loop routine runs over and over again forever:
void loop() {
  digitalWrite(led, HIGH);   // turn the LED on (HIGH is the voltage
level)
  [-delay(1000);-]{+delay(3000);+}               // wait for a second
  digitalWrite(led, LOW);    // turn the LED off by making the voltage LOW
  [-delay(1000);-]{+delay(3000);+}               // wait for a second
}

$ git commit -a -m 'three seconds is better' ⑤
[slow-blink 5ca509d] three seconds is better
 1 file changed, 2 insertions(+), 2 deletions(-)

$ git push origin slow-blink ⑥
Username for 'https://github.com': tonychacon
Password for 'https://tonychacon@github.com':
Counting objects: 5, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 340 bytes | 0 bytes/s, done.
Total 3 (delta 1), reused 0 (delta 0)
To https://github.com/tonychacon/blink
 * [new branch]      slow-blink -> slow-blink
```

① フォークしたプロジェクトを、ローカルにクローンする

② わかりやすい名前のトピックブランチを作る

③ コードを変更する

④ 問題はなさそうだ

⑤ この変更をトピックブランチにコミットする

⑥ 新しいトピックブランチを、GitHub 上のフォークに書き戻す

この状態で GitHub 上のフォークに戻ると、GitHub 上に新しいトピックブランチがプッシュされたことを伝えてくれます。 また、大きな緑色のボタンを使えば、変更点を確認したり、元のプロジェクトへのプルリクエストを送ったりできます。

あるいは、`https://github.com/<user>/<project>/branches` にある “Branches” ページから自分のトピックブランチに移動して、そこからプルリクエストを送ることもできます。

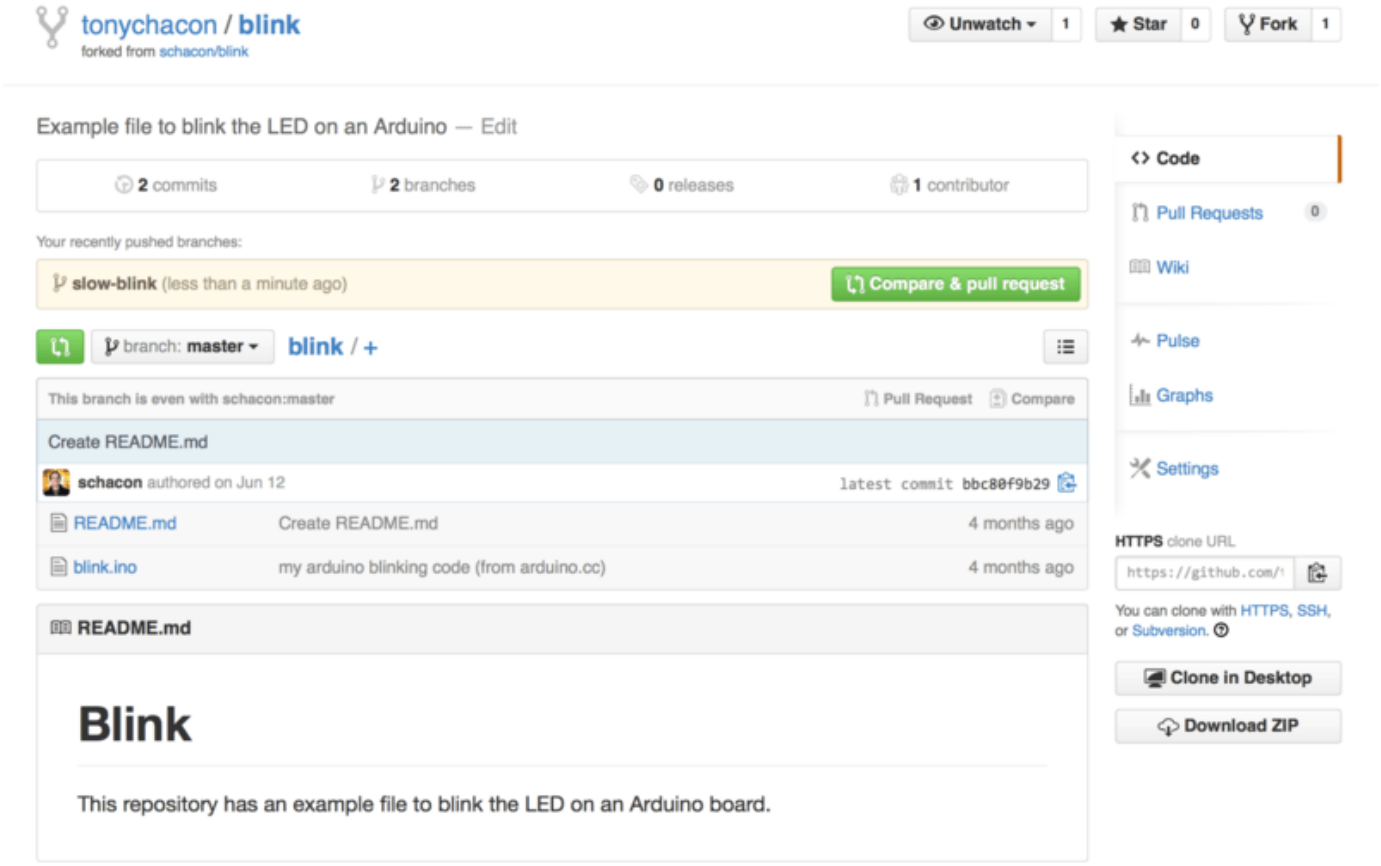



*Figure 90. プルリクエストのボタン*

この緑のボタンをクリックすると、プルリクエストのタイトルと説明を入力する画面に遷移します。 ちゃんと時間をかけて説明を書きましょう。損はしないはずです。プルリクエストを受ける側のプロジェクトオーナーからすれば、説明文がよければあなたの意図が汲み取りやすくなるからです。そうすれば、オーナーはプルリクエストの内容を正確に評価できますし、それを取り込むことがプロジェクトにとってプラスかどうかを判断できるでしょう。

この画面では、トピックブランチ内のコミットのうち、`master`よりも先行しているコミットの一覧 (今回の場合はひとつだけ) も確認できます。 また、このブランチをオーナーがマージしたときに適用される変更の、unified形式の差分も表示されます。

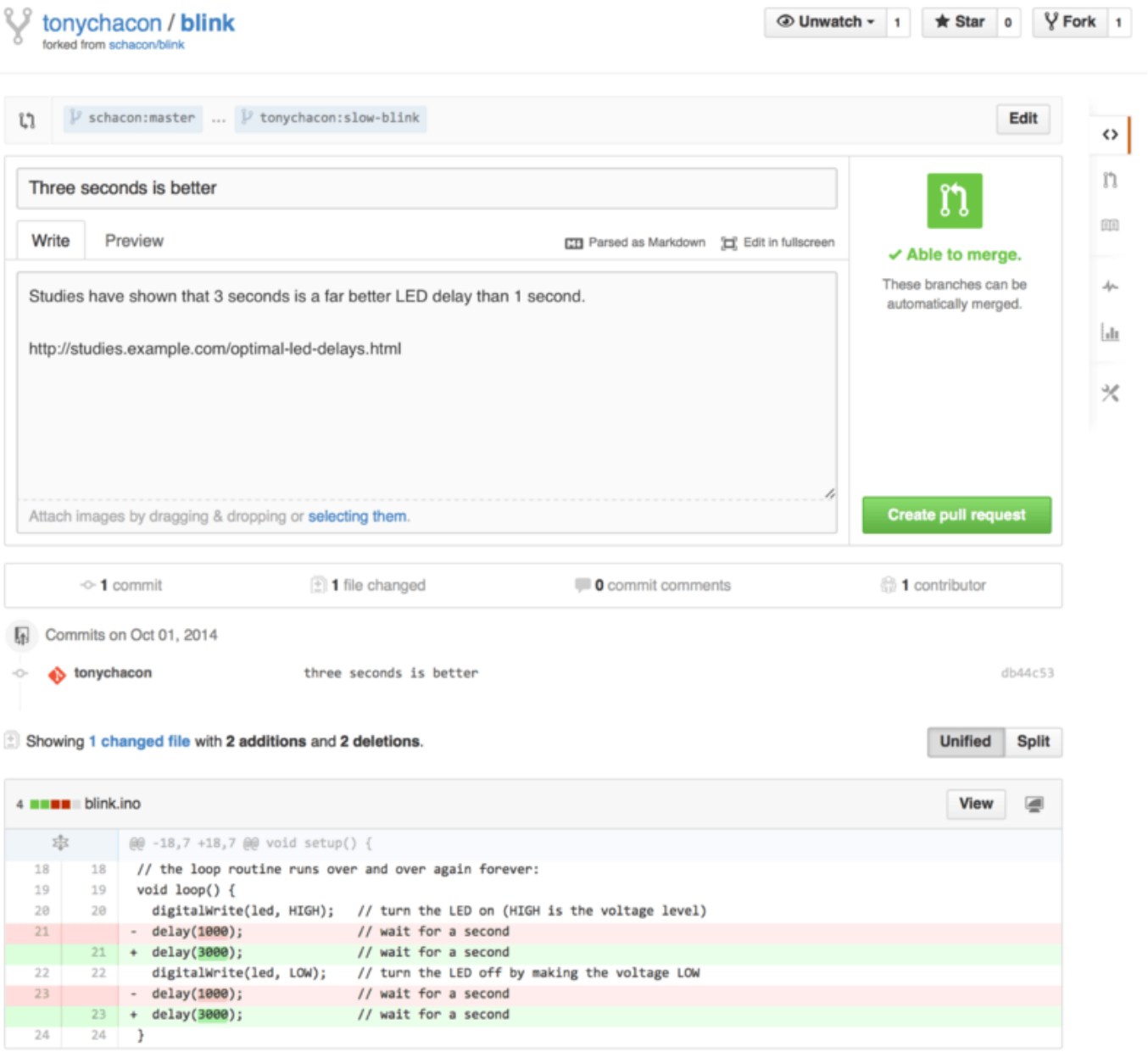



*Figure 91. プルリクエストの作成ページ*

この画面で *Create pull request* ボタンを押すと、フォーク元のプロジェクトのオーナーに、 誰かが変更を提案しているという通知が届きます。この通知には、変更に関するすべての情報が記載されたページへのリンクが含まれています。

**NOTE**

一般にプルリクエストは、こういった公開プロジェクトに対する変更を、その準備が整った時点で提案するために作るものです。 しかし、内部的なプロジェクトの開発サイクルにおいて、 *開発を始めるタイミングで* プルリクエストを作ることもよくあります。 プルリクエストを作った*後でも*、そのトピックブランチへのプッシュを続けることができます。 最後の最後にプルリクエストを行うのではなく、早い時点でプルリクエストを作っておけば、 その後の作業状況をチーム内で共有できます。

### プルリクエストの繰り返し

これで、元のプロジェクトのオーナーは、変更の提案を見られるようになりました。それをマージしたり、却下したり、コメントしたりすることができます。 ここでは、オーナーが変更提案を気に入ったものの、ライトが消えている時間を点灯している時間よりも少しだけ長くしたほうがいいと感じたことにしましょう。

Git での分散作業 のワークフローなら、この手のやりとりはメールで行うところですが、GitHub の場合はこれをオンラインで行います。 プロジェクトのオーナーはunfied diffをレビューして、コメントを残します。コメントしたい行をクリックすれば、コメントを残せます。

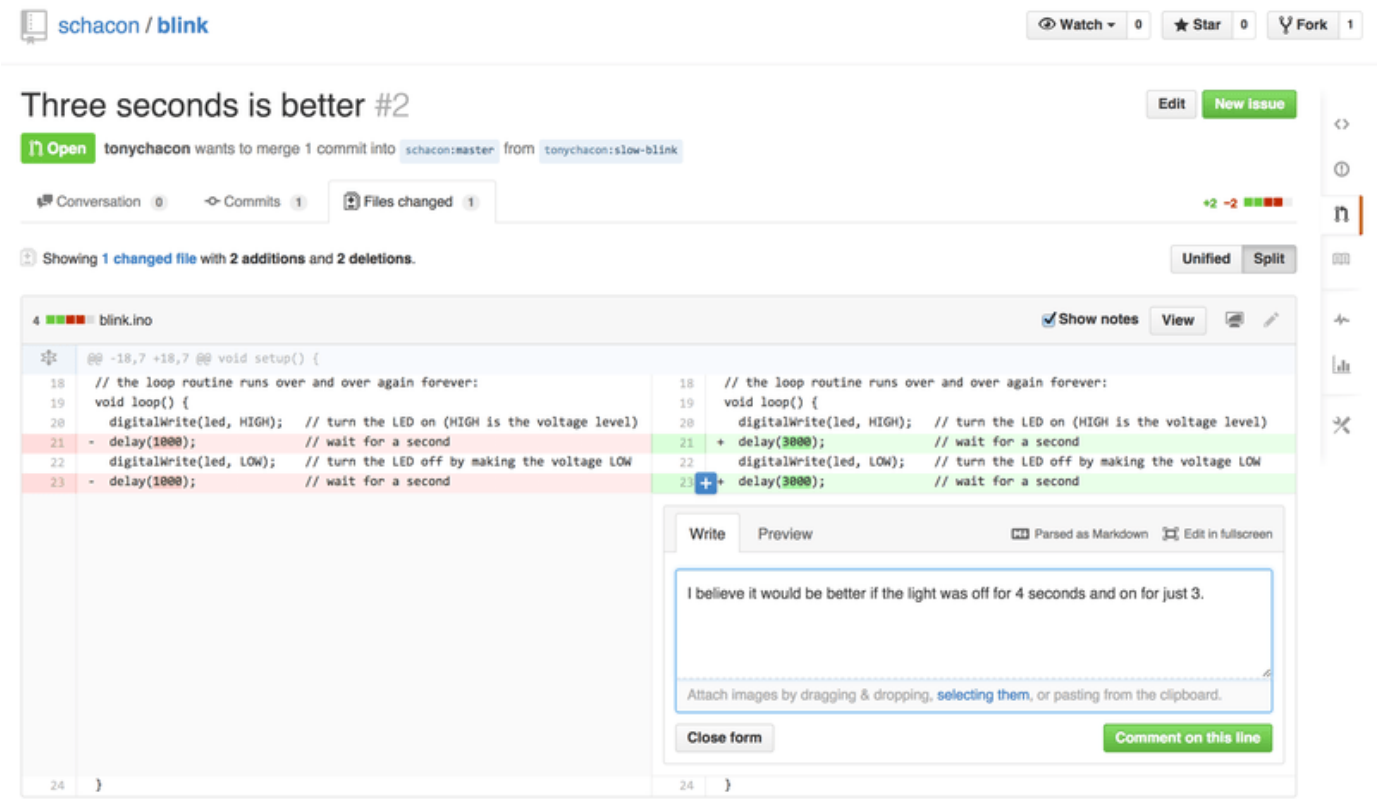



*Figure 92. プルリクエストのコードの特定の行へのコメント*

メンテナがコメントを入れると、プルリクエストの作者 (そして、そのリポジトリをウォッチしているすべての人たち) に、通知が届きます。 通知をカスタマイズする方法については後述しますが、メールでの通知を受け取るように設定している場合は、以下のようなメールも届きます。


*Figure 93. 通知メールで送られたコメント*

オーナーだけでなく誰でも、プルリクエスト全体に対するコメントができます。 プルリクエストのディスカッションページ では、プロジェクトのオーナーがコードの特定の行についてコメントしたうえで、さらにプルリクエスト全体に関するコメントも残しています。 また、コードへのコメントが、一連の会話に組み込まれていることにもお気づきでしょう。

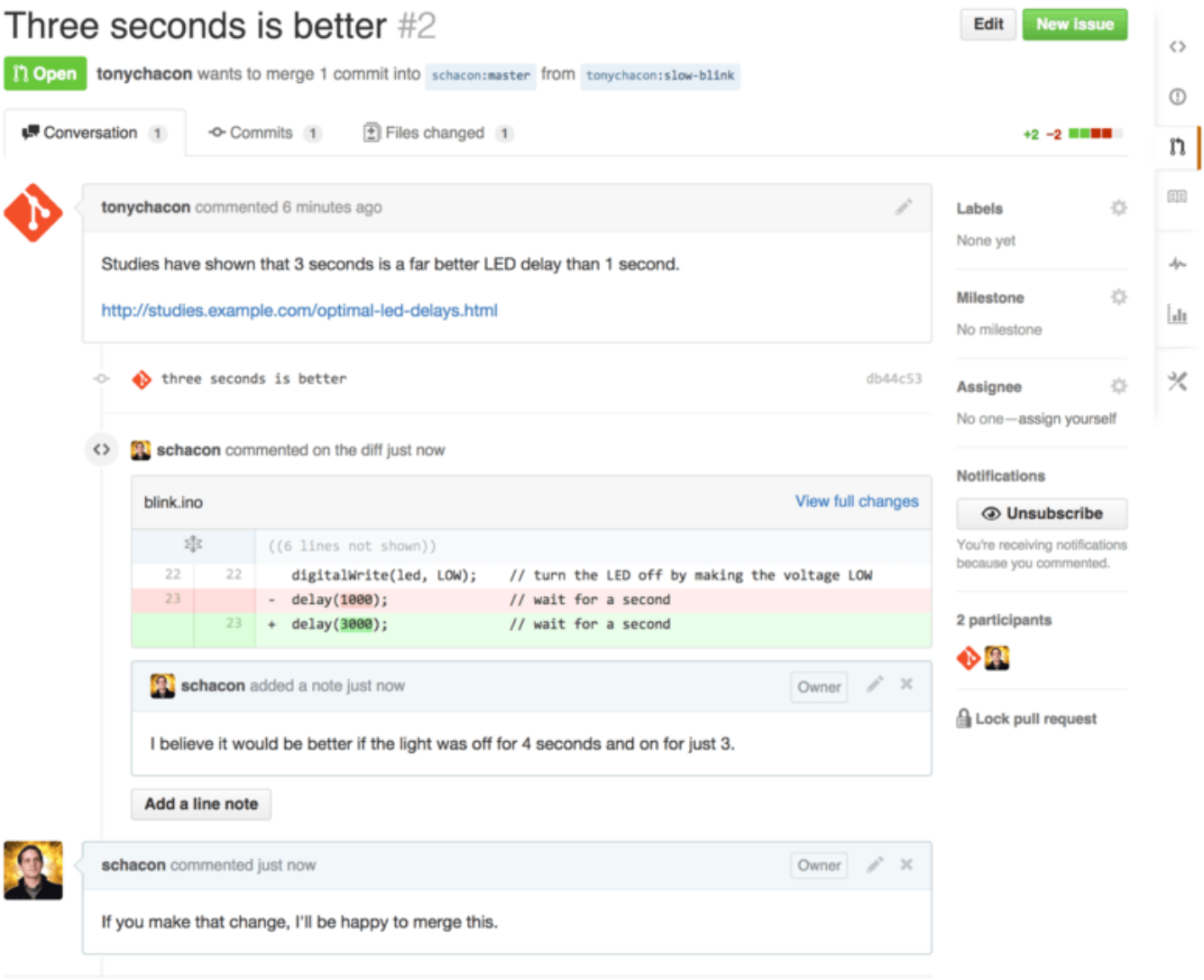



*Figure 94. プルリクエストのディスカッションページ*

プルリクエストの作者は、自分の変更を受け入れてもらうために何が必要なのかがわかりました。 幸運にも、そんなに手間のかかることではありません。 メールでのやりとりの場合は、一連の作業をやり直した上でもう一度メーリングリストに投稿する必要がありますが、 GitHub なら、単にトピックブランチにコミットしてそれをプッシュするだけで済みます。 また、プルリクエストの最終形 にあるように、更新されたプルリクエストでは変更前のコードへのコメント表示が省略されています。追加されたコミットによって変更されたコードへのコメントだからです。

なお、既存のプルリクエストにコミットを追加しても、通知は送られません。そこで、修正をプッシュしたトニーは、修正が終わったことをコメントでプロジェクトオーナーに伝えることにしました。

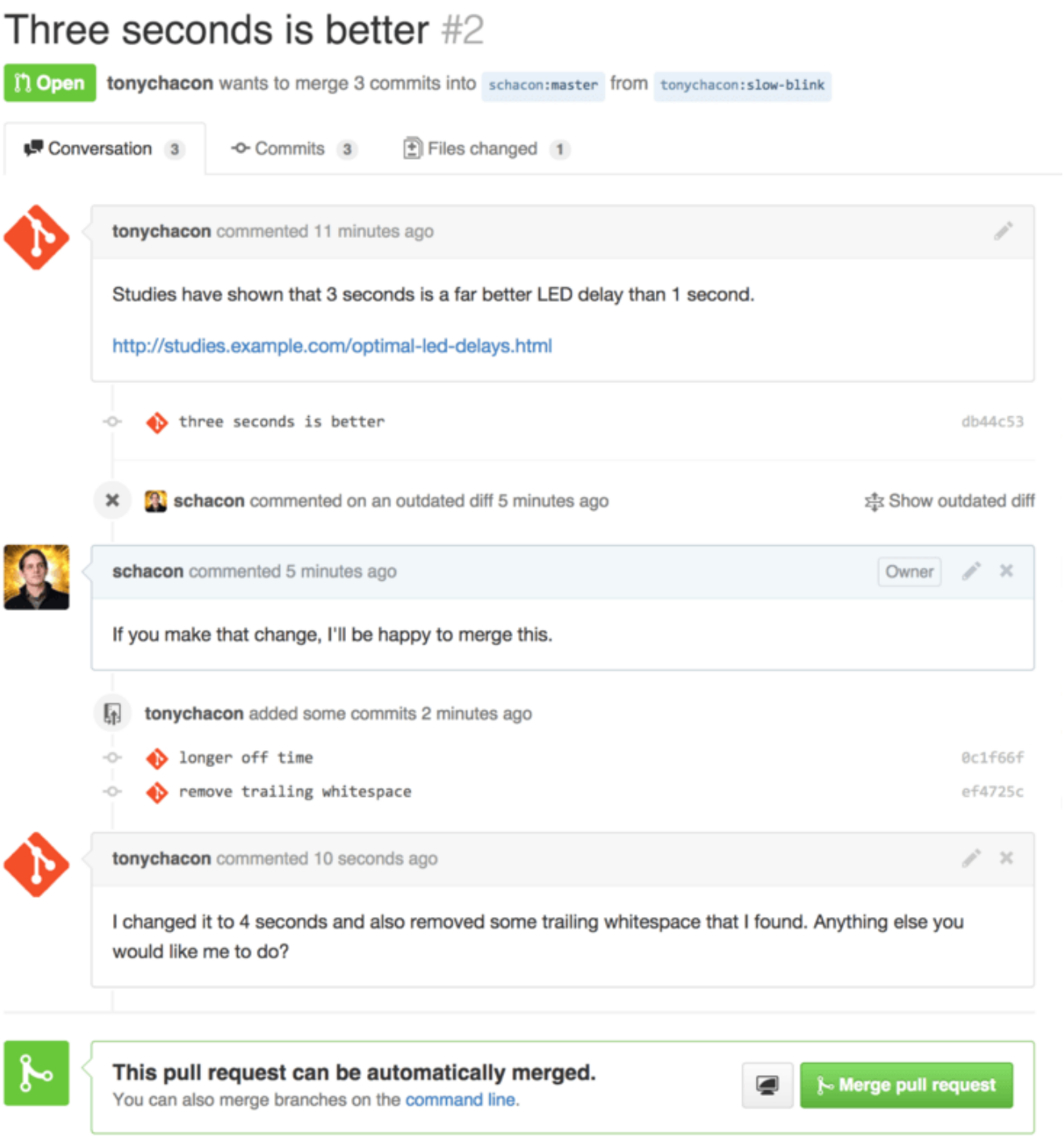



*Figure 95. プルリクエストの最終形*

このプルリクエストのページで “Files Changed” タブをクリックすると、“unified” 形式の diff を確認できます。 つまり、このトピックブランチをマージしたときにどんな変更が施されるのかを、まとめて確認できるのです。 `git diff` の用語に直すと、このタブを開いたときに表示される内容は、プルリクエストの対象になっているブランチ上で `git diff master...<branch>` を実行した結果になります。 この形式の diff についての詳細は、何が変わるのかの把握 を参照ください。

もうひとつお気づきのことがあることでしょう。 GitHub は、このプルリクエストが問題なくマージできることを確認したうえで、サーバー上でマージを実行するためのボタンを表示します。 このボタンが表示されるのは、あなたがこのリポジトリへの書き込みアクセス権限を持っていて、かつ問題なくマージ可能な場合だけ

です。 このボタンをクリックすると、GitHub は “non-fast-forward” なマージを行います。 つまり、仮に fast-forward 可能なマージであったとしても、明示的にマージコミットを作ります。

お望みなら、このブランチを取得した上で、ローカルでマージすることもできます。 このブランチを `master` にマージしてから GitHub にプッシュすると、このプルリクエストは自動的に閉じられます。

これが、大半の GitHub プロジェクトが使っている基本的なワークフローです。 トピックブランチを作り、そこからプルリクエストを作って、議論を重ね、必要に応じてさらに作業を重ねて、最終的にそのリクエストをマージするか、あるいはマージせずに終了します。

**NOTE**

*フォークしなくてもかまわない*

同じリポジトリのふたつのブランチ間でのプルリクエストもできるということを知っておきましょう。 誰かと一緒に何らかのフィーチャーの作業をしていて、両方ともそのプロジェクトへの書き込み権限を持っている場合なら、 トピックブランチをそのリポジトリにプッシュした上で、同じプロジェクトの `master` ブランチへのプルリクエストを作ることができます。 そこで、コードのレビューや議論を進めればいいでしょう。 このときに、わざわざフォークする必要はありません。

## プルリクエストの応用テクニック

GitHub のプロジェクトに貢献する際の基本がわかったところで、 プルリクエストに関するちょっとしたヒントやテクニックを紹介しましょう。これらを使えば、プルリクエストをさらに活用できるでしょう。

### パッチとしてのプルリクエスト

実際のところ、多くのプロジェクトは、プルリクエストを完璧なパッチ群である (つまり、きちんと順序どおりに適用しなければいけない) とは考えていません。 これは、メーリングリストベースで運営するプロジェクトで一般的な考えかたとは異なります。 GitHub のプロジェクトでは、プルリクエストのブランチを変更提案に関する議論の場と捕らえていることが多く、 最終的にできあがった unified diff をマージするのだと考えています。

この違いを認識しておくことが大切です。一般に、変更を提案するのは、コードが完璧に仕上がる前の段階です。 一方、メーリングリストベースの運営では、まだできあがってもいないパッチを投稿することなど、まずないでしょう。 未完成の段階で変更を提案することで、メンテナとの議論を早めに始めることができます。 コミュニティの協力で、より適切なソリューションにたどり着けるようになるでしょう。 プルリクエストで提案したコードに対してメンテナやコミュニティから変更の提案があったときに、 パッチをゼロから作り直す必要はありません。 差分だけを、新たなコミットとしてプッシュすればいいのです。 その後の議論は、これまでの経緯を踏まえた上で進みます。

プルリクエストの最終形 をもう一度見てみましょう。プルリクエストの作者は、自分のコミットをリベースして新たなプルリクエストを作ったわけではありません。 単に、新しいコミットを追加して、それを既存のブランチにプッシュしただけです。 そのおかげで、今後このプルリクエストのページを見直すことがあったときにも、最終的な決定に至るまでの経緯を簡単に確認できるのです。 “Merge” ボタンを押したときに、本来不要な場面でも意図的にマージコミットを作っているのは、 後からそのプルリクエストを参照しやすいようにするためです。 必要に応じて、それまでの流れをすぐに調べることができます。

### 上流への追従

プルリクエストを作った後で元のプロジェクトに変更が加わったなどの理由で、プルリクエストがそのままではマージできなくなることがあります。 そんな場合は、そのプルリクエストを修正して、メンテナがマージしやすいようにしておきたいことでしょう。 GitHub は、そのままでマージできるかどうかをチェックして、すべてのプルリクエストのページの最下部に結果を表示します。

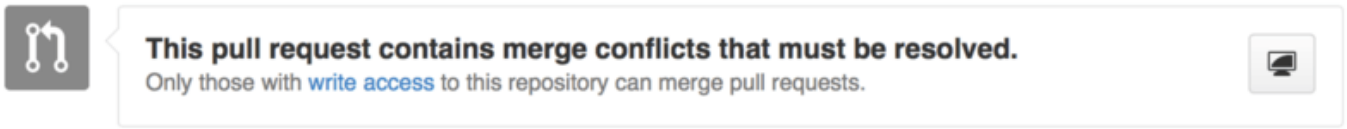



*Figure 96. そのままではマージできないプルリクエスト*

そのままではマージできないプルリクエスト のようになっていたら、自分のブランチを修正して、この表示がグリーンになるようにしたいところです。 そうすれば、メンテナに余計な手間をかけさせずに済みます。

グリーンにするための主な選択肢は、二種類あります。 ひとつは、自分のブランチを、プルリクエストの対象ブランチ (普通は、フォーク元のリポジトリの `master`) の先端にリベースすること。 もうひとつは、その対象ブランチを自分のブランチにマージすることです。

GitHub 上の開発者の多くは、後者を選んでいるようです。その理由は、先述したとおりです。 重要なのは、そこにいたるまでの歴史と、最終的にマージしたという事実だと考えているのでしょう。 リベースをすると、歴史がすっきりするという以外の利点はありません。そして、リベースはマージに比べて **ずっと** 難しいし、間違いを起こしやすいものです。

対象ブランチをマージして、自分のプルリクエストをそのまま取り込んでもらえるようにする手順は、次のとおりです。 まず、オリジナルのリポジトリを新しいリモートとして追加して、それをフェッチします。 そして、そのリポジトリのメインブランチを自分のトピックブランチにマージします。 何か問題があれば修正し、その結果をプルリクエストと同じブランチにプッシュします。

先ほどの “tonychacon” の例に戻りましょう。プルリクエストを出した後にオリジナルの作者がリポジトリに変更を加えたため、 プルリクエストがそのままでは取り込めなくなってしまいました。そんな場合の手順は、以下のとおりです。

```
$ git remote add upstream https://github.com/schacon/blink ①

$ git fetch upstream ②
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (3/3), done.
Unpacking objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 0 (delta 0)
From https://github.com/schacon/blink
 * [new branch]      master     -> upstream/master

$ git merge upstream/master ③
Auto-merging blink.ino
CONFLICT (content): Merge conflict in blink.ino
Automatic merge failed; fix conflicts and then commit the result.

$ vim blink.ino ④
$ git add blink.ino
$ git commit
[slow-blink 3c8d735] Merge remote-tracking branch 'upstream/master' \
    into slower-blink

$ git push origin slow-blink ⑤
Counting objects: 6, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (6/6), done.
Writing objects: 100% (6/6), 682 bytes | 0 bytes/s, done.
Total 6 (delta 2), reused 0 (delta 0)
To https://github.com/tonychacon/blink
   ef4725c..3c8d735  slower-blink -> slow-blink
```

① オリジナルのリポジトリを “upstream” という名前のリモートとして追加する

② そのリモートの、最新の状態をフェッチする

③ メインブランチを、自分のトピックブランチにマージする

④ 衝突を解決する

⑤ 同じトピックブランチに、再びプッシュする

これでプルリクエストが自動的に更新されて、マージ可能かどうかが再びチェックされます。

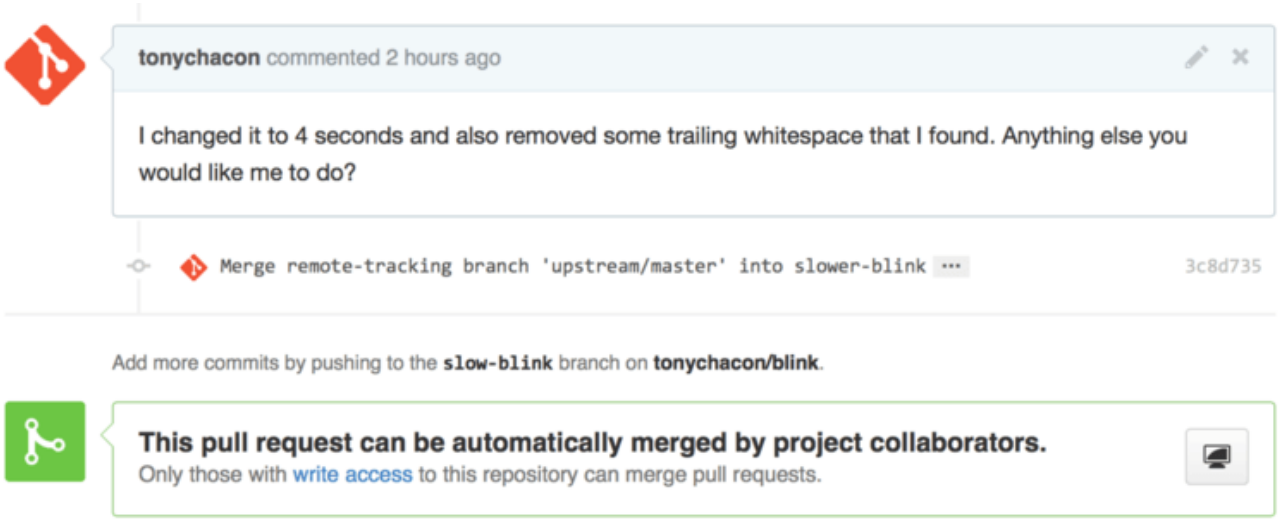



*Figure 97. そのままマージできるようになったプルリクエスト*

Git のすばらしいところのひとつが、これらの作業を継続的に行えるということです。 長期にわたるプロジェクトでも、対象ブランチからのマージを何度でも繰り返せるので、前回のマージ以降に発生した衝突さえ気をつけていれば、混乱なく作業を続けられます。

ブランチをリベースしてすっきりさせたい場合は、そうしてもかまいません。 しかし、既に作成済みのプルリクエストに対して、それを強制的にプッシュするのは避けたほうがいいでしょう。 もし他の人がそれを手元に取得して何かの作業を進めると、ほんとうは怖いリベース で説明したような問題が発生します。 リベースした場合は、それを GitHub 上で新しいブランチにして、新しいプルリクエストを作るようにしましょう。 新しいプルリクエストから元のプルリクエストを参照して、そして元のプルリクエストは閉じてしまいます。

**参照**

…と言われて気になるのは、「元のプルリクエストをどうやって参照すればいいの？」ということでしょう。 GitHub 上で他のものを参照するにはいろんな方法があって、GitHub 上で何かを書ける場所ならほぼどこでも他のものを参照できます。

まずは、別のプルリクエストあるいは Issue を相互参照する方法から紹介します。 プルリクエストや Issue には番号が振られていて、この番号はプロジェクト内で一意になっています。 つまり、たとえばプルリクエスト#3とIssue #3が *両方とも* 存在することはありえないのです。 他のプルリクエストや Issue を参照したい場合は、コメントや説明文の中で単に `#<num>` と書くだけでかまいません。 あるいは、もう少し細かく、誰か他の人が作った Issue やプルリクエストを指定することもできます。 `username#<num>` と書けば、今いるリポジトリの別のフォーク上での Issue やプルリクエストを参照できるし、 `username/repo#<num>` と書けば、別のリポジトリ上のものも参照できます。

実例を見てみましょう。 先ほど説明したとおり、リベースをした上で新しいプルリクエストを作ったものとします。新しいプルリクエストから、古いプルリクエストを参照したいところです。 また、そのリポジトリのフォーク上にある Issue や、まったく別のプロジェクトにある Issue も参照するつもりです。 説明文は、プルリクエスト内での相互参照 のようになります。

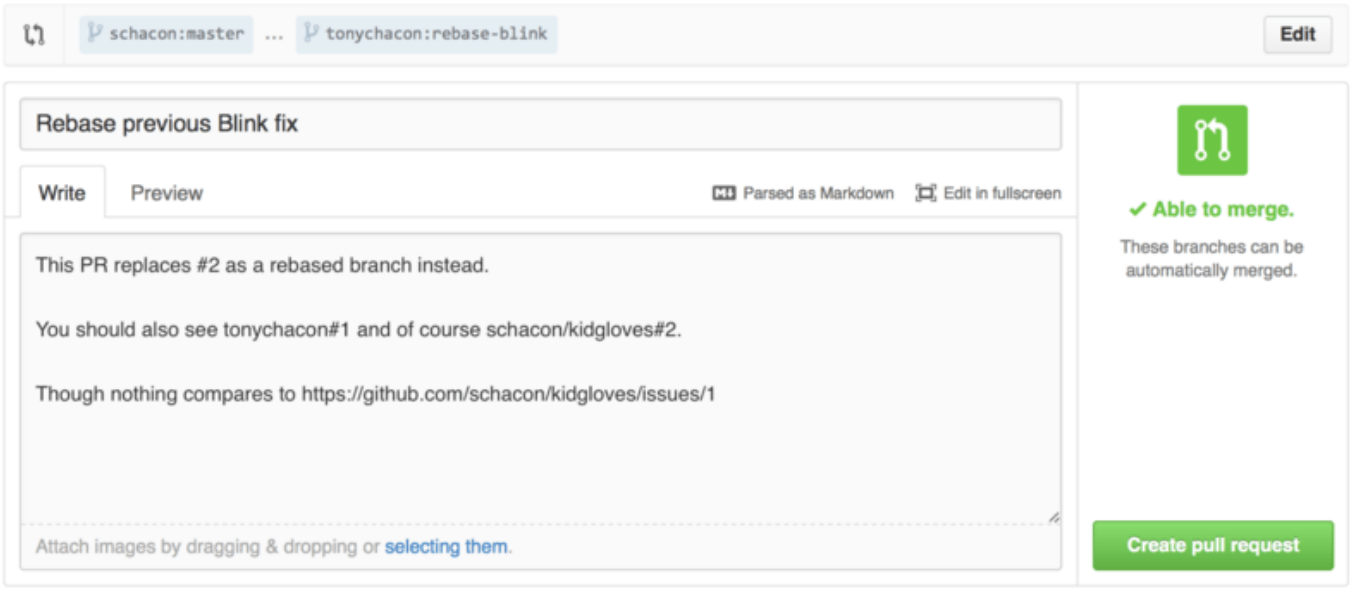



*Figure 98. プルリクエスト内での相互参照*

このプルリクエストを投稿すると、画面上では プルリクエスト内での相互参照のレンダリング のような表示になります。

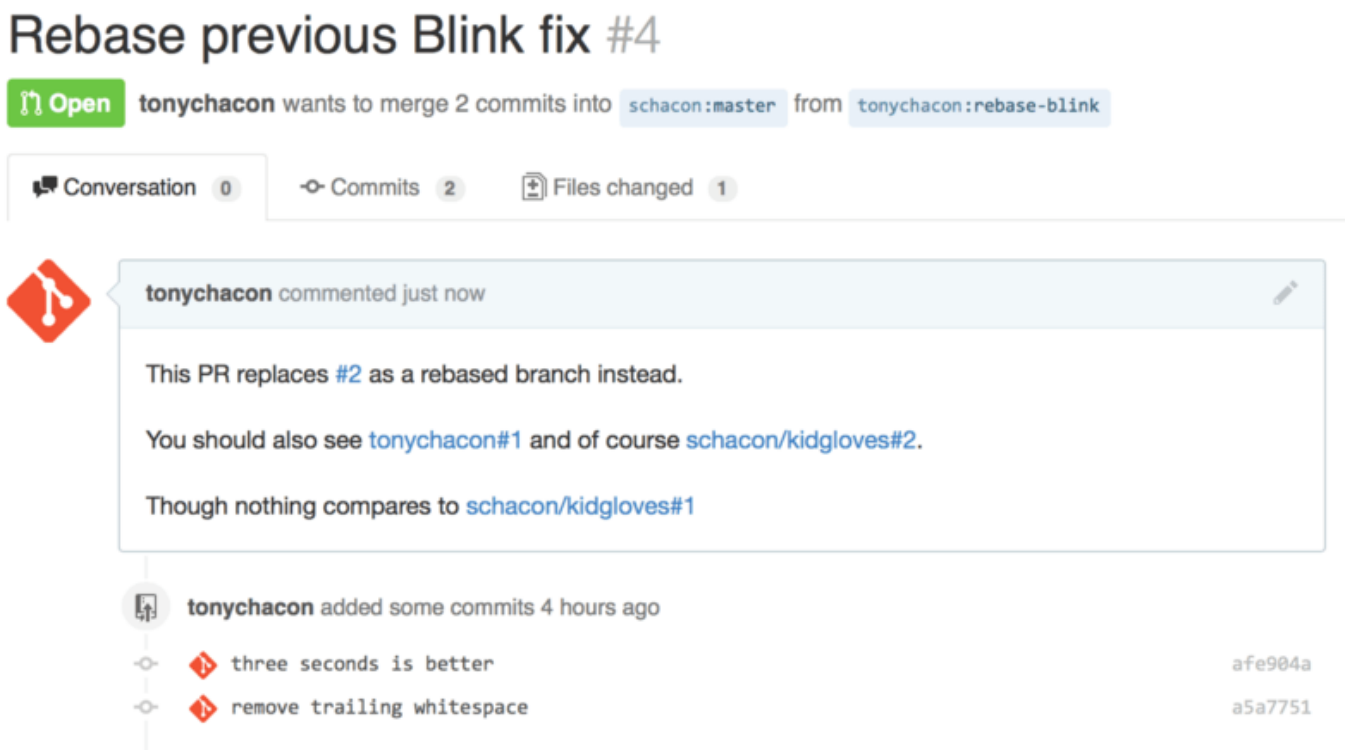



*Figure 99. プルリクエスト内での相互参照のレンダリング*

GitHub の完全な URL を入力したところも、画面上では短縮されて、必要な情報だけが見えていることがわかるでしょう。

トニーが元のプルリクエストを閉じると、そのことが新しいプルリクエストのほうにも表示されることがわかります。 GitHub が、プルリクエストのタイムラインに自動的にトラックバックを送ったのです。 これで、古

いプルリクエストを見にきたすべての人は、そのリクエストの後継となる新しいプルリクエストにたどり着けるようになるのです。 リンクは、プルリクエスト内での相互参照のレンダリング のように表示されます。

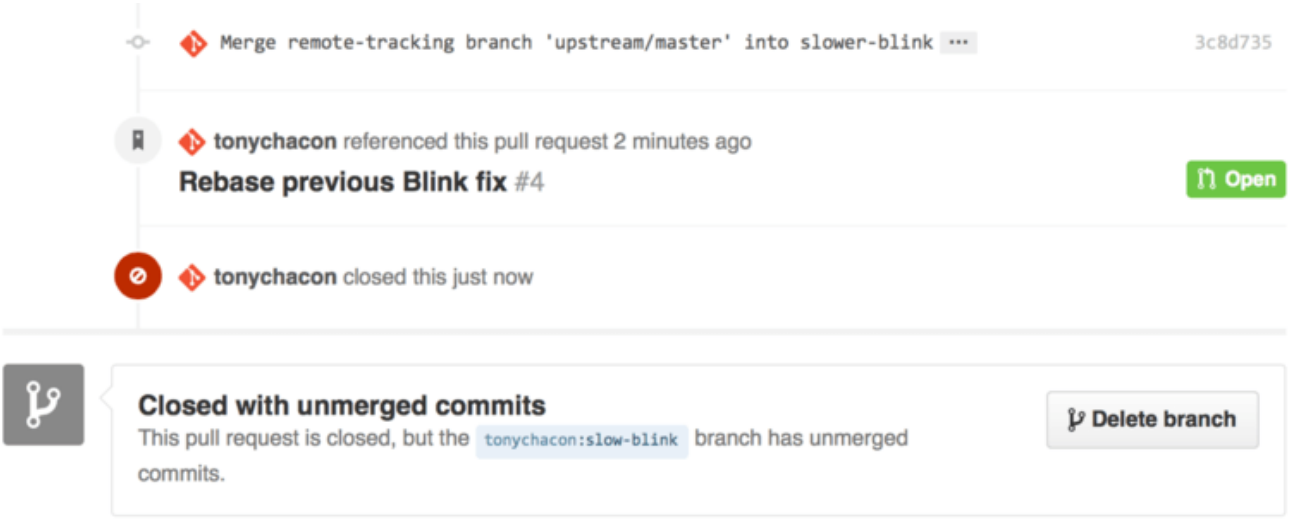



*Figure 100. プルリクエスト内での相互参照のレンダリング*

issue の番号だけでなく、SHA-1 を示して特定のコミットを参照することもできます。 SHA-1 を指定する際には 40 文字ぶんすべてを示す必要がありますが、コメントの中に SHA-1 を発見すると、GitHub はそれを当該コミットへリンクしてくれます。 他のフォークやその他のリポジトリのコミットを参照する場合の方法は、issue の場合と同じです。

## Markdown

他の Issue へのリンクは、GitHub のテキストボックスでできるさまざまなことのうちの、ほんの始まりに過ぎません。 Issue やプルリクエストの説明、それに対するコメント、コードに対するコメントなどなどでは、いわゆる “GitHub Flavored Markdown” を使うことができます。 Markdown はプレーンテキストと似ていますが、よりリッチなレンダリングを行います。

コメントや説明文を、Markdown を使って書いた例を Markdown での記述例と、そのレンダリング結果 に示します。

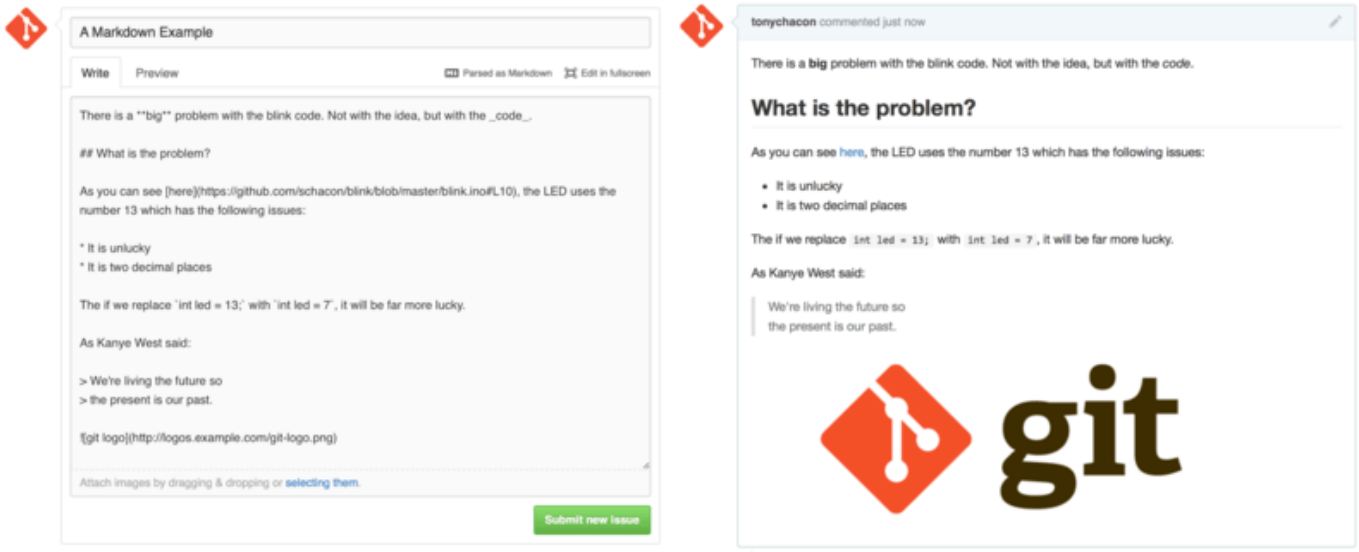



*Figure 101. Markdown での記述例と、そのレンダリング結果*

### GitHub Flavored Markdown

GitHub Flavored Markdownは、基本的なMarkdownの文法に、GitHub 流の味付けをしたものです。 プルリクエストや Issue を作ったり、それにコメントしたりするときに、役立つことでしょう。

#### タスクリスト

GitHub 流の Markdown で追加された便利な機能の中で、最初に紹介する機能が、タスクリストです。これは、プルリクエストで特に便利です。 タスクリストとは、チェックボックス付きの、やることリストです。これを Issue やプルリクエストで使うと、完了させるまでに何を済ませなければいけないのかを表せます。

タスクリストの作りかたは、以下のとおりです。

```
- [X] Write the code
- [ ] Write all the tests
- [ ] Document the code
```

プルリクエストや Issue の説明文にこのように書いておくと、Markdown でのコメント内に表示されたタスクリスト のような表示になります。


*Figure 102. Markdown でのコメント内に表示されたタスクリスト*

これはたとえば、プルリクエストに対して、「これだけのことを済ませればマージの準備が整う」ということを示すために使うことがあります。 この機能のすばらしいところは、単にチェックボックスをクリックするだけで、コメントが更新できるということです。 タスクが完了したときに、わざわざ Markdown を直接編集する必要はありません。

さらに、GitHub は、Issue やプルリクエストの中にあるタスクリストを見つけて、そのメタデータを一覧ページにも表示してくれます。 たとえば、あるプルリクエストの中でタスクを作ったときに、プルリクエストの一覧ページを見ると、タスクがどの程度完了しているのかを確認できるのです。 これは、プルリクエストをサブタスクに切り分けたり、他のひとたちがそのブランチの進捗を追いかけたりする際にも役立ちます。 この機能の実例を プルリクエスト一覧における、タスク一覧の概要表示 に示します。

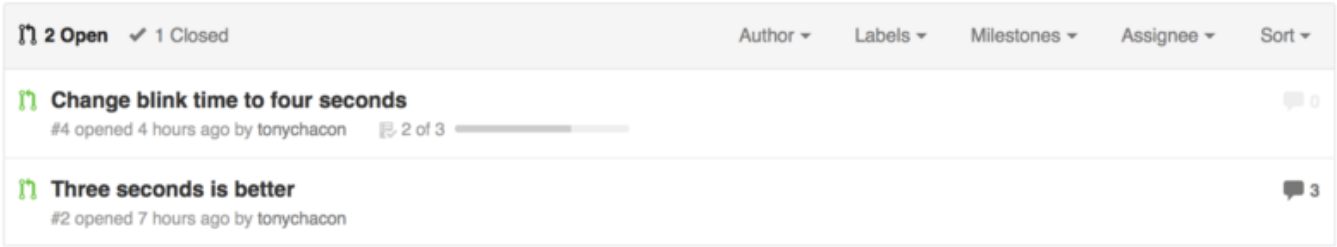



*Figure 103. プルリクエスト一覧における、タスク一覧の概要表示*

この機能は、 トピックブランチを作ったばかりのときにプルリクエストを出して、その後の実装の進捗をプルリクエスト上で追いかけていくような場合に、とても便利です。

**コードスニペット**

コメントに、コードスニペットを追加することもできます。 これは、これから *やろうとしている* ことを、実際に実装する前に表明したりするときに便利です。 また、うまく動かないサンプルコードや、このプルリクエストで実装できることを説明するサンプルコードなどを示すときにも使われます。

コードスニペットを追加するには、バッククォートで「囲む」必要があります。

```
```java
for(int i=0 ; i < 5 ; i++)
{
   System.out.println("i is : " + i);
}
```
```

このサンプルでの *java* のように言語名を追加すると、GitHub はスニペットのシンタックスハイライトを行います。 このサンプルは、最終的に サンプルコードをレンダリングした結果 のような表示になります。


*Figure 104. サンプルコードをレンダリングした結果*

#### 引用

長いコメントの一部に返信するときは、その部分を引用することができます。引用するには、各行の先頭に `>` を付け加えます。 これはとても便利で、よく使われるものなので、キーボードショートカットも用意されています。 コメントの中で返信したい部分を選択して `r` キーを押すと、選択した部分を引用した、新しいコメント入力欄が現れます。

引用は、このような感じになります。

```
> Whether 'tis Nobler in the mind to suffer
> The Slings and Arrows of outrageous Fortune,

How big are these slings and in particular, these arrows?
```

このコメントが、画面上では 引用のレンダリングの例 のようにレンダリングされます。

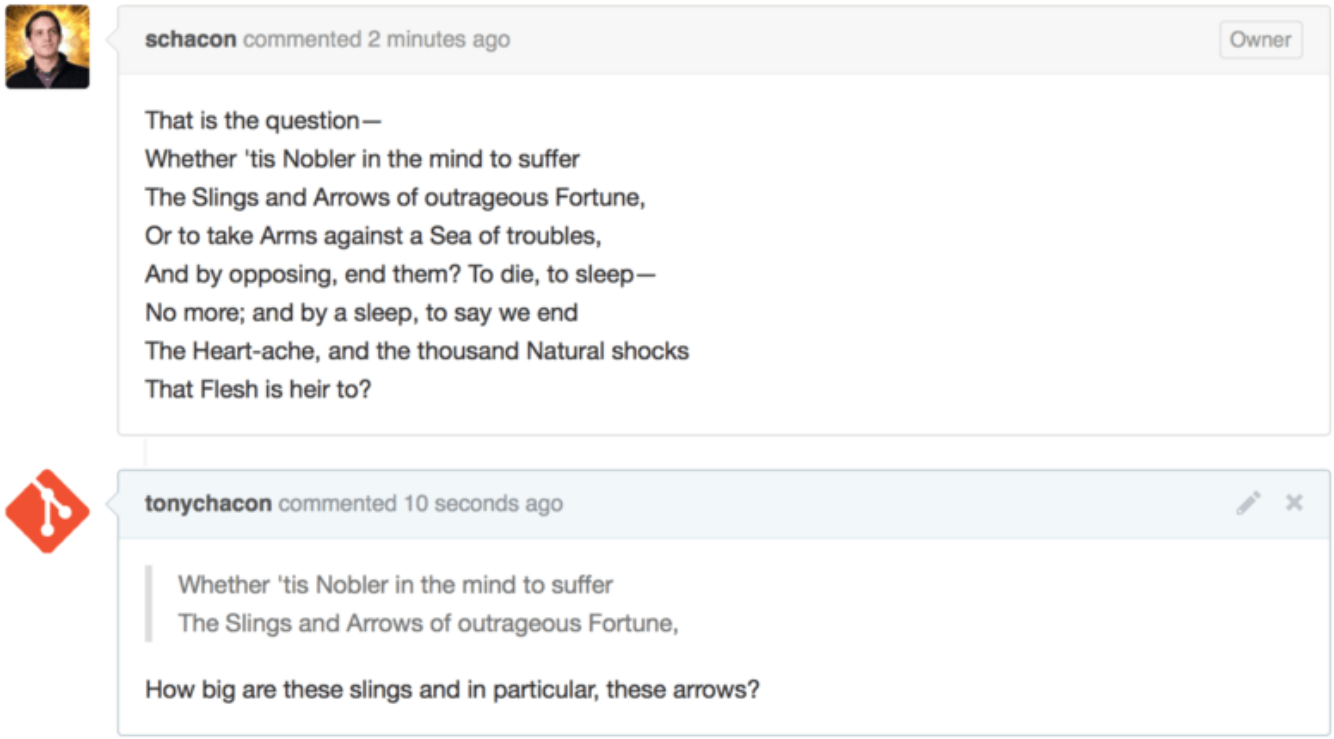



*Figure 105. 引用のレンダリングの例*

#### 絵文字

最後に紹介するのが絵文字です。コメントの中で、絵文字を使えます。 実際に、GitHub の Issue やプルリクエストの多くで、絵文字が使われています。 GitHub には、絵文字の入力支援機能もあるのです。 コメントの記入中に `:` を入力すると、オートコンプリート機能が立ち上がって、絵文字を探すのを手伝ってくれます。

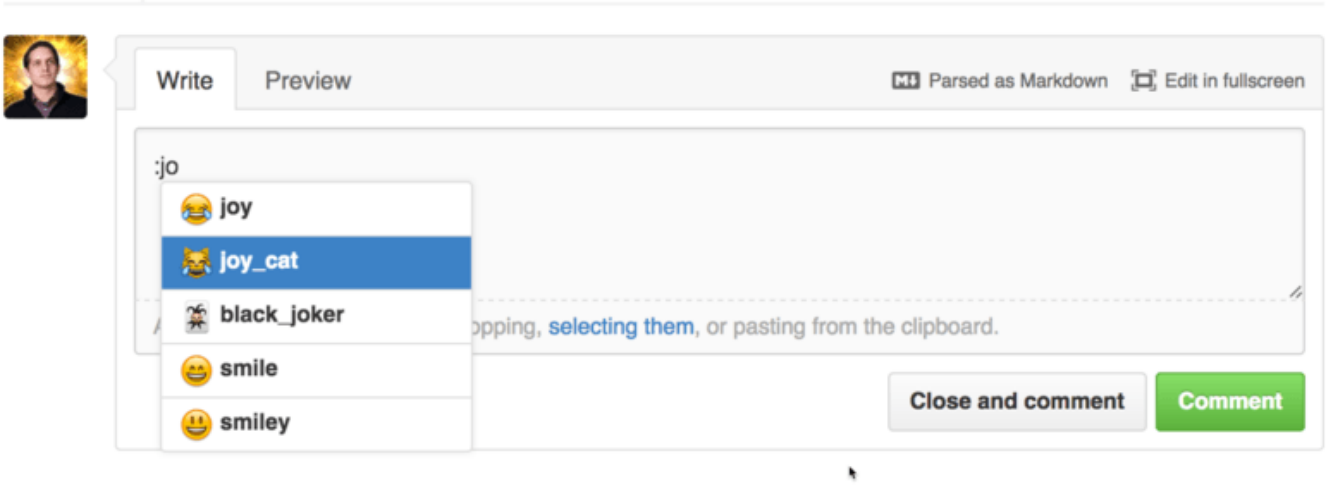



*Figure 106. 絵文字のオートコンプリートの例*

絵文字は `:<name>:` 形式で表し、コメント内のどこでも使えます。 たとえば、このように書いたとしましょう。

```
I :eyes: that :bug: and I :cold_sweat:.

:trophy: for :microscope: it.

:+1: and :sparkles: on this :ship:, it's :fire::poop:!

:clap::tada::panda_face:
```

これをレンダリングした結果は、絵文字だらけのコメント のようになります。

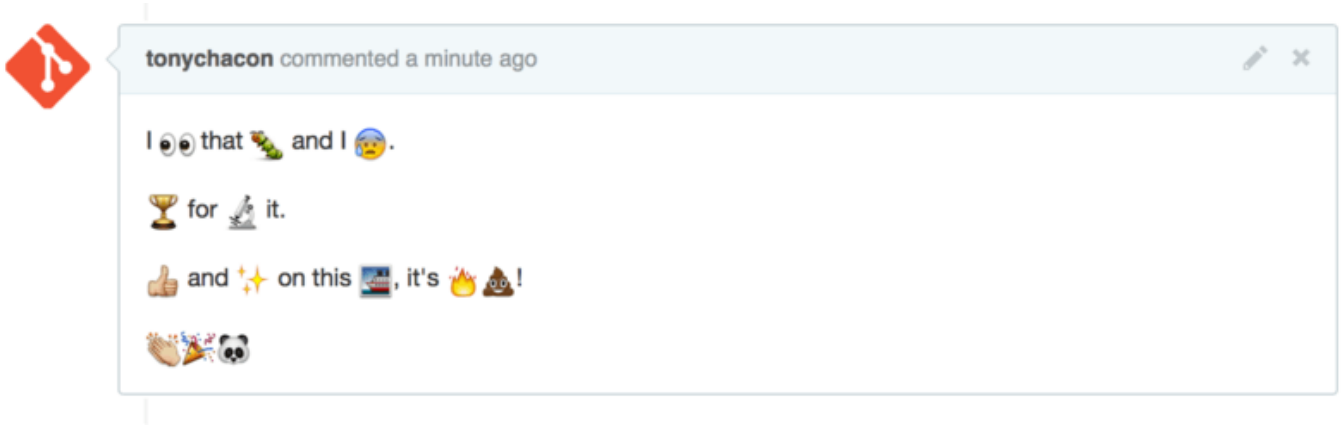



*Figure 107. 絵文字だらけのコメント*

めちゃめちゃ便利というほどのものではありませんが、 楽しさや熱意を伝える手段としては他の追随を許さないものでしょう。

**NOTE** 最近は、絵文字を使えるウェブサービスも多くなってきました。 自分の言いたいことをうまく伝えられる絵文字を見つけるための、チートシートも公開されています。

http://www.emoji-cheat-sheet.com

#### 画像

厳密に言うと GitHub Flavored Markdown とは関係ありませんが、これはとても便利な機能です。 Markdown でのコメントに画像のリンクを追加するのは、画像を探したり URL を埋め込んだりと面倒くさいものです。 しかし GitHub では、テキストエリアに画像をドラッグ＆ドロップするだけで、それを埋め込めるのです。

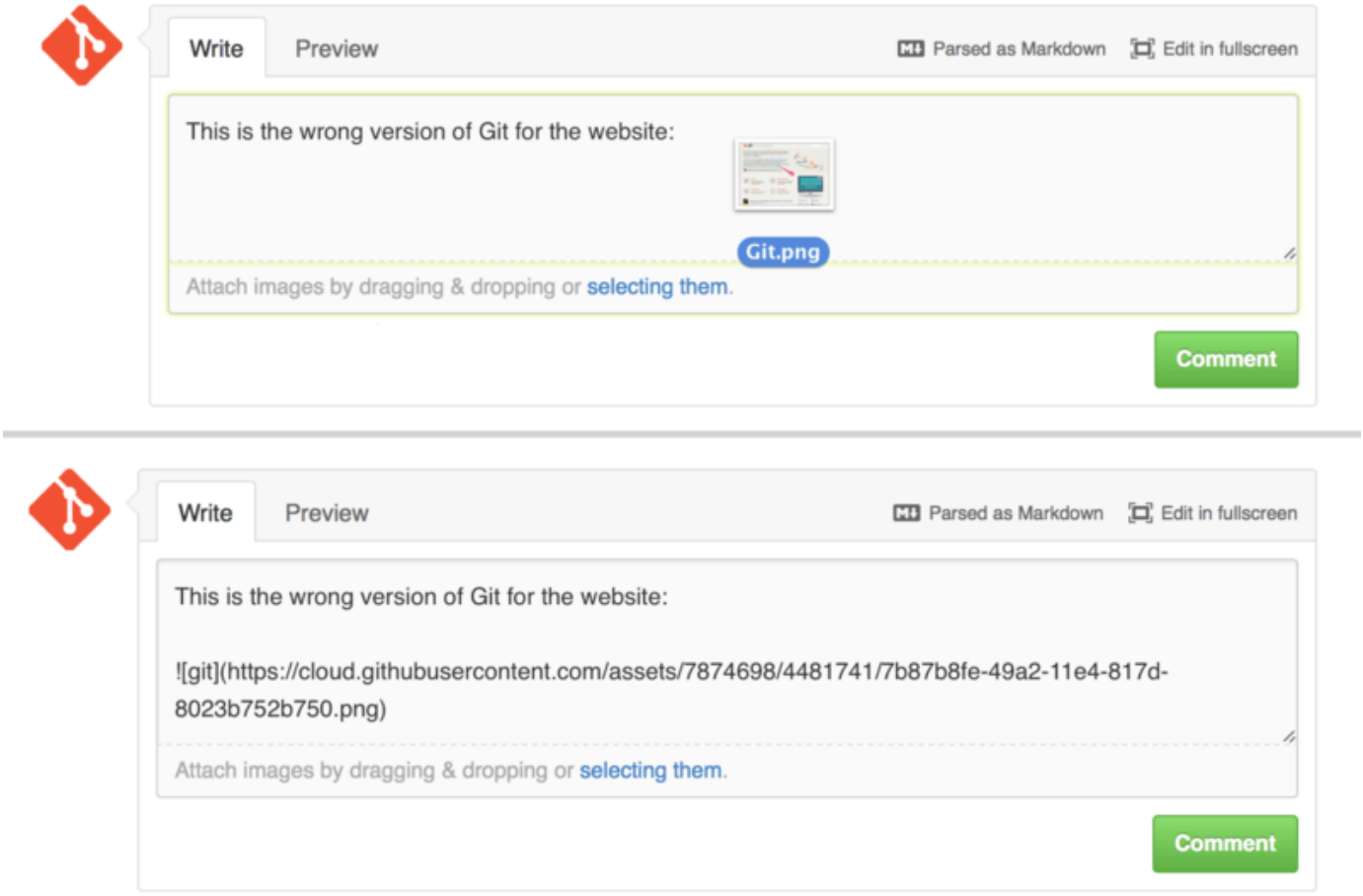



*Figure 108. ドラッグ＆ドロップで画像をアップロードして、自動的に埋め込む*

プルリクエスト内での相互参照 に戻ると、テキストエリアの上に小さく “Parsed as Markdown” とヒントが書かれていることがわかります。 これをクリックすると、GitHub 上での Markdown でできるすべてのことをまとめた、チートシートを見ることができます。

# プロジェクトのメンテナンス

既存のプロジェクトへの貢献のしかたがわかったところで、次はもう一方の側面を見てみましょう。自分自身のプロジェクトを作ったりメンテナンスしたり、管理したりする方法です。

## 新しいリポジトリの作成

新しいプロジェクトを作って、自分たちのプロジェクトのコードを共有しましょう。 まずはダッシュボードの右側にある “New repository” ボタンをクリックするか、 上のツールバーでユーザー名の隣にある + ボタン (“New repository” ドロップダウン を参照) をクリックしましょう。

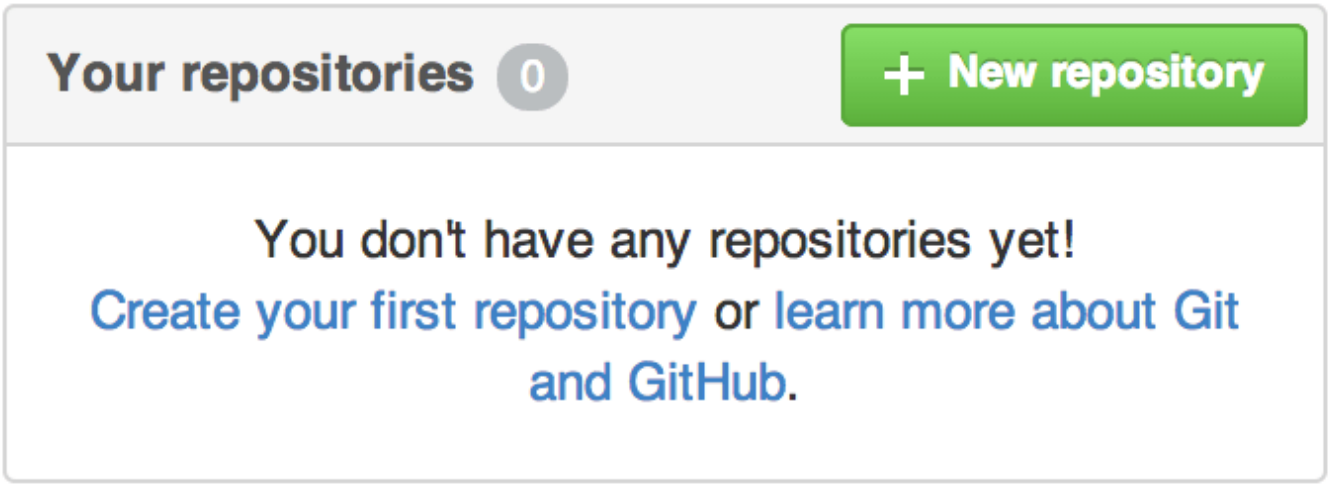



*Figure 109. “Your repositories” エリア*

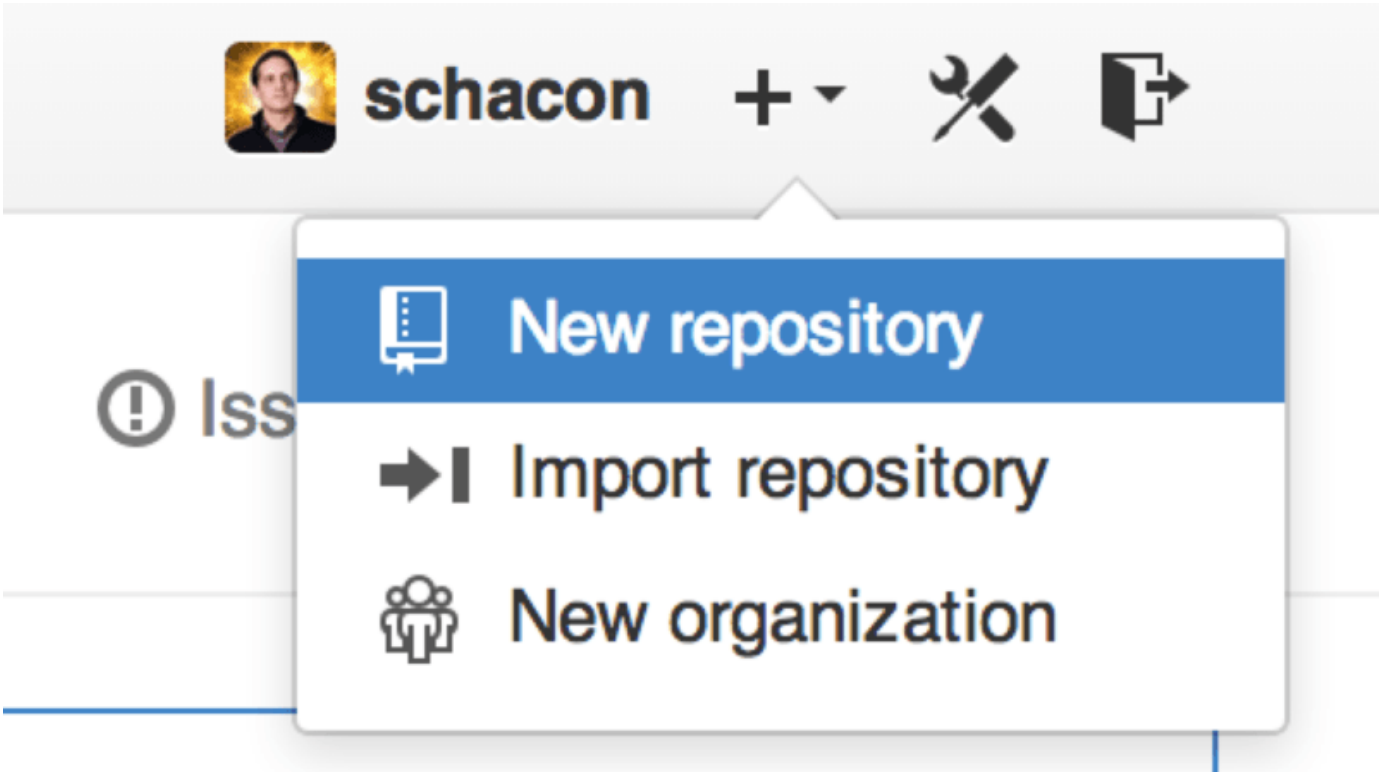



*Figure 110. “New repository” ドロップダウン*

これで、“new repository” フォームが表示されます。

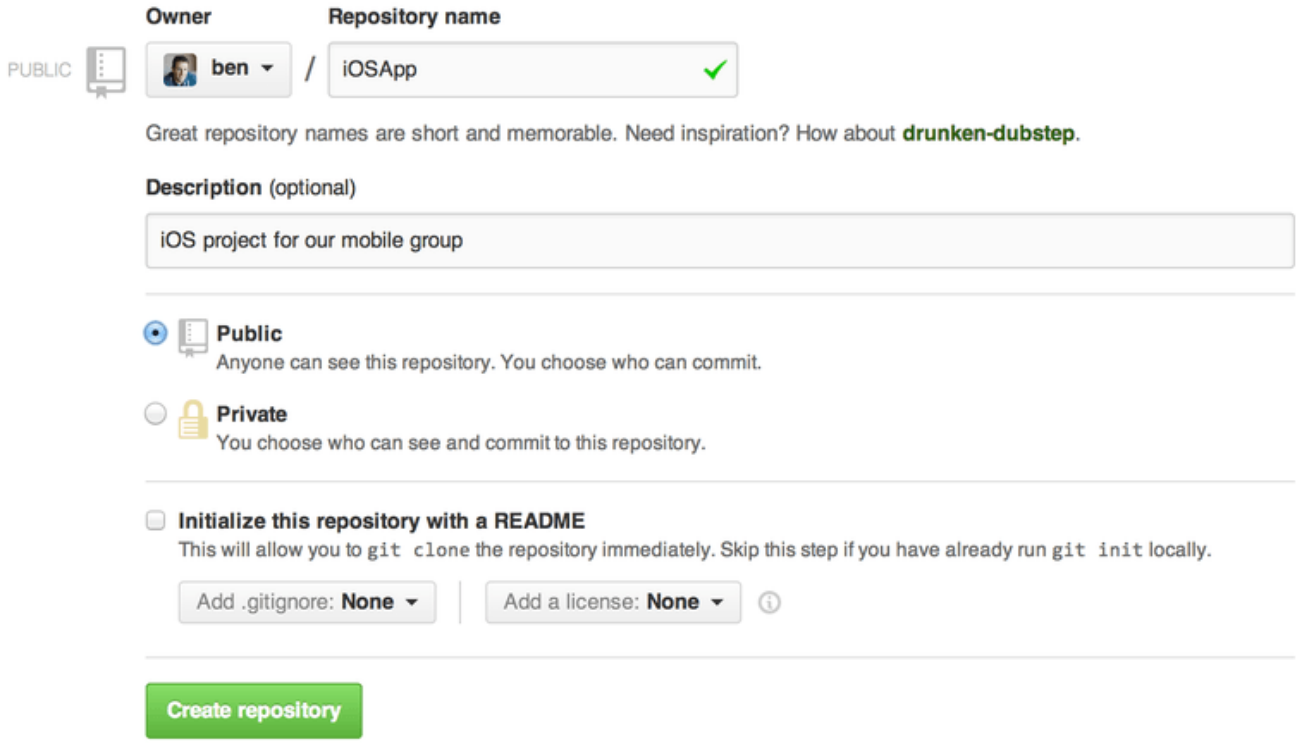



*Figure 111. “new repository” フォーム*

ここで必須なのは、プロジェクト名を入力することだけです。それ以外のフィールドは空のままでもかまいません。 プロジェクト名を入力して “Create Repository” ボタンを押せば、はいできあがり。 これで GitHub 上に、`<user>/<project_name>` という新しいリポジトリができました。

まだ何もコードが存在しないので、GitHub はここで、新しい Git リポジトリを作る方法と既存の Git プロジェクトを取り込む方法を教えてくれます。 ここでは、それらの手順について長々と繰り返したりはしません。 忘れてしまった人は、Git の基本 を見直しましょう。

これで GitHub 上にプロジェクトが用意でき、他の人たちにその URL を示せるようになりました。 GitHub 上のすべてのプロジェクトには、HTTP を使って `https://github.com/<user>/<project_name>` でアクセスすることができます。 また、SSH 経由での `git@github.com:<user>/<project_name>` へのアクセスもできます。 どちらの方式を使ってもデータのフェッチやプッシュができますが、そのプロジェクトに関連付けられたユーザーの認証情報に基づいた、アクセス制御がなされています。

**NOTE** 公開プロジェクトの共有には、HTTP ベースの URL を使うことをお勧めします。 SSH ベースの場合は、プロジェクトをクローンするためには GitHub のアカウントが必要になるからです。 SSH の URL だけを示した場合、それをクローンするには、GitHub のアカウントを作ったうえで SSH 鍵をアップロードする必要があります。 HTTP の URL は、ブラウザでそのプロジェクトのページを表示するときに使うものと同じです。

## コラボレーターの追加

他の人たちにもコミットアクセス権を渡したい場合は、その人たちを “コラボレーター” として追加しなければいけません。 すでに GitHub のアカウントを持っている Ben、Jeff、Louise に、あなたのリポジトリへのプッシュ権限を渡したい場合は、彼らを自分のプロジェクトに追加しましょう。 そうすれば、そのプロジェク

トと Git リポジトリに対して、読み込みだけではなく書き込みアクセスもできるようになります。

右側のサイドバーの一番下にあるリンク “Settings” をクリックしましょう。

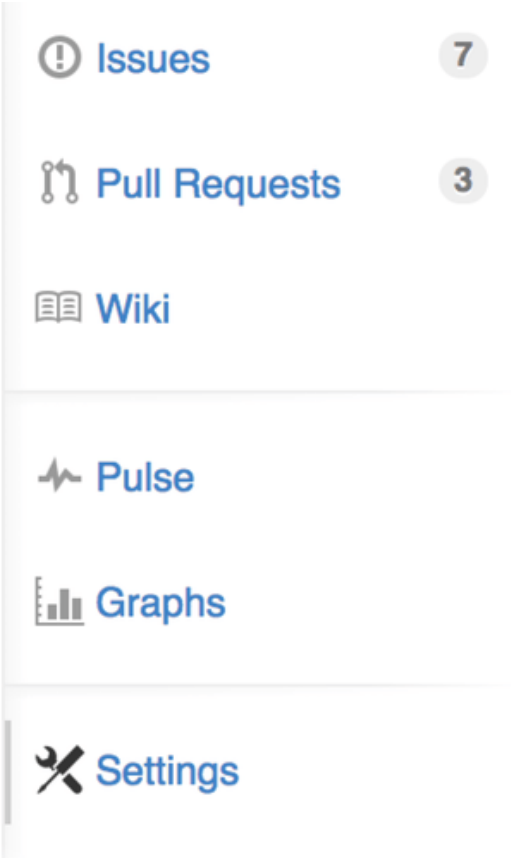



*Figure 112. リポジトリの設定用のリンク*

そして、左側のメニューから “Collaborators” を選びます。 そこで、ユーザー名を入力して “Add collaborator” をクリックしましょう。 これを、アクセス権を追加したいすべての人に対して繰り返します。 アクセス権を破棄したい場合は、そのアカウントの右側にある “X” をクリックします。

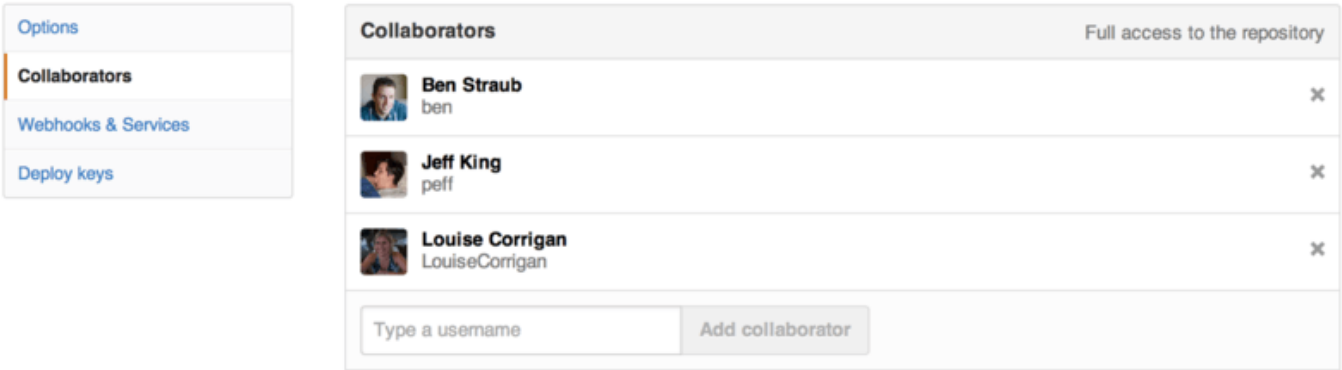



*Figure 113. リポジトリのコラボレーター*

## プルリクエストの管理

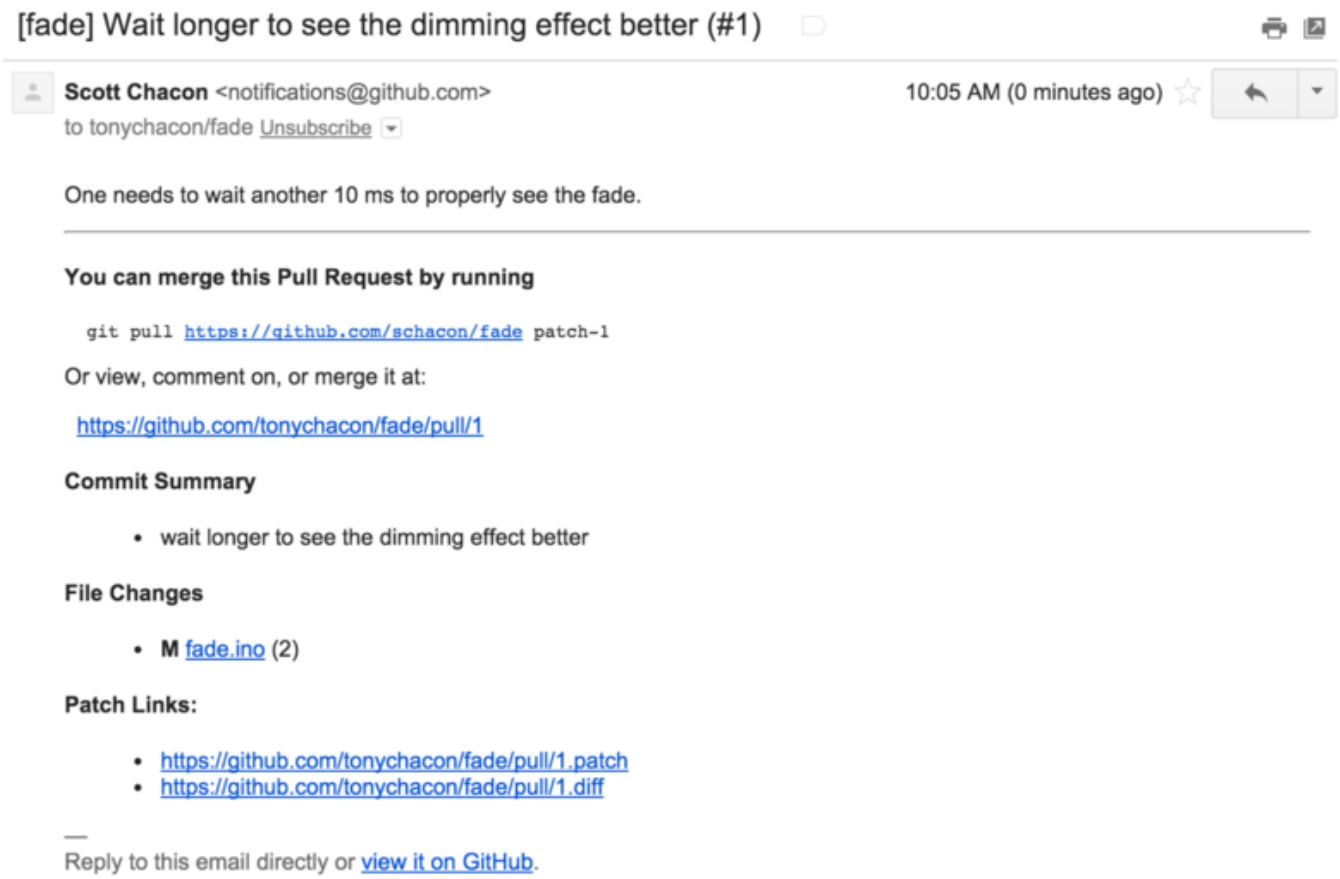
さて、プロジェクトに何らかのコードが追加して、何人かのコラボレーターにプッシュ権限も渡せたかと思います。 ここで、プルリクエストを受け取ったときにやるべきことを紹介しましょう。

プルリクエストは、あなたのリポジトリをフォークした先のブランチからやってくることもあれば、同じリポジトリ内の別ブランチから受け取ることもあります。 フォーク先からやってくるプルリクエストの場合は、あなたはそのリポジトリにプッシュできないし、逆にフォークした側の人もあなたのリポジトリにプッシュできないことが多いでしょう。 一方、同一リポジトリからのプルリクエストの場合は、どちらもお互いに、もう一方のブランチにプッシュできることが多くなります。両者の違いは、ただその一点だけです。

ここでは、あなたが “tonychacon” の立場にいて、Arduino のコードを管理する “fade” プロジェクトを作ったものとしましょう。

**メールでの通知**

あなたのプロジェクトを見つけた誰かが、コードに手を加えてプルリクエストを送ってきました。 このときあなたは、プルリクエストのメールでの通知 のような通知メールを受け取るはずです。


[fade] Wait longer to see the dimming effect better (#1)

**Scott Chacon** <notifications@github.com> 10:05 AM (0 minutes ago)

to tonychacon/fade Unsubscribe

One needs to wait another 10 ms to properly see the fade.

---

**You can merge this Pull Request by running**

```
git pull https://github.com/schacon/fade patch-1
```

Or view, comment on, or merge it at:

https://github.com/tonychacon/fade/pull/1

**Commit Summary**

- wait longer to see the dimming effect better

**File Changes**

- **M** fade.ino (2)

**Patch Links:**

- https://github.com/tonychacon/fade/pull/1.patch
- https://github.com/tonychacon/fade/pull/1.diff

—

Reply to this email directly or view it on GitHub.

*Figure 114. プルリクエストのメールでの通知*

このメールの通知の内容を見てみましょう。 まず差分の簡単な状況（このプルリクエストで変更されたファイルの一覧と、どの程度変更されたのか）がわかります。 また、GitHub 上のプルリクエストのページへのリンクがあります。 さらに、コマンドラインから使えるいくつかの URL も挙げられています。

`git pull <url> patch-1` と書いてある行に注目しましょう。 このようにすれば、リモートを追加しなくても、このブランチをマージできます。 この件については、リモートブランチのチェックアウト で簡単に紹介しました。 もしお望みなら、トピックブランチを作ってそこに移動し、そしてこのコマンドを実行すれば、プルリクエストの変更をマージできます。

さらに、`.diff` と `.patch` の URL も記載されています。
拡張子から想像できるとおり、これらはそれぞれ、このプルリクエストの unified diff とパッチを取得するための URL です。 技術的には、たとえば以下のようにすれば、このプルリクエストをマージできます。

```
$ curl http://github.com/tonychacon/fade/pull/1.patch | git am
```

#### プルリクエスト上での共同作業

GitHub Flow で説明したとおり、プルリクエストの作者とのやりとりができるようになりました。 コードの特定の行にコメントをしたり、コミット全体やプルリクエストそのものに対してコメントしたりすることができ、 その際には GitHub Flavored Markdown が使えます。

プルリクエストに対して誰かがコメントするたびに通知メールが届くので、何らかの動きがあったことを知ることができます。 そのメールには、動きがあったプルリクエストへのリンクが含まれています。そして、通知メールに直接返信すれば、そのプルリクエストのスレッドにコメントをすることができます。

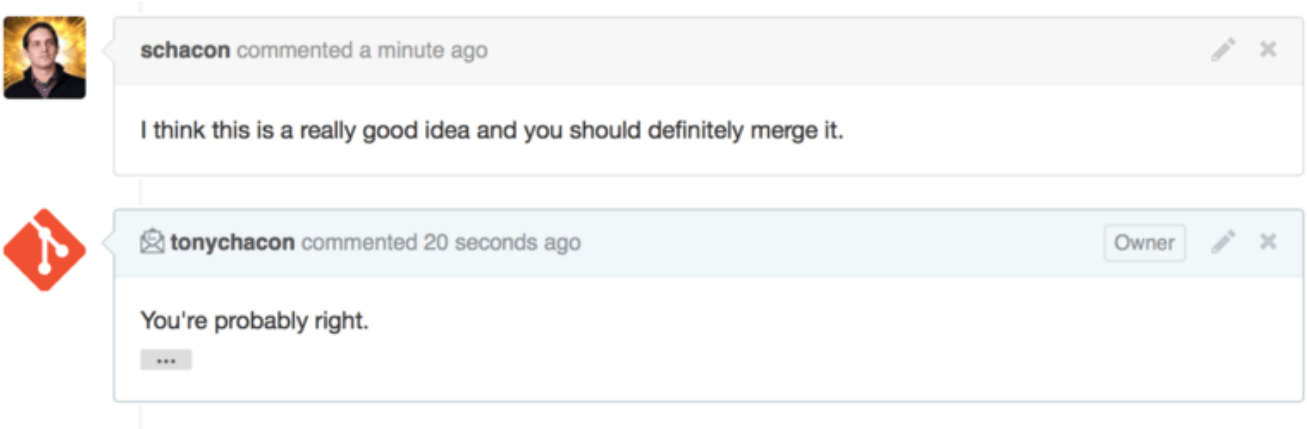



*Figure 115. メールでの返信が、スレッドに含まれる*

コードが望みどおりの状態になり、取り込みたいと思えるようになったら、ローカルにそのコードを取得してマージできます。先述の `git pull <url> <branch>` 構文を使ってもいいし、 そのフォークをリモートとして追加した上で、フェッチしてからマージしてもいいでしょう。

もし特別な作業をせずにマージできる状態なら、GitHub のサイト上で単に “Merge” ボタンを押すだけでマージを済ませることもできます。 このボタンを押すと “non-fast-forward” マージを行います。つまり、fast-forward マージが可能な場合でも、強制的にマージコミットを作ります。 要するに、どんな場合であっても、マージボタンを押したらマージコミットが作られるということです。 マージボタンと、プルリクエストを手動でマージするための手順 にあるとおり、ヒントのリンクをクリックすれば、GitHub がこれらの情報をすべて教えてくれます。

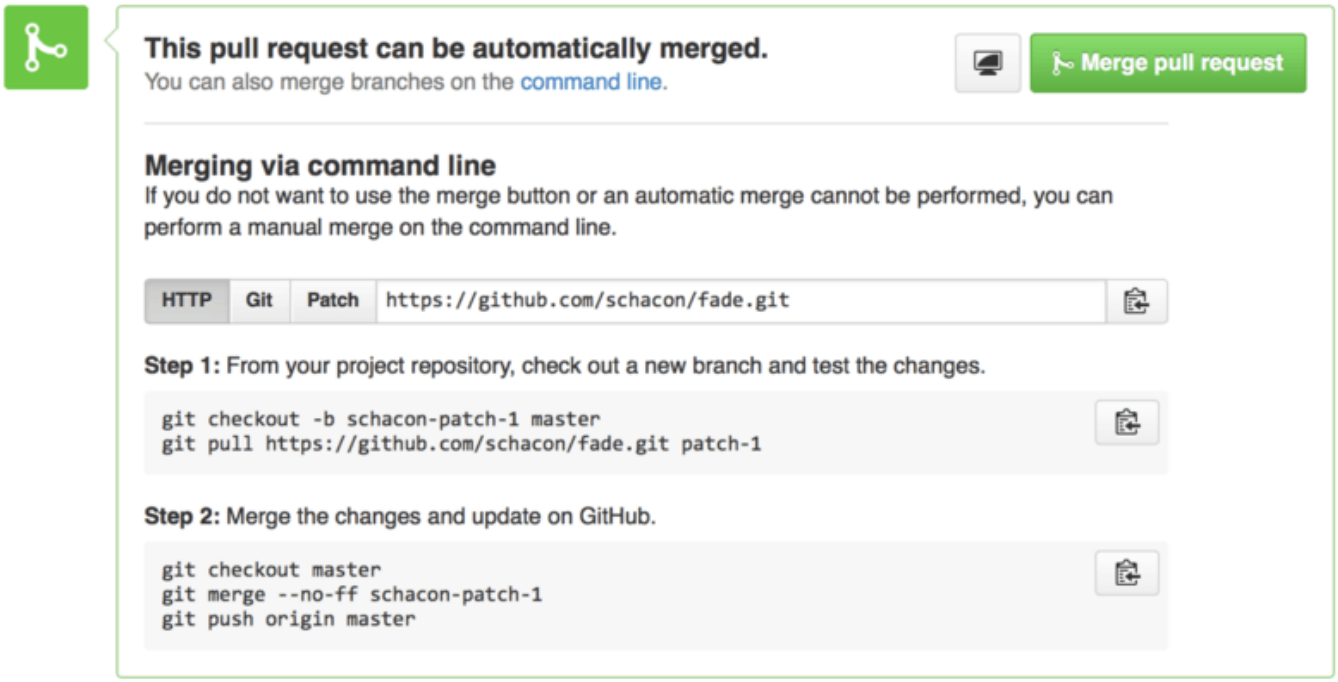



*Figure 116. マージボタンと、プルリクエストを手動でマージするための手順*

マージしたくないと思った場合は、単にそのプルリクエストをクローズするだけでかまいません。プルリクエストの作者には、その旨通知が届きます。

**プルリクエストの参照**

**大量の** プルリクエストを扱っていて、取り込むたびにいちいちリモートを追加するのが面倒な場合は、GitHub が提供するちょっとしたトリックを使えます。 これは高度な話題なので、その詳細は Refspec であらためて取り上げます。ただ、これはとても便利です。

GitHub は、個々のプルリクエストのブランチを、サーバー上で擬似ブランチとして公開しています。 クローンするときに、デフォルトでは取り込まれませんが、目立たないところに存在していて、簡単にアクセスできます。

その様子を示すために、ここでは、下位レベルのコマンド (「配管」コマンド) である `ls-remote` を使います。 このコマンドを日々の Git の操作で使うことはあまりありませんが、サーバー上に何があるのかを見るためには便利です。

先ほどの “blink” リポジトリに対してこのコマンドを実行すると、すべてのブランチやタグ、そしてその他の参照の一覧を取得できます。

```
$ git ls-remote https://github.com/schacon/blink
10d539600d86723087810ec636870a504f4fee4d    HEAD
10d539600d86723087810ec636870a504f4fee4d    refs/heads/master
6a83107c62950be9453aac297bb0193fd743cd6e    refs/pull/1/head
afe83c2d1a70674c9505cc1d8b7d380d5e076ed3    refs/pull/1/merge
3c8d735ee16296c242be7a9742ebfbc2665adec1    refs/pull/2/head
15c9f4f80973a2758462ab2066b6ad9fe8dcf03d    refs/pull/2/merge
a5a7751a33b7e86c5e9bb07b26001bb17d775d1a    refs/pull/4/head
31a45fc257e8433c8d8804e3e848cf61c9d3166c    refs/pull/4/merge
```

もちろん、自分のリポジトリにいるときに `git ls-remote origin` のようにリモートを指定すると、これと同じような結果が得られるでしょう。

GitHub 上にあるリポジトリで、オープン中のプルリクエストがある場合は、 プルリクエストへの参照も表示されます。これらの参照は、先頭が `refs/pull/` となります。 基本的にはブランチですが、`refs/heads/` の配下にあるわけではないので、通常のクローンやフェッチで取得することはできません。 フェッチの際には通常、これらのブランチを無視します。

ひとつのプルリクエストにつき、二つの参照が表示されています。 一方は `/head` で終わるもので、これは、そのプルリクエストのブランチの最新のコミットを指しています。 誰かが私たちのリポジトリにプルリクエストを送ってきたとして、仮にそのブランチ名が `bug-fix` で参照先のコミットが `a5a775` だったとしましょう。 **私たちの** リポジトリには `bug-fix` ブランチがありません (彼らのフォーク上にしかありません) が、`pull/<pr#>/head` が `a5a775` を指すようになるのです。 つまり、大量にリモートを追加したりしなくても、あらゆるプルリクエストのブランチをコマンドひとつで手元に取り込めるのです。

この参照を直接指定して、以下のようにフェッチすることができます。

```
$ git fetch origin refs/pull/958/head
From https://github.com/libgit2/libgit2
 * branch            refs/pull/958/head -> FETCH_HEAD
```

このコマンドは Git に対して、「リモート `origin` に接続して、`refs/pull/958/head` をダウンロードしなさい」という指示を出します。 Git はその指示に従い、必要なものをすべてダウンロードして、あなたが必要とするコミットへのポインタを `.git/FETCH_HEAD` に置きます。 これを `git merge FETCH_HEAD` で自分のブランチに取り込んで試すこともできますが、マージコミットのメッセージは少しわかりにくくなります。また、**大量の** プルリクエストを処理するときには、この作業は退屈でしょう。

すべての プルリクエストを取得して、リモートに接続するたびに最新の状態を保つようにする方法もあります。 `.git/config` をお好みのエディタで開いて、リモート `origin` の記載を探しましょう。 きっと、このようになっているはずです。

```
[remote "origin"]
    url = https://github.com/libgit2/libgit2
    fetch = +refs/heads/*:refs/remotes/origin/*
```

`fetch =` で始まっている行が、“refspec” です。 ここで、リモートでの名前とローカルの `.git` ディレクトリ内での名前のマッピングができます。 この例では、Git に対して「リモートの `refs/heads` 配下にあるものを、ローカルのリポジトリ内では `refs/remotes/origin` 配下に置くこと」と指示しています。 このセクションを書き換えて、別の refspec を追加できます。

```
[remote "origin"]
    url = https://github.com/libgit2/libgit2.git
    fetch = +refs/heads/*:refs/remotes/origin/*
    fetch = +refs/pull/*/head:refs/remotes/origin/pr/*
```

最後のに追加した行は、「`refs/pull/123/head` のような参照はすべて、ローカルでは `refs/remotes/origin/pr/123` のように保存すること」という意味です。 さて、このファイルを保存したら、`git fetch` を実行してみましょう。

```
$ git fetch
# …
 * [new ref]         refs/pull/1/head -> origin/pr/1
 * [new ref]         refs/pull/2/head -> origin/pr/2
 * [new ref]         refs/pull/4/head -> origin/pr/4
# …
```

リモートのすべてのプルリクエストが、ローカルでも、まるで追跡ブランチであるかのように表されるようになりました。 これらのブランチは読み込み専用で、フェッチするたびに更新されます。 これで、プルリクエストのコードをローカルで簡単に試せるようになりました。

```
$ git checkout pr/2
Checking out files: 100% (3769/3769), done.
Branch pr/2 set up to track remote branch pr/2 from origin.
Switched to a new branch 'pr/2'
```

リモート側の refspec の最後に `head` と表示されていることに、目ざとい人なら気づいたかもしれません。 GitHub 上には、これだけではなく `refs/pull/#/merge` という参照もあります。 これは、サイト上で「マージ」ボタンを押したときに作られるコミットを指す参照です。 これを使えば、マージしたらどうなるかを、ボタンを押す前に確かめることができるのです。

#### プルリクエスト上でのプルリクエスト

別に、プルリクエストの対象がメインブランチ (`master` ブランチ) でなければいけないなどという決まりはありません。 ネットワーク上にあるあらゆるブランチに対して、プルリクエストを作ることができます。 別のプルリクエストに対して、プルリクエストを送ることだってできるのです。

正しい方向に進みつつあるプルリクエストに対して、それを元にした新たな変更のアイデアが浮かんだ場合や、 単にそのプルリクエストの対象ブランチへのプッシュ権限がない場合などに、 プルリクエストに対するプルリクエストを作ることができます。

プルリクエストを作る際に、ページの上のほうに二つの入力欄があることがわかります。 それぞれ、どのブランチに対するリクエストなのかと、どのブランチからプルしてほしいのかを指定する欄です。 この欄の右側にある「編集」ボタンを押すと、ブランチ名だけではなく、どのフォークを使うのかも変更できます。

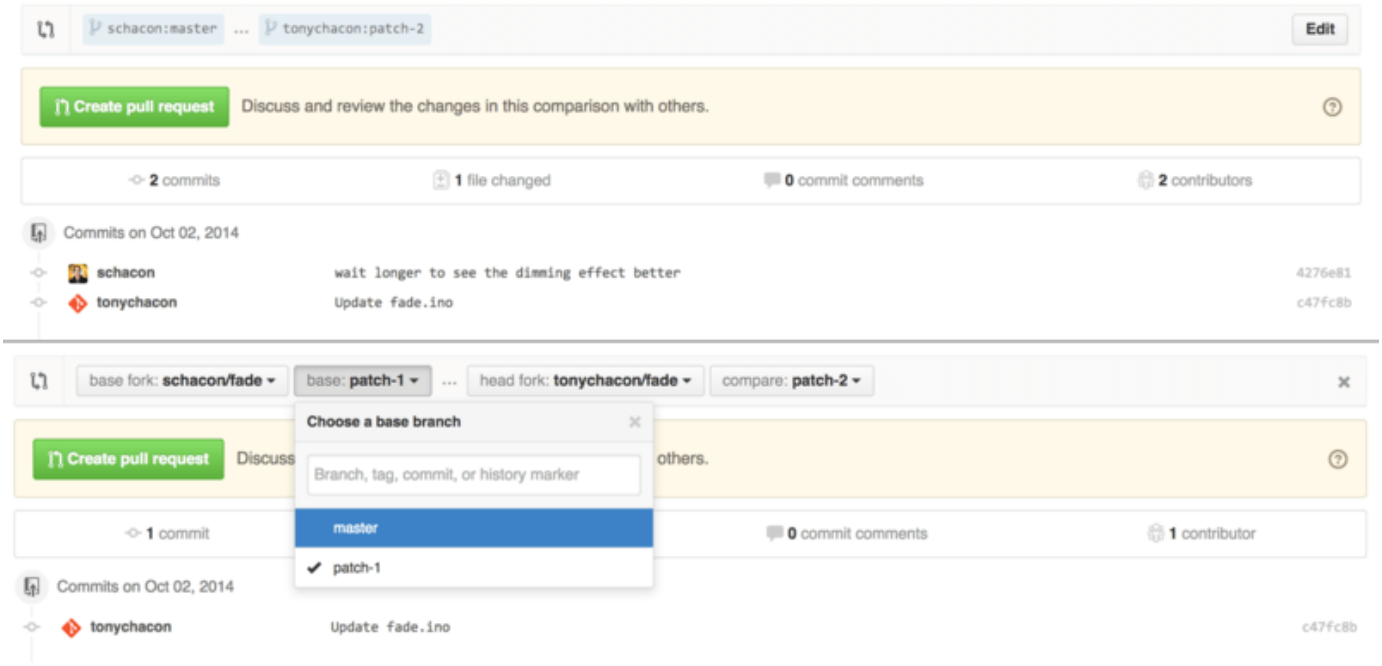



*Figure 117. プルリクエストの対象となるフォークとブランチを手動で変更する*

これを使えば、あなたのブランチを別のプルリクエストにマージするよう指定したり、そのプロジェクトの別のフォークへのマージ依頼を出したりするのも簡単です。

## 言及と通知

GitHub には、よくできた通知システムも組み込まれています。特定の人やチームに質問をしたり、何かのフィードバックが必要だったりする場合に便利です。

コメントの記入時に @ を入力すると、自動補完が始まります。 そのプロジェクトの Collaborator や、これまでの貢献者たちの、名前やユーザー名を補完できます。

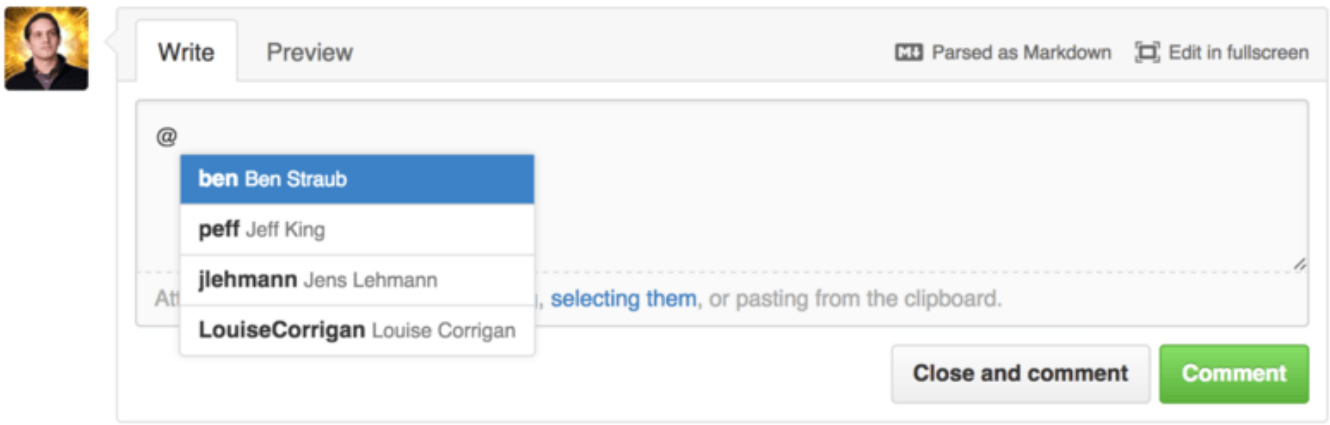



*Figure 118. 誰かについて言及するには、@ を入力する*

このドロップダウンに登場しないユーザーについても言及できますが、
通常は、この自動補完を使ったほうがずっとお手軽でしょう。

コメントの中でユーザーについて言及すると、そのユーザーに通知が届きます。 他の人を議論に巻き込みたいときに、これをうまく活用できるでしょう。 GitHub 上のプルリクエストでは、チームや社内の他のメンバーを巻き込んだレビューが行われることも、珍しくありません。

プルリクエストや Issue の中で言及された人は、自動的にそれを「購読した」状態になり、 何らかのアクションがあるたびに通知が届くことになります。 また、自分がウォッチしていたり、何かのコメントをしたりしたことがあるリポジトリに対してプルリクエストや Issue を作った場合も、 あなたはそれを自動的に「購読した」ことになります。 その通知を受け取りたくなくなった場合は、ページ上にある “Unsubscribe” ボタンをクリックすると、更新の通知が届かないようになります。

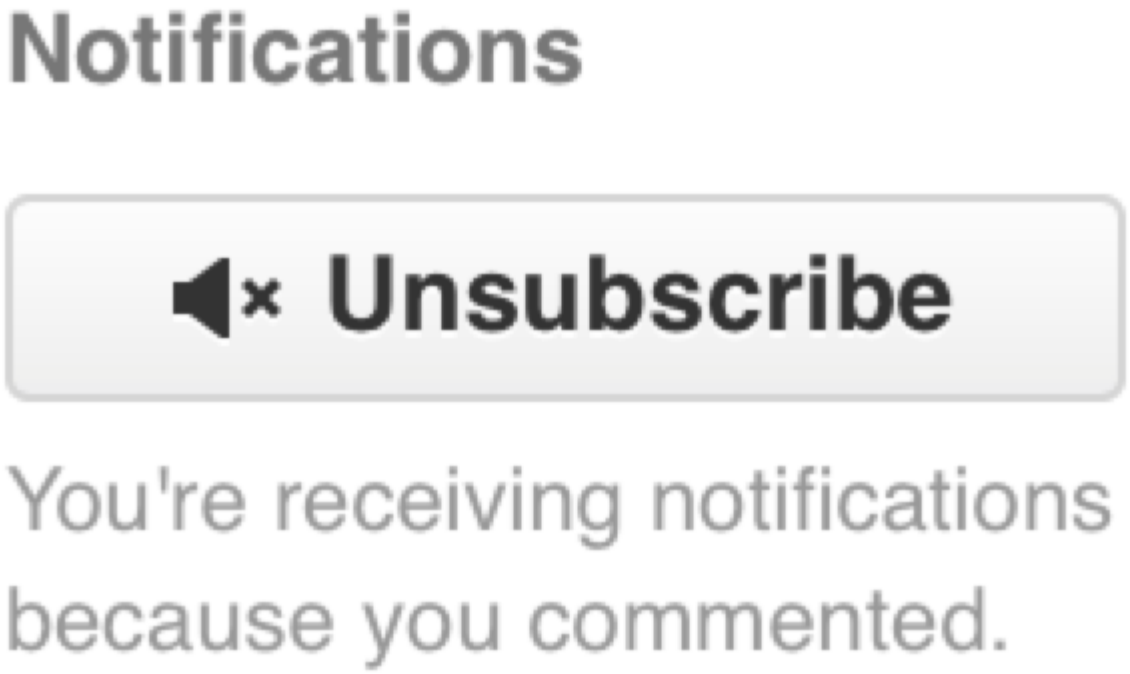



*Figure 119. Issue やプルリクエストの購読の解除*

**通知ページ**

GitHub に関する話題で「通知」と言ったときには、それは、 何かの出来事が起こったときに GitHub が私たちにそれを伝える手段のことを指します。 どのように通知を受け取るのかについては、いくつか設定できる項目があります。 設定ページの “Notification center” タブに移動すると、設定可能な選択肢を確認できるでしょう。

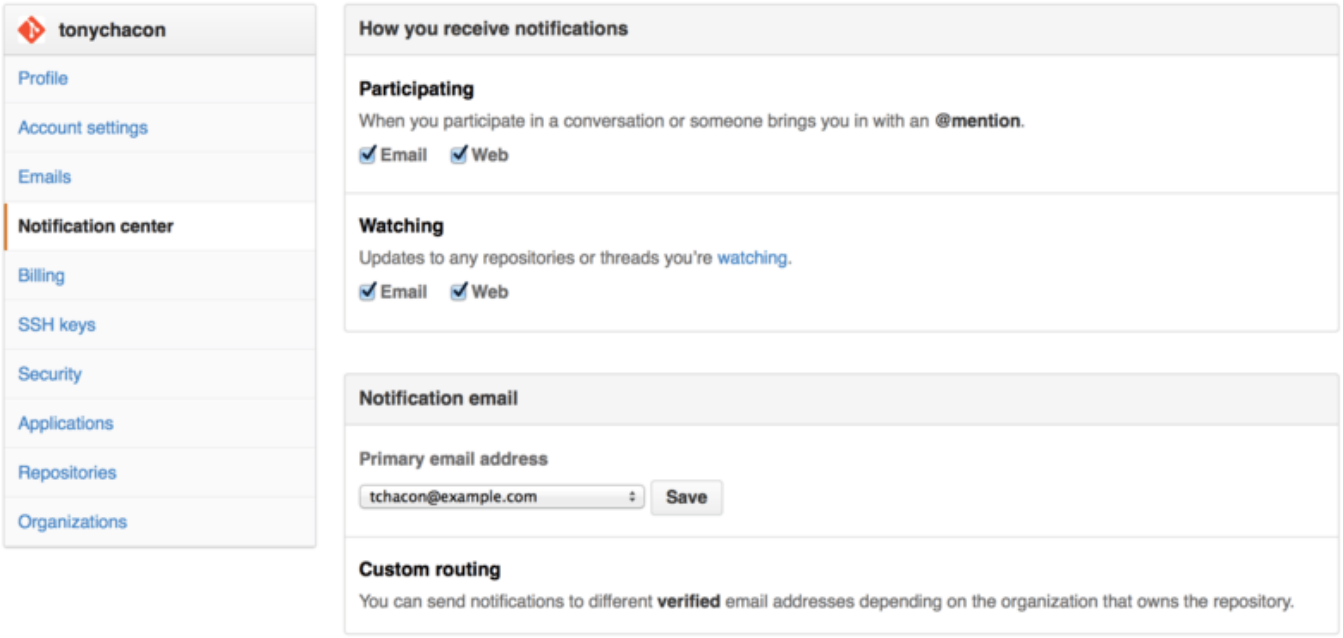



*Figure 120. 通知センターのオプション*

通知の受け取りかたを、「メールで受け取る」のか「Webで受け取る」のか (あるいはその両方で受け取るのか、どちらでも受け取らないのか) を、 自分がかかわっているものについてと自分がウォッチしているリポジトリについてとで、それぞれ選べます。

**Web での通知**

Web での通知は GitHub 上でだけ行われるもので、GitHub のサイトに行かないと確認できません。 このオプションを選んだ場合、あなたに届いた通知は、画面上部の通知アイコンに青い点として表示されて、 通知センター のようになります。

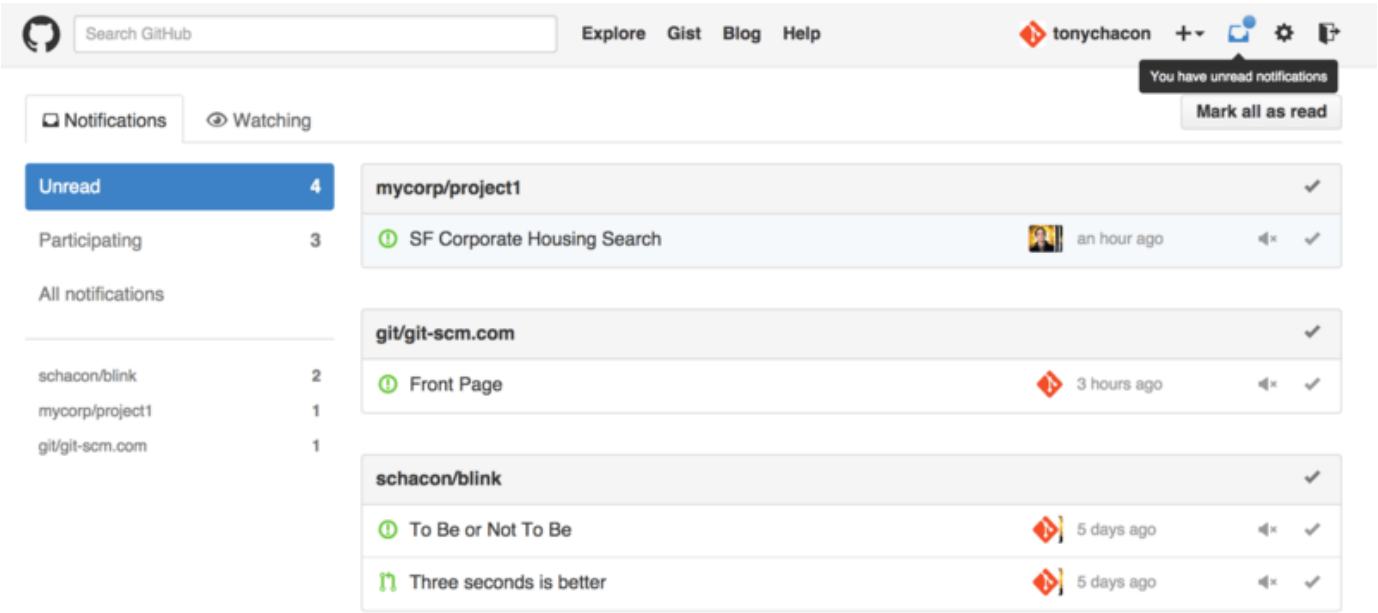



*Figure 121. 通知センター*

これをクリックすると、通知の一覧が、プロジェクトごとにまとまった形式で表示されます。 特定のプロジェクトの通知だけに絞り込むには、左側のサイドバーにあるプロジェクト名をクリックしましょう。 通知の受け取り確認をするには、個々の通知の隣にあるチェックマークをクリックします。 または、プロジェクトごとのグループのプロジェクト名のところにあるチェックマークをクリックすると、そのプロジェクトの すべての 通知を確認済みにできます。 チェックマークの隣にあるのがミュートボタンで、これをクリックすると、その件に関する通知が今後届かなくなります。

これらをうまく活用すれば、通知が大量に届いても、うまくさばくことができます。 GitHub のパワーユーザーの多くは、メールでの通知を完全にオフにしてしまって、通知はすべてこの画面だけで管理しているようです。

#### メールでの通知

メールでの通知を使って、GitHub からの通知を処理することもできます。 この機能を有効にしておくと、さまざまな通知をメールで受け取れるようになります。 その例を 通知メールで送られたコメント と プルリクエストのメールでの通知 に示します。 メールのスレッド機能にも対応しているので、スレッド対応のメールソフトを使えば適切に表示できることでしょう。

GitHub が送るメールのヘッダーには、さまざまなメタデータが埋め込まれています。 これらを使えば、フィルタリングやフォルダ分けの設定も簡単に行えます。

プルリクエストのメールでの通知 に示す、Tony に送られたメールのヘッダーには、このような情報が含まれています。

```
To: tonychacon/fade <fade@noreply.github.com>
Message-ID: <tonychacon/fade/pull/1@github.com>
Subject: [fade] Wait longer to see the dimming effect better (#1)
X-GitHub-Recipient: tonychacon
List-ID: tonychacon/fade <fade.tonychacon.github.com>
List-Archive: https://github.com/tonychacon/fade
List-Post: <mailto:reply+i-4XXX@reply.github.com>
List-Unsubscribe: <mailto:unsub+i-XXX@reply.github.com>,...
X-GitHub-Recipient-Address: tchacon@example.com
```

いろいろ興味深い内容が含まれていることがわかるでしょう。 特定のプロジェクト、あるいは特定のプルリクエストに関するメールを強調したり転送したりしたければ、 `Message-ID` を利用できます。これは `<user>/<project>/<type>/<id>` 形式になっています。 もしこれば issue に関する通知なら、`<type>` の部分が “pull” ではなく “issues” になります。

`List-Post` や `List-Unsubscribe` フィールドを解釈できるメールソフトを使っている場合は、 そのスレッドへの投稿やスレッドからの「脱退」(通知を受け取らないようにすること) を簡単に行えます。 スレッドからの脱退とは、Web の通知画面でミュートボタンを押したり、Issue やプルリクエストのページで “Unsubscribe” をクリックしたりするのと同じことです。

メールと Web の両方で通知を受け取っている場合は、メールでの通知を読んだ時点で、Web 版の通知も既読になります。 ただし、お使いのメールソフトでメール本文中の画像の表示を許可している場合に限ります。

## 特別なファイル

以下の名前のファイルがリポジトリ内にあった場合、GitHub はそれを特別扱いします。

## README

特別扱いする最初のファイルは `README` です。ほとんどのファイル形式について、GitHub 自身がそのフォーマットを解釈します。 たとえば `README`、`README.md`、`README.asciidoc` などが使えます。 README ファイルを発見すると、GitHub はそれをレンダリングして、プロジェクトのトップページに表示します。

多くのチームは、このファイルを使って、プロジェクトに関する情報をまとめています。 そのリポジトリやプロジェクトに初めて参加する人たち向けの情報を含めているのです。たとえば以下のような内容です。

- そのプロジェクトの目的
- インストール手順
- 利用例や、動作させるための手順
- そのプロジェクトのライセンス情報
- プロジェクトに参加する方法

GitHub がこのファイルをレンダリングしてくれるので、画像やリンクを追加したりして、わかりやすい説明を書くことができます。

## CONTRIBUTING

GitHub は、`CONTRIBUTING` も特別扱いするファイルとして認識します。 `CONTRIBUTING` という名前 (拡張子は何でもかまいません) のファイルを用意すると、 誰かがプルリクエストを作ろうとしたときに、GitHub がその内容を CONTRIBUTING ファイルが存在するプロジェクトへのプルリクエスト のように表示します。

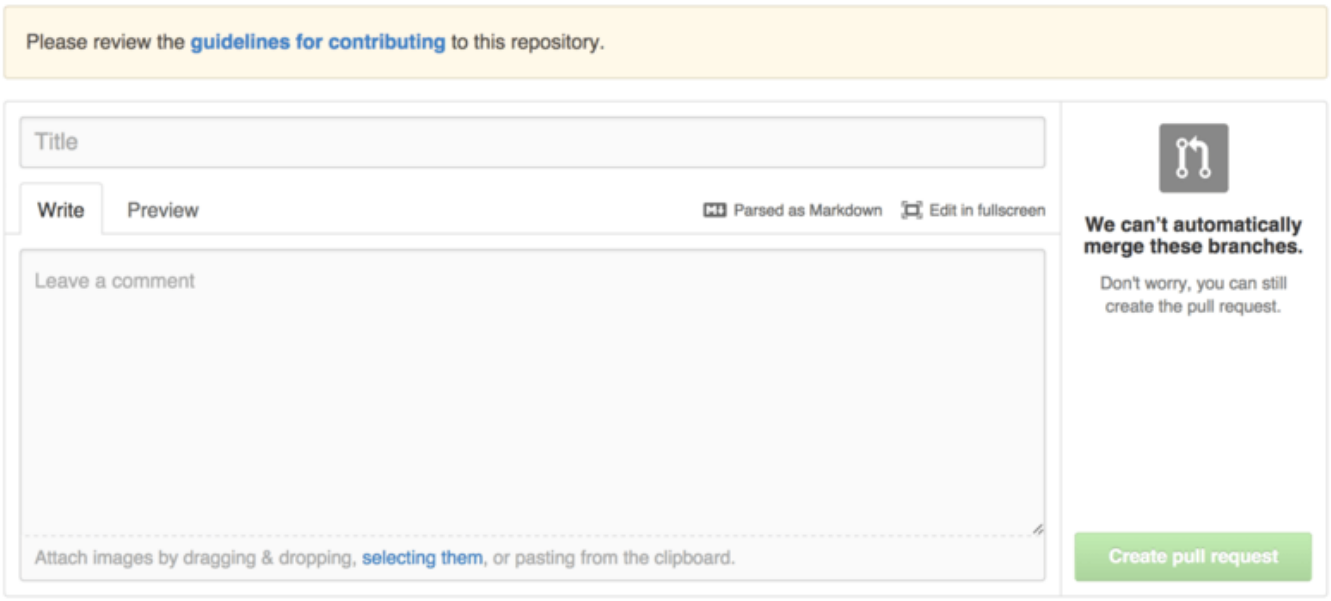



*Figure 122. CONTRIBUTING ファイルが存在するプロジェクトへのプルリクエスト*

このファイルには、プロジェクトへのプルリクエストを送る際に気をつけてほしいこと (あるいは、してほし

くないこと) などを書いておくといいでしょう。
プルリクエストを作ろうとした人は、このガイドラインを見ることになります。

## プロジェクトの管理

実際のところ、単独のプロジェクトについての管理操作は、そんなに多くはありません。 しかし、中には皆さんの興味をひくものもあることでしょう。

### デフォルトブランチの変更

“master” 以外のブランチをデフォルトにして、他の人たちからのプルリクエストのデフォルトの送り先をそこにすることができます。 デフォルトブランチを変更するには、“Options” タブの中にある設定ページを使います。

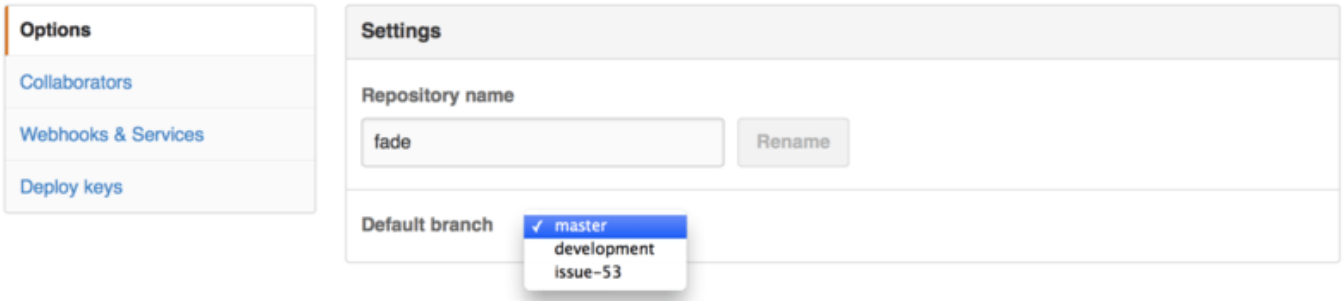



*Figure 123. プロジェクトのデフォルトブランチの変更*

ドロップダウンでブランチを変更すれば、それが主要な操作のデフォルトの対象となります。 誰かがそのリポジトリをクローンしたときに、デフォルトでチェックアウトされるのも、このブランチです。

### プロジェクトの移管

GitHub 上で、別のユーザーや組織にプロジェクトを移管したい場合に使えるのが、 同じくリポジトリの設定ページの “Options” タブの一番下にある “Transfer ownership” 欄です。

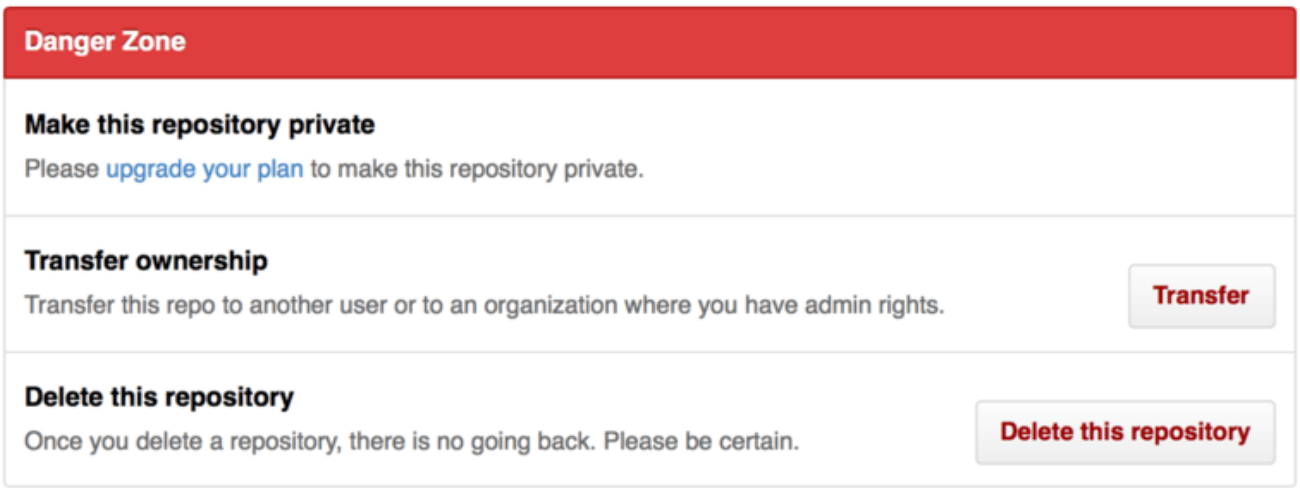



*Figure 124. 別の GitHub ユーザーや組織への、プロジェクトの移管*

自分のリポジトリを手放して他の誰かに運営してもらう場合や、プロジェクトが成長したこともあって個人管理から組織での管理に移行したい場合などに使えます。

これは、リポジトリそのものだけではなく、そのリポジトリをウォッチしたり、スターを付けたりしている人の情報も含めて移行します。 さらに、移管前の URL から新しい URL へのリダイレクトの設定も行われます。もちろん、Web のリクエストに限らず、Git のクローンやフェッチのリクエストもリダイレクトされます。

# 組織の管理

GitHub には、個人ユーザー用のアカウント以外にも、組織 (Organization) 用アカウントが用意されています。 個人アカウントと同様に組織アカウントでも、その名前空間にプロジェクトを持つことができます。 しかし、それ以外の点では異なるところが多数あります。 組織アカウントは、複数の人たちによるプロジェクトの共同所有を表すもので、 さらにその内部でのグループ管理をするための、さまざまなツールが用意されています。 組織アカウントは一般に、オープンソースのグループ (“perl” や “rails” など) や、一般企業 (“google” や “twitter” など) が使うものです。

## 組織についての基本

組織アカウントの作成はきわめて簡単です。GitHub 上のすべてのページの右上にある “+” アイコンをクリックして、メニューから “New organization” を選びましょう。

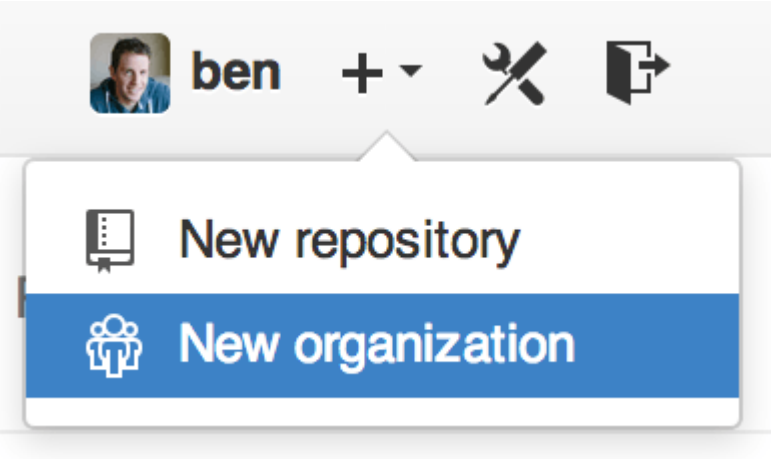



*Figure 125. “New organization” メニュー*

まず必要になるのが、組織名と、そのグループの連絡先となるメールアドレスです。 もし望むなら、他のユーザーを、共同オーナーとしてこのアカウントに招待することもできます。

この手順に従えば、新しい組織のオーナーになれます。 個人アカウントと同様、組織アカウントも、すべてのプロジェクトをオープンソースにするのであれば無料で使えます。

組織のオーナーであるあなたが何かのプロジェクトをフォークするときには、個人の名前空間にフォークするのか組織の名前空間にフォークするのかを選べるようになります。 新しいプロジェクトを作るときにも同様に、個人アカウントの配下に作るのか組織の配下に作るのかを選べます。 また、組織の配下に作ったリポジトリは、自動的に、個人アカウントからの “ウォッチ” の対象になります。

アバター と同様に、組織アカウントにもアバターを設定できるようになっています。 さらに、個人アカウン

トと同様のランディングページも用意されています。
その組織アカウントが抱えるリポジトリの一覧を、他の人にも見てもらえることでしょう。

さて、ここから先は、個人アカウントとは異なる組織アカウント独特の内容について、説明しましょう。

## チーム

組織アカウントの中では、個々のメンバーをチームとして関連付けることができます。 これは単に、個人ユーザーアカウントと組織内のリポジトリをとりまとめたものであり、 そのリポジトリに対するアクセス権の設定などを行います。

たとえば、あなたの所属する企業の組織アカウントに `frontend`、`backend`、`deployscripts` の三つのリポジトリがあるものとします。 HTML/CSS/Javascript の開発者たちには、`frontend` と、おそらくは `backend` についてもアクセスさせたいことでしょう。 一方、運用部門の人たちには、`backend` や `deployscripts` にアクセスできるようにしておきたいところです。 チーム機能を使えば、簡単に実現できます。リポジトリごとに Collaborators を管理する必要はありません。

組織アカウントにはシンプルなダッシュボードがあり、すべてのリポジトリやユーザーそしてチームの情報を確認できます。

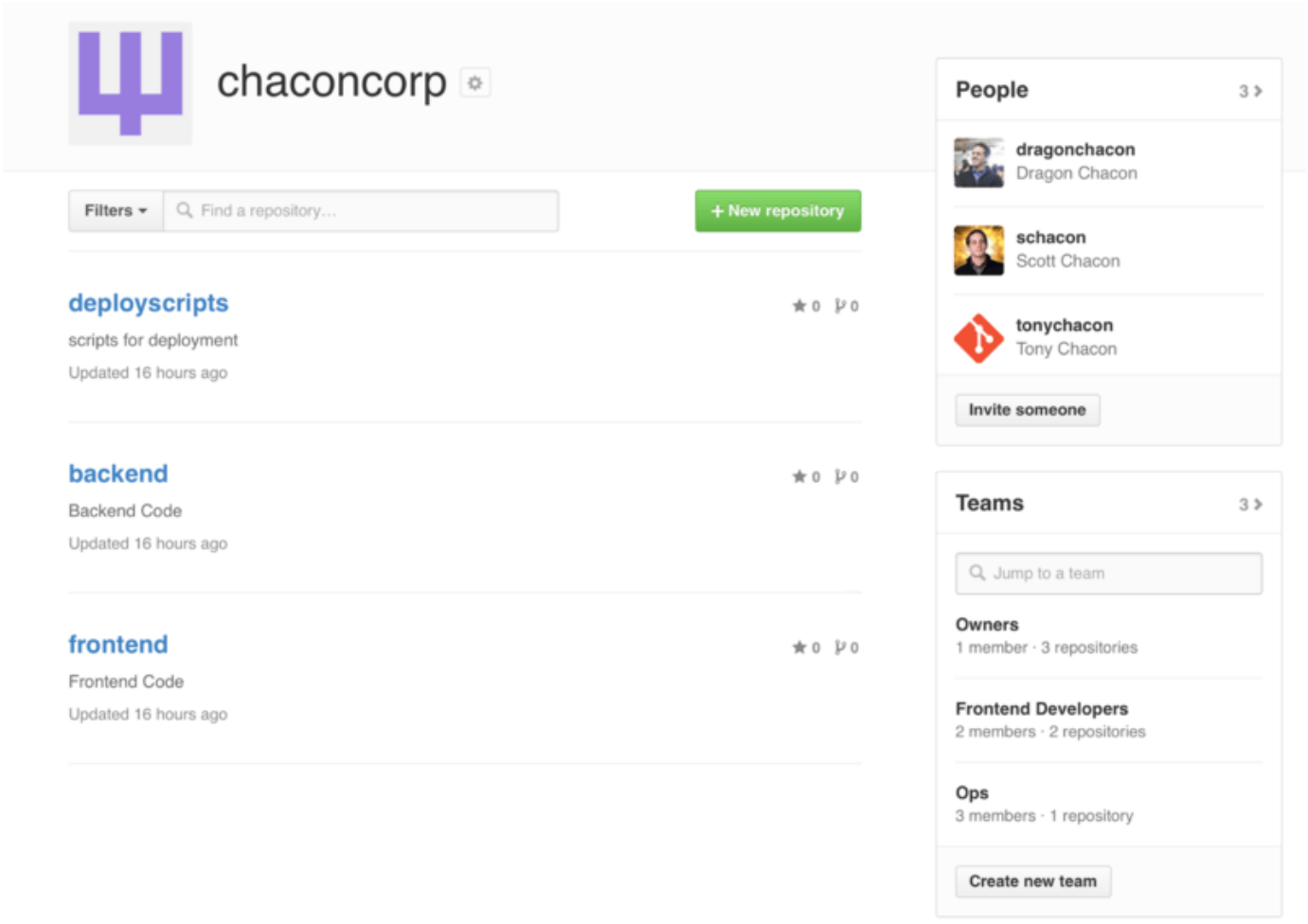



*Figure 126. 組織アカウントのページ*

チームの管理作業を行うには、組織アカウントのページ のページ右側にあるサイドバー Teams をクリックし

ます。
移動した先のページでは、チームにメンバーを追加したり、チームにリポジトリを追加したり、チームの設定やアクセス権を管理したりすることができます。 リポジトリに対するチームのアクセス権は、「読み込み限定」「読み書き可能」「管理者」の中から選べます。 この設定の切り替えは、チームのページ の “Settings” ボタンをクリックして行います。

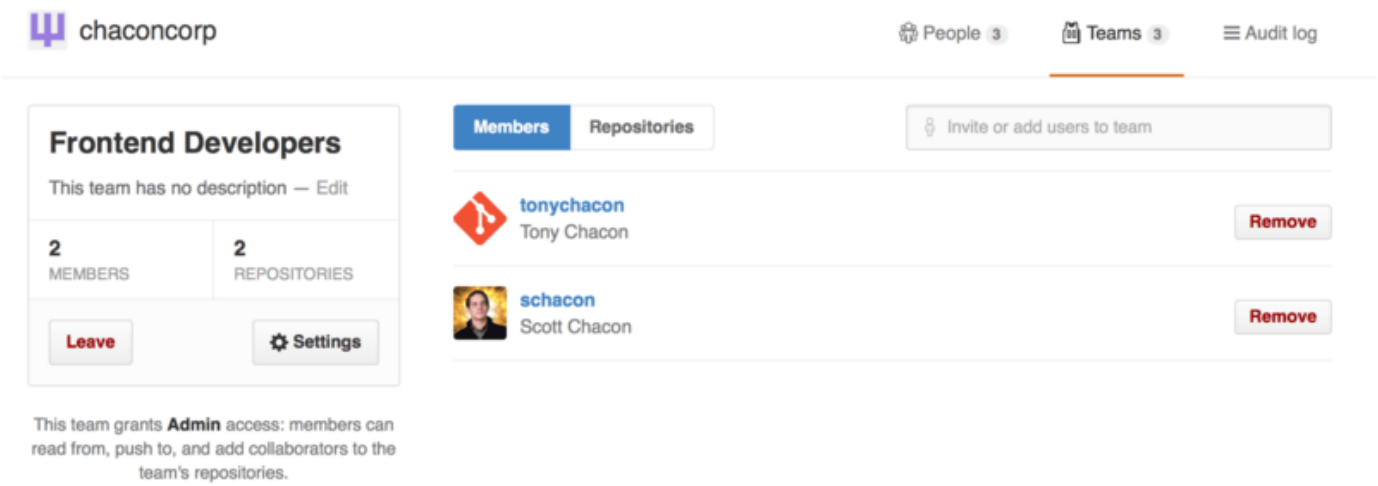



*Figure 127. チームのページ*

誰かをチームに招待すると、チームに招待されたことを伝えるメールが、その人に届きます。

さらに、チームへの (`@acmecorp/frontend` のような) 言及も、個人アカウントへの言及と同じように機能します。 ただ、個人アカウントと違うところは、このように言及した場合は、チームの **すべての** メンバーが、そのスレッドを購読した状態になるという点です。 これは、そのチームに伝えたいことがある (が、誰に伝えればいいのかわからない) という場合に便利です。

一人のユーザーが複数のチームに属することもできるので、単なるアクセス制御以外の目的でチームを使うこともできます。 たとえば、`ux` や `css` あるいは `refactoring` などのようなチームを用意して、その手の質問に対応させることもできるでしょうし、 `legal` や `colorblind` など、まったく異なる種類のチームを作ることだってできます。

## 監査ログ

組織アカウントのオーナーは、その組織の配下で起こっていることについてのあらゆる情報を取得できます。 *Audit Log* タブを開くと、組織レベルで発生した出来事やそれを行った人、そしてそれを行った場所などを確認できます。

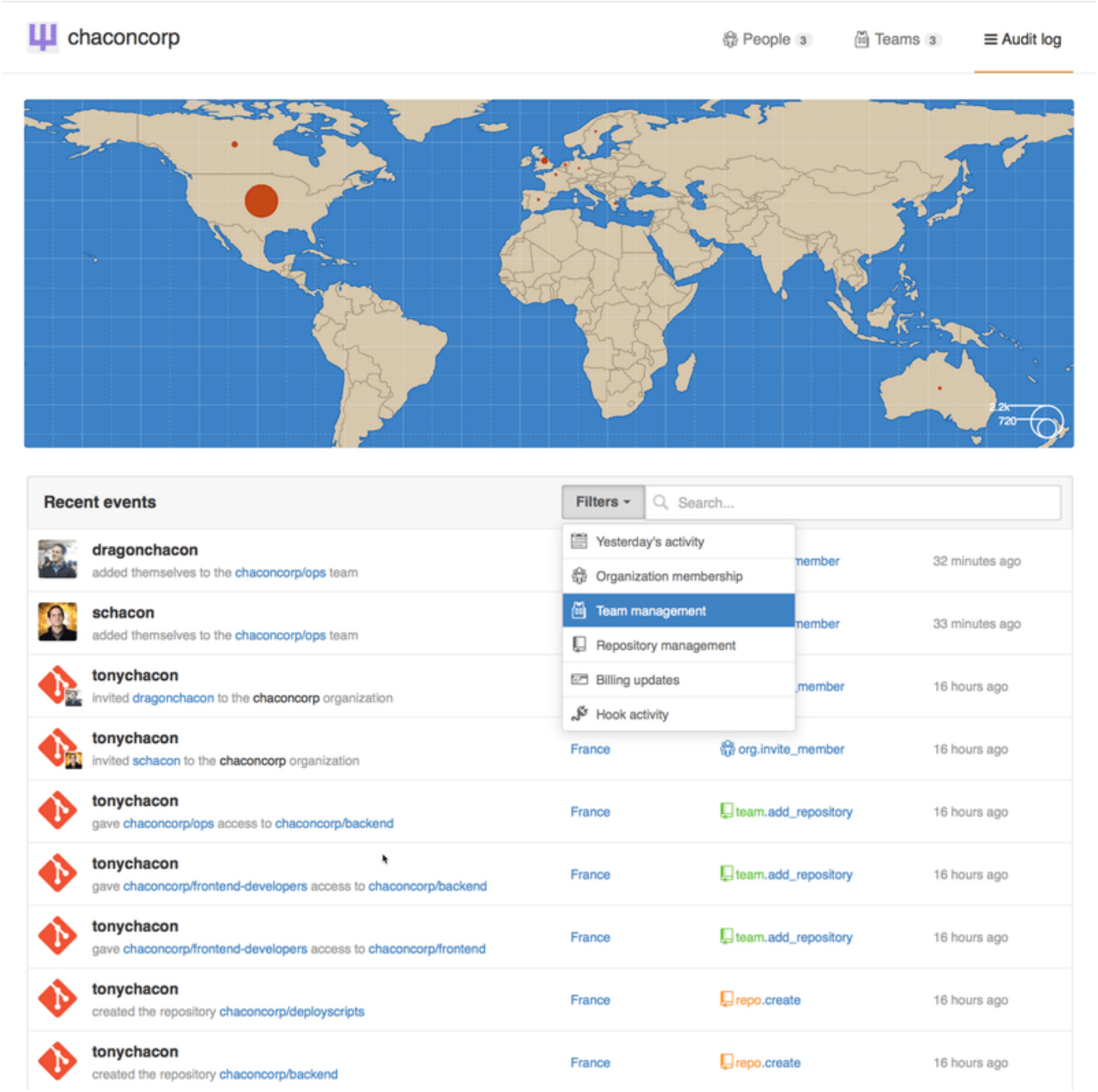



*Figure 128. 監査ログ*

このログを、特定の出来事や場所、あるいはユーザーなどに絞って確認することもできます。

# スクリプトによる GitHub の操作

ここまでで、GitHub の主要な機能や作業の流れはすべて紹介し終えました。 しかし、大規模なグループやプロジェクトでは、もう少しカスタマイズしたり、外部のサービスを組み込んだりしたくなることもあるかもしれません。

GitHub は、そういったハックも簡単にできるようになっています。 ここでは、GitHub のフックシステムとその API の使いかたを説明します。GitHub の動きが望みどおりになるようにしてみましょう。

## フック

GitHub のリポジトリのページ上にある Hooks や Services を利用すると、GitHub と外部のシステムとのやりとりを簡単に行えます。

### サービス

まずはサービスから見てみましょう。 フックやサービスの統合は、どちらもリポジトリの設定画面から行えます。 先ほど Collaborator を追加したり、デフォルトのブランチを変更したりしたのと同じ画面です。 “Webhooks and Services” タブを開くと、サービスとフックの設定画面 のような表示になるでしょう。

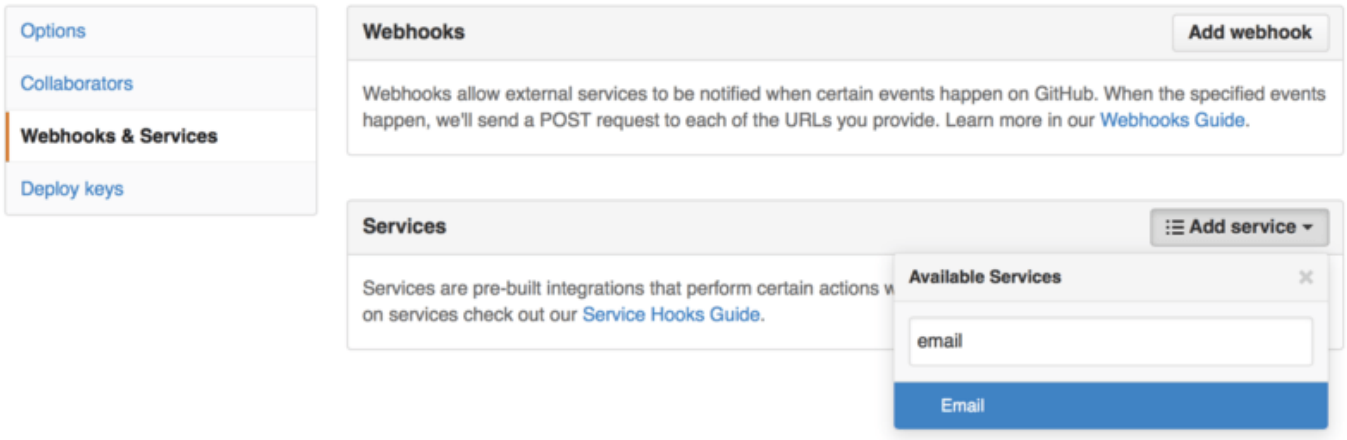



*Figure 129. サービスとフックの設定画面*

何十種類ものサービスの中から、追加するサービスを選べます。そのほとんどが、他の商用システムやオープンソースシステムとの統合を行うものです。 継続的インテグレーションサービス、バグ (課題) 追跡システム、チャットシステム、ドキュメント作成システムなどと統合できます。 ここでは、シンプルなサービスの例として、メール送信機能を組み込む方法を示します。 “Add Service” のドロップダウンから “email” を選ぶと、メールサービスの設定 のような設定画面が表示されます。

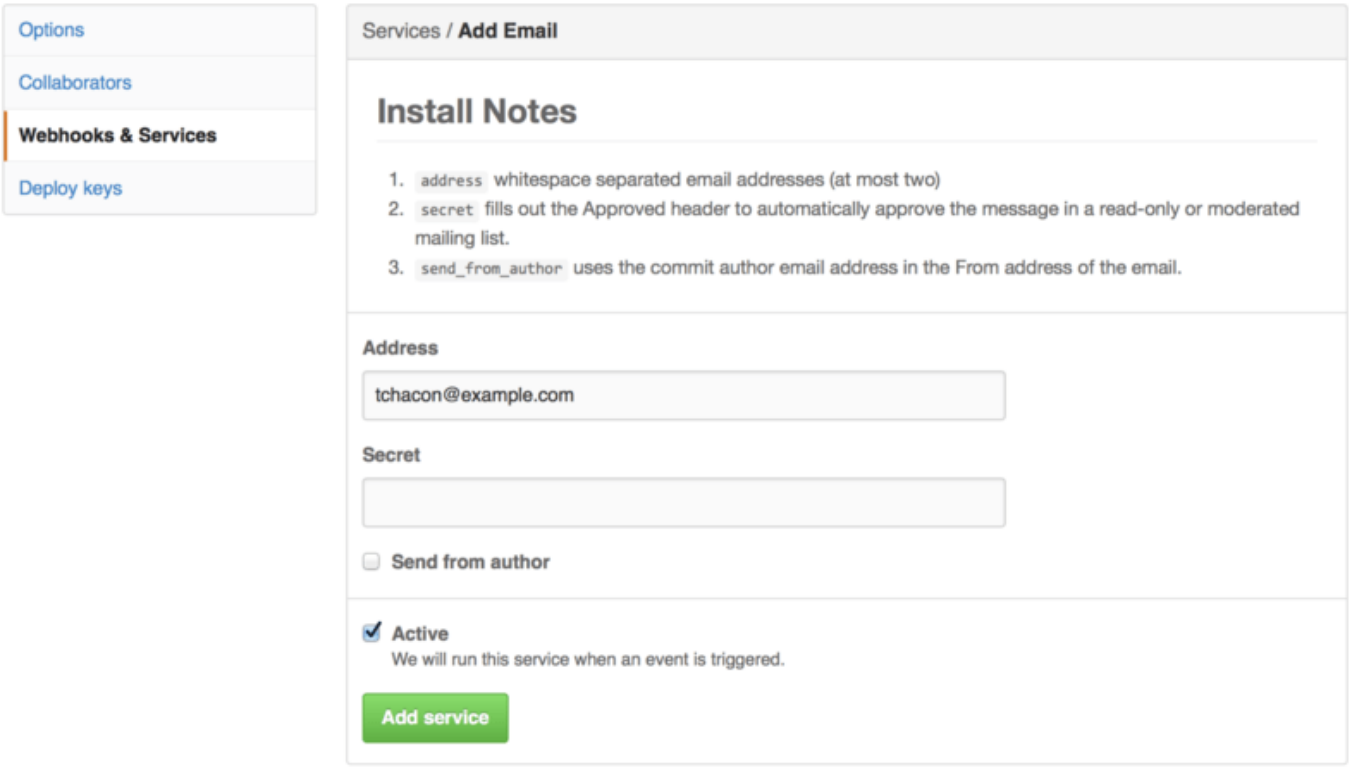



*Figure 130. メールサービスの設定*

ここで “Add service” ボタンを押すと、誰かがリポジトリにプッシュするたびに、指定したアドレスにメールが届くようになります。 サービスでは、プッシュ以外にもさまざまなイベントを待ち受けることができます。 しかし、大半のサービスは、プッシュイベントだけを待ち受けて、そのデータを使って何かをするというものです。

自分たちが使っているシステムを GitHub と統合したいという場合は、 まずここをチェックして、統合のためのサービスが用意されていないかどうかを確かめましょう。 たとえば Jenkins を使ってテストを実行している場合は、Jenkins のサービスを組み込めば、 誰かがプロジェクトにプッシュするたびにテストを実行できるようになります。

### フック

もう少し細やかな処理をしたい場合や、統合したいサービスが一覧に含まれていない場合は、 より汎用的な機能であるフックシステムを使うことができます。 GitHub リポジトリのフック機能は、きわめてシンプルです。 URL を指定すると、何かのイベントが発生するたびに、GitHub がその URL に HTTP POST を行います。

この機能を使うには、GitHub のフック情報を含む投稿を待ち受けるちょっとした Web サービスを準備して、受け取ったデータに対して何かの操作をさせればいいでしょう。

フックを有効にするには、サービスとフックの設定画面 で “Add webhook” ボタンを押します。すると、Web フックの設定 のようなページに移動します。

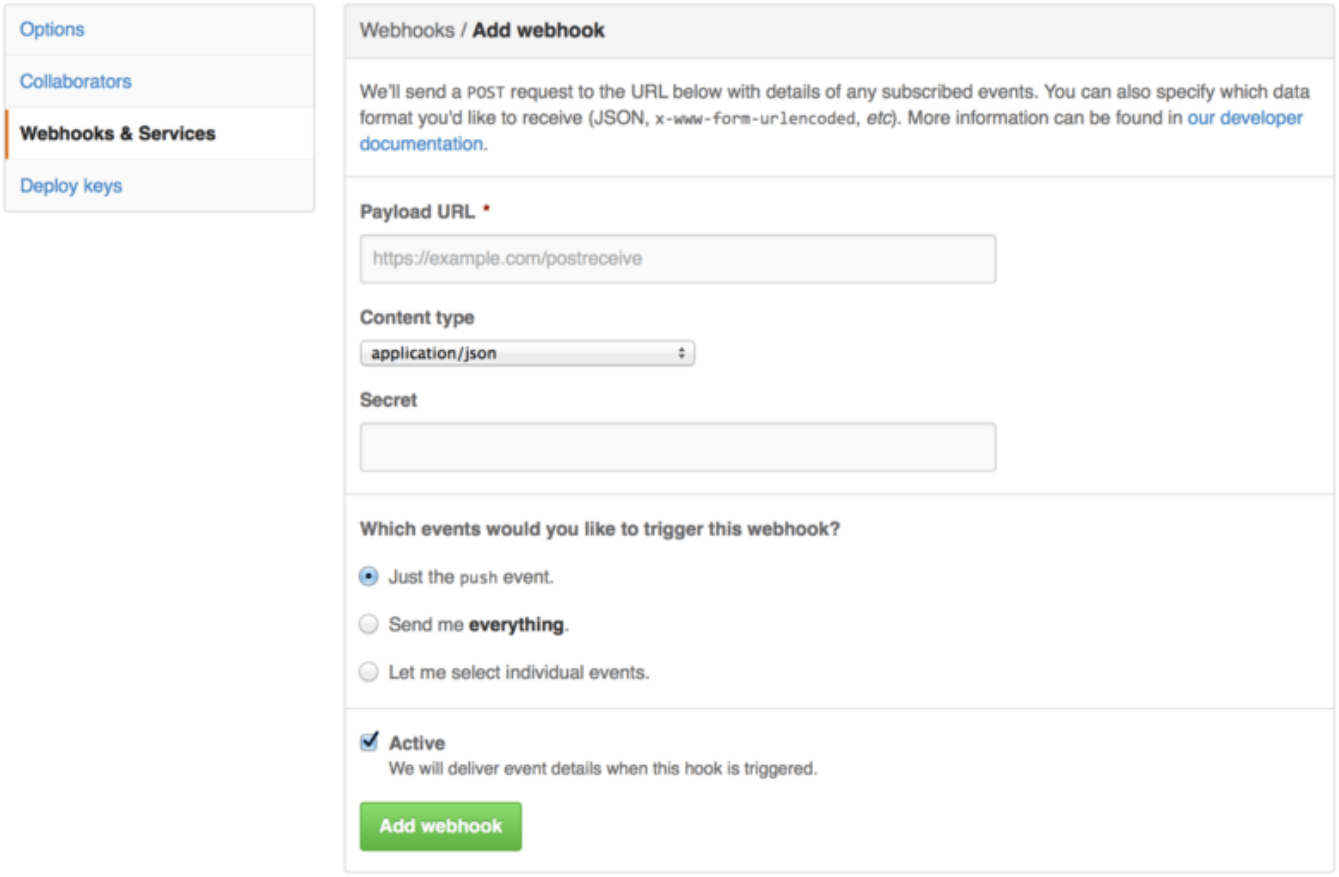



*Figure 131. Web フックの設定*

設定項目は、このようにシンプルです。 たいていは、URL とシークレットキーを入力して “Add webhook” を押すだけで済むことでしょう。 どのイベントに対して GitHub から情報を送らせたいのかを選ぶこともできます。 デフォルトでは、push イベントの情報だけを送るようになっており、 誰かがどこかのブランチにプッシュするたびに、情報が送られます。

Web フックを処理するための、ちょっとした Web サービスの例を見てみましょう。 ここでは、Ruby のフレームワークである Sinatra を使いました。コードが簡潔で、何をやっているかがわかりやすいだろうからです。

特定のプロジェクトの特定のブランチ上にある特定のファイルへの変更を、特定の誰かがプッシュしたときにだけ、メールを送ろうとしています。 こんなコードを書けば、これを簡単に実現できます。

```
require 'sinatra'
require 'json'
require 'mail'

post '/payload' do
  push = JSON.parse(request.body.read) # JSONをパースする

  # 使いたいデータを収集する
  pusher = push["pusher"]["name"]
  branch = push["ref"]

  # 変更されたファイルの一覧を取得する
  files = push["commits"].map do |commit|
    commit['added'] + commit['modified'] + commit['removed']
  end
  files = files.flatten.uniq

  # 条件をチェックする
  if pusher == 'schacon' &&
     branch == 'ref/heads/special-branch' &&
     files.include?('special-file.txt')

    Mail.deliver do
      from     'tchacon@example.com'
      to       'tchacon@example.com'
      subject  'Scott Changed the File'
      body     "ALARM"
    end
  end
end
```

このコードは、GitHub から送られてくる JSON ペイロードを受け取って、 誰がどのブランチにプッシュしたのか、そしてそのコミットがどのファイルを変更したのかを調べています。 そして、条件を満たす変更であった場合に、メールを送信します。

この手のプログラムの開発やテストに使える、便利な開発コンソールが用意されています。これは、フックの設定と同じ画面から利用できます。 このコンソールには、GitHub がそのフックを使おうとした際の記録が、直近の数回ぶん残されています。 それぞれのフックについて、この記録をたどれば、成功したかどうかを調べたり、リクエストとレスポンスの内容を確認したりすることができます。 これを利用すれば、フックのテストやデバッグがとても楽になることでしょう。

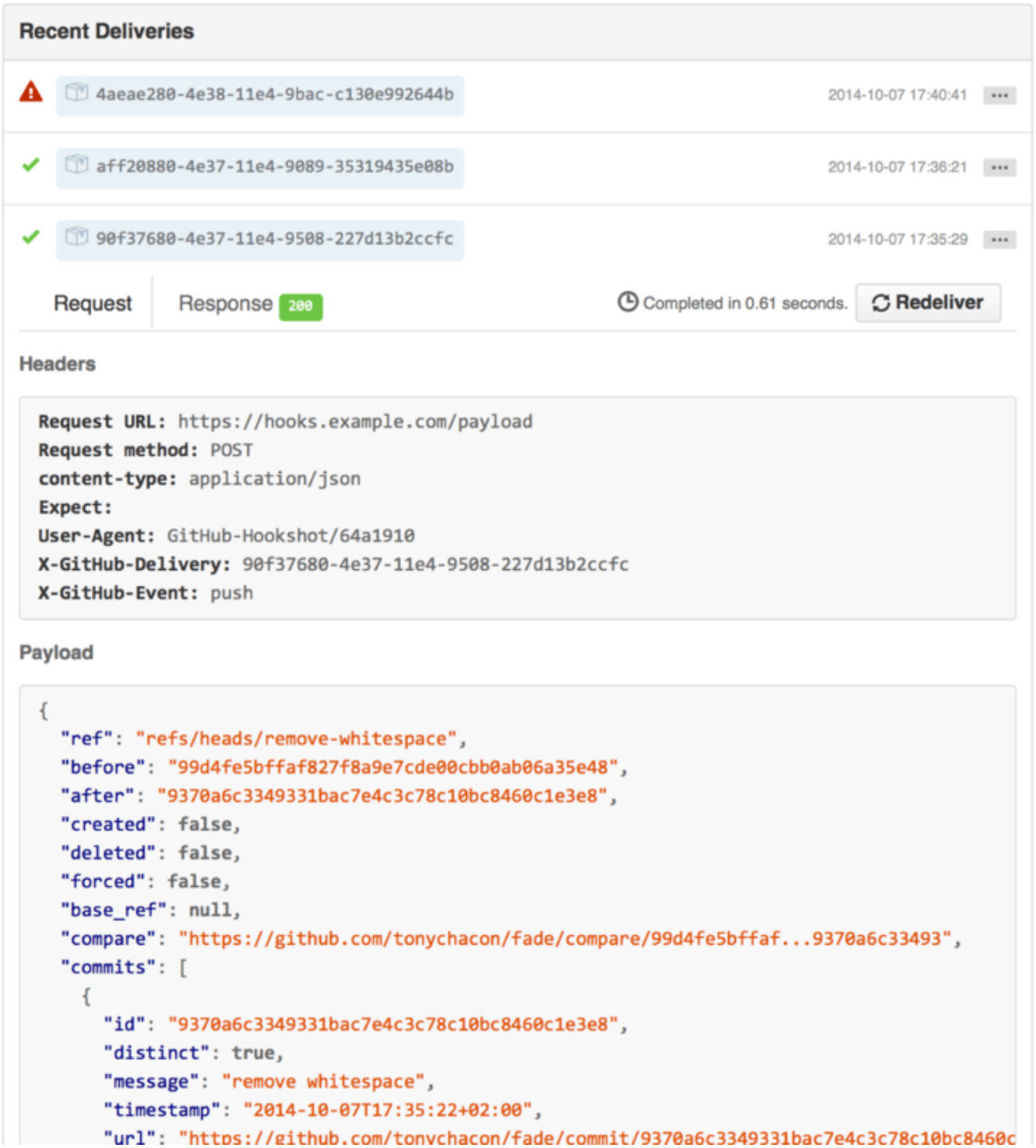



*Figure 132. Web フックのデバッグ情報*

また、このコンソールからは、任意のペイロードをサービスに再送することもできます。

Web フックの書きかたや待ち受け可能なイベントなどの情報は、GitHub の開発者向けドキュメント (https://developer.github.com/webhooks/) をご覧ください。

## GitHub API

サービスやフックを使えば、リポジトリ上で発生したイベントについてのプッシュ通知を受け取ることができます。 しかし、そのイベントについて、さらに詳しい情報が知りたい場合はどうすればいいのでしょう？ Collaborator への追加や issue へのラベル付けなどを自動化したい場合は、どうすればいいのでしょう？

そんなときに使えるのが GitHub API です。 GitHub はさまざまな API エンドポイントを提供しており、Web サイト上でできることならほぼすべて、自動化できます。 ここでは、API の認証と接続の方法を学び、 さらに、issue にコメントしたりプルリクエストの状態を変更したりといった操作を、API を使って行います。

## 基本的な使いかた

一番基本的な使いかたは、認証が不要なエンドポイントへのシンプルな GET リクエストです。 ユーザーの情報や、オープンなプロジェクトの情報 (読み込みのみ) を取得できます。 たとえば、“schacon” というユーザーに関する情報を知りたければ、次のようにします。

```
$ curl https://api.github.com/users/schacon
{
  "login": "schacon",
  "id": 70,
  "avatar_url": "https://avatars.githubusercontent.com/u/70",
# …
  "name": "Scott Chacon",
  "company": "GitHub",
  "following": 19,
  "created_at": "2008-01-27T17:19:28Z",
  "updated_at": "2014-06-10T02:37:23Z"
}
```

このようなエンドポイントが山ほど用意されており、組織やプロジェクト、issue、コミットなどなど、GitHub 上で公開されているあらゆる情報を取得できます。 API を使って任意の Markdown をレンダリングしたり、`.gitignore` のテンプレートを探したりといったことすらできるのです。

```
$ curl https://api.github.com/gitignore/templates/Java
{
  "name": "Java",
  "source": "*.class

# Mobile Tools for Java (J2ME)
.mtj.tmp/

# Package Files #
*.jar
*.war
*.ear

# virtual machine crash logs, see
http://www.java.com/en/download/help/error_hotspot.xml
hs_err_pid*
"
}
```

## Issue へのコメント

しかし、Issue やプルリクエストに対してコメントしたり、プライベートなコンテンツを操作したりしたい場合は、 認証が必要になります。

認証には、いくつかの方法があります。 ベーシック認証を使ってユーザー名とパスワードを渡すこともできますが、 通常は、アクセストークンを使うことをお勧めします。 アクセストークンは、自分のアカウントの設定ページの “Applications” タブから生成できます。

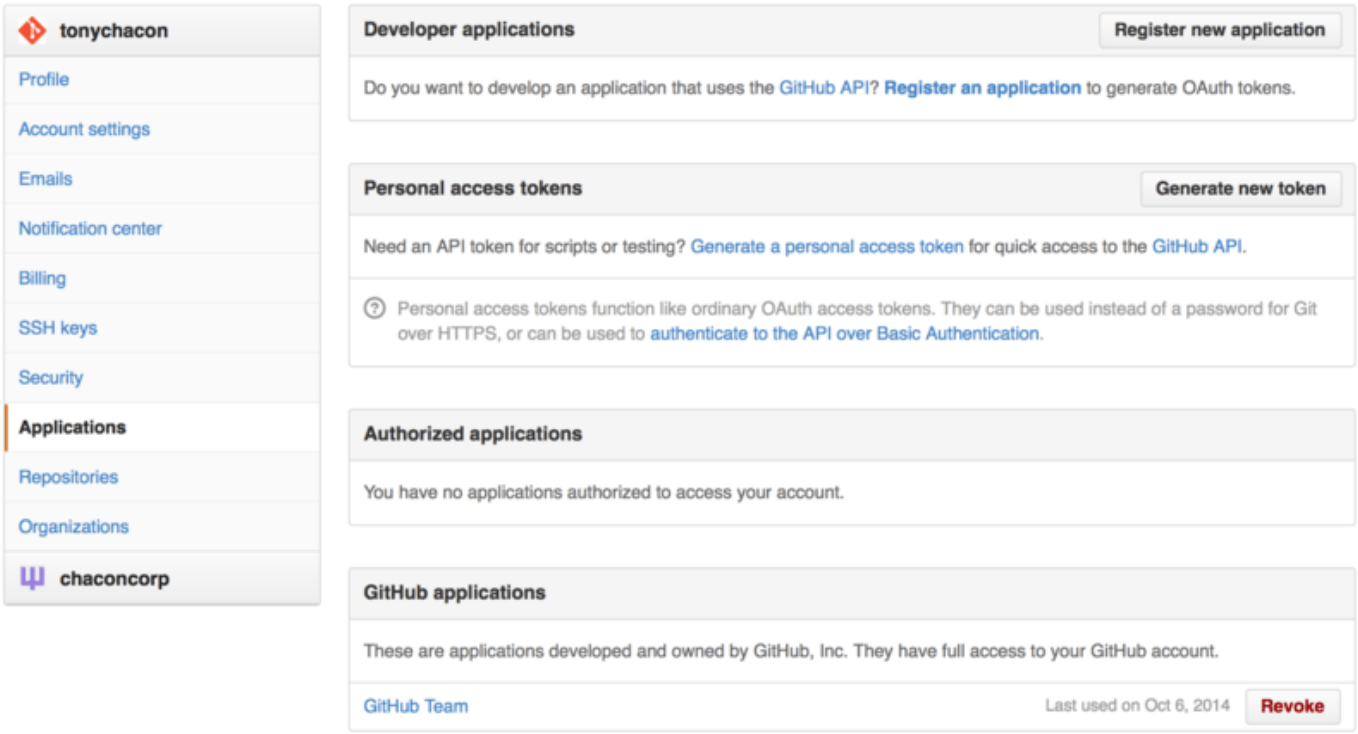



*Figure 133. 設定ページの “Applications” タブからの、アクセストークンの生成*

ここでは、新しいトークンを利用するスコープや、そのトークンについての説明の入力を求められます。 わかりやすい説明を登録するようにしましょう。 そのトークンを使っているスクリプトやアプリケーションを利用しなくなったときに、どのトークンを破棄すればいいのかが、わかりやすくなります。

GitHub は、生成したトークンを一度だけしか表示しません。忘れずにコピーしましょう。 これを使えば、ユーザー名やパスワードを使わなくても、スクリプト内で認証できるようになります。 この方式の利点は、やりたいことにあわせてトークンのスコープを絞れることと、 不要になったトークンを破棄できることです。

さらに、利用制限を緩和できるというメリットもあります。 認証なしの場合は、一時間当たり60リクエストまでという制限がかかります。 認証を済ませると、この制限が、一時間当たり5,000リクエストまでに緩和されます。

では、API を使って issue にコメントをしてみましょう。 ここでは、Issue #6 にコメントします。 そのためには、`repos/<user>/<repo>/issues/<num>/comments` に対して HTTP POST リクエストを送ります。 その際に、先ほど生成したトークンを Authorization ヘッダに含めます。

```
$ curl -H "Content-Type: application/json" \
       -H "Authorization: token TOKEN" \
       --data '{"body":"A new comment, :+1:"}' \
       https://api.github.com/repos/schacon/blink/issues/6/comments
{
  "id": 58322100,
  "html_url": "https://github.com/schacon/blink/issues/6#issuecomment-
58322100",
  ...
  "user": {
    "login": "tonychacon",
    "id": 7874698,
    "avatar_url": "https://avatars.githubusercontent.com/u/7874698?v=2",
    "type": "User",
  },
  "created_at": "2014-10-08T07:48:19Z",
  "updated_at": "2014-10-08T07:48:19Z",
  "body": "A new comment, :+1:"
}
```

さて、実際にこの issue のページを開いてみると、GitHub API を使って投稿したコメント のようにコメントに成功していることがわかるでしょう。

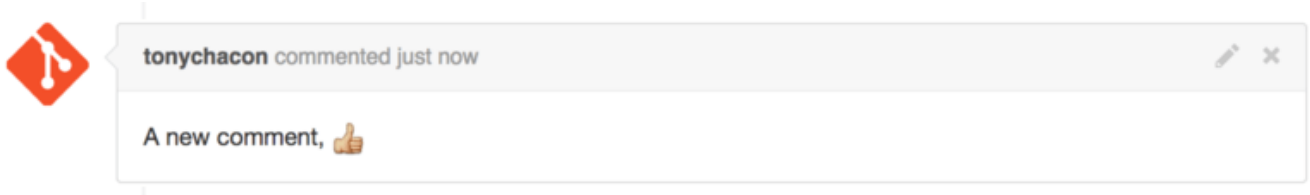



*Figure 134. GitHub API を使って投稿したコメント*

API を使えば、Web サイト上でできることならほぼすべて実行できます。 マイルストーンの作成や設定、Issue やプルリクエストの担当者の割り当て、ラベルの作成や変更、 コミット情報へのアクセス、新しいコミットやブランチの作成、 プルリクエストのオープン、クローズ、そしてマージ、 チームの作成や編集、プルリクエストの特定の行へのコメント、 サイト内検索なども、API で行えます。

## プルリクエストのステータスの変更

最後にもうひとつ、サンプルを見てみましょう。これは、プルリクエストに対応するときに、とても便利なものです。 各コミットには、ひとつあるいは複数のステータスを持たせることができるようになっています。そして、API を使って、このステータスを追加したり、問い合わせたりすることができるのです。

継続的インテグレーションやテスティングのサービスの大半は、この API を使っています。 コードがプッシュされたらそのコードをテストして、そのコミットがすべてのテストをパスした場合は、結果報告を返したりしているのです。 同様に、コミットメッセージが適切な書式になっているかどうかを調べたり、 コードを貢献するときのガイドラインに沿っているかどうかを調べたり、 適切に署名されているかどうかを調べたり、さまざまなことを行えます。

ここでは、コミットメッセージに `Signed-off-by`
という文字列が含まれているかどうかを調べるちょっとした Web サービスを、 リポジトリのフック機能で利用することを考えてみましょう。

```
require 'httparty'
require 'sinatra'
require 'json'

post '/payload' do
  push = JSON.parse(request.body.read) # JSONをパースする
  repo_name = push['repository']['full_name']

  # コミットメッセージを調べる
  push["commits"].each do |commit|

    # 文字列 Signed-off-by を探す
    if /Signed-off-by/.match commit['message']
      state = 'success'
      description = 'Successfully signed off!'
    else
      state = 'failure'
      description = 'No signoff found.'
    end

    # 状態を GitHub に投稿する
    sha = commit["id"]
    status_url =
"https://api.github.com/repos/#{repo_name}/statuses/#{sha}"

    status = {
      "state"       => state,
      "description" => description,
      "target_url"  => "http://example.com/how-to-signoff",
      "context"     => "validate/signoff"
    }
    HTTParty.post(status_url,
      :body => status.to_json,
      :headers => {
        'Content-Type'  => 'application/json',
        'User-Agent'    => 'tonychacon/signoff',
        'Authorization' => "token #{ENV['TOKEN']}" }
    )
  end
end
```

おそらく、何をやっているのかを追うのはそんなに難しくないかと思います。 この Web フックは、プッシュされたコミットについて、コミットメッセージに *Signed-off-by* という文字列が含まれるているかどうかを調べて、 API エンドポイント `/repos/<user>/<repo>/statuses/<commit_sha>` への HTTP POST でス

テータスを指定します。

ここで送信できる情報は、ステータス (*success*, *failure*, *error*) と説明文、詳細な情報を得るための URL、 そして単一のコミットに複数のステータスがある場合の “コンテキスト” です。 たとえば、テスティングサービスがステータスを送ることもあれば、このサンプルのようなバリデーションサービスがステータスを送ることもあります。 それらを区別するのが “context” フィールドです。

誰かが GitHub 上で新しいプルリクエストを作ったときに、もしこのフックを設定していれば、API で設定したコミットのステータス のようになるでしょう。

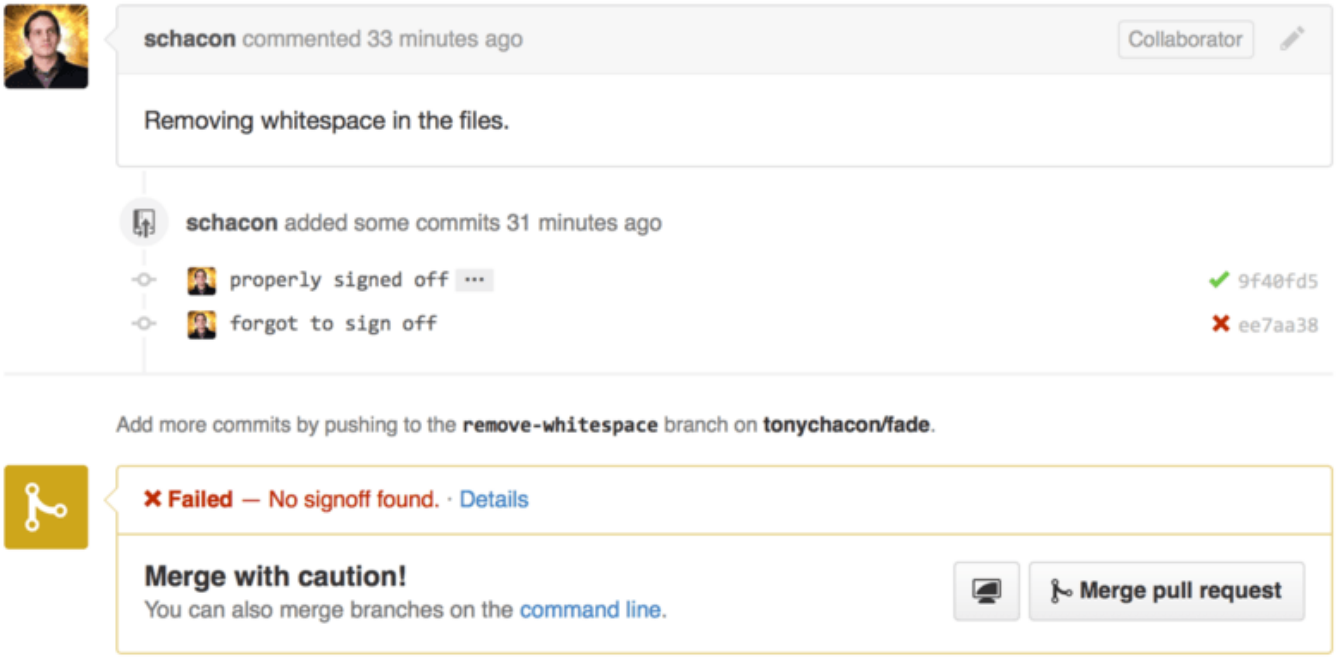



*Figure 135. API で設定したコミットのステータス*

メッセージに “Signed-off-by” という文字列が含まれているコミットの隣にはグリーンのチェックマークが表示されています。 一方、作者が署名し忘れたコミットの隣には、赤い×印がついています。 また、そのプルリクエストの最新のコミットのステータスを見て、もし failure だったら警告を発しているということもわかります。 テストの結果を見てこの API を使うようにすると、とても便利です。テストが通らなかったコミットを、うっかりマージしてしまわずに済むでしょう。

## Octokit

ここまでほぼすべてのサンプルは、`curl` を使ったシンプルな HTTP リクエストだけで実現してきましたが、オープンソースのライブラリを使えば、これらの API を、もっと慣用的な書きかたで使えるようになります。本書の執筆時点では、Go や Objective-C、Ruby、そして .NET 用のライブラリが公開されています。 詳細は http://github.com/octokit をご覧ください。HTTP がらみの大半を、あなたの代わりに処理してくれることでしょう。

これらのツールをうまく活用して GitHub をカスタマイズして、自分自身のワークフローにうまくあてはまるようにしてみましょう。 API の完全なドキュメントや、一般的な使いかたの指針は、https://developer.github.com をご覧ください。

# まとめ

これであなたも GitHub ユーザーです。 アカウントの作りかたもわかったし、 組織を管理したりリポジトリを作ったり、リポジトリにプッシュしたり、 他のプロジェクトに貢献したり、他のユーザーからの貢献を受け入れたりする方法も覚えました。 次の章では、さらに強力なツールやヒントについて学びます。 複雑な状況に対処できるようになり、本当の意味での Git の達人になれることでしょう。

# Git のさまざまなツール

Git を使ったソースコード管理のためのリポジトリの管理や保守について、日々使用するコマンドやワークフローの大半を身につけました。 ファイルの追跡やコミットといった基本的なタスクをこなせるようになっただけではなくステージングエリアの威力もいかせるようになりました。また気軽にトピックブランチを切ってマージする方法も知りました。

では、Git の非常に強力な機能の数々をさらに探っていきましょう。日々の作業でこれらを使うことはあまりありませんが、いつかは必要になるかもしれません。

## リビジョンの選択

Git で特定のコミットやコミットの範囲を指定するにはいくつかの方法があります。 明白なものばかりではありませんが、知っておくと役立つでしょう。

### 単一のリビジョン

SHA-1 ハッシュを指定すれば、コミットを明確に参照することができます。しかしそれ以外にも、より人間にやさしい方式でコミットを参照することもできます。 このセクションでは単一のコミットを参照するためのさまざまな方法の概要を説明します。

### SHA の短縮形

Git は、最初の数文字をタイプしただけであなたがどのコミットを指定したいのかを汲み取ってくれます。条件は、SHA-1 の最初の 4 文字以上を入力していることと、それでひとつのコミットが特定できる (現在のリポジトリに、入力した文字ではじまる SHA-1 のコミットがひとつしかない) ことです。

あるコミットを指定するために `git log` コマンドを実行し、とある機能を追加したコミットを見つけました。

```
$ git log
commit 734713bc047d87bf7eac9674765ae793478c50d3
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri Jan 2 18:32:33 2009 -0800

    fixed refs handling, added gc auto, updated tests

commit d921970aadf03b3cf0e71becdaab3147ba71cdef
Merge: 1c002dd... 35cfb2b...
Author: Scott Chacon <schacon@gmail.com>
Date:   Thu Dec 11 15:08:43 2008 -0800

    Merge commit 'phedders/rdocs'

commit 1c002dd4b536e7479fe34593e72e6c6c1819e53b
Author: Scott Chacon <schacon@gmail.com>
Date:   Thu Dec 11 14:58:32 2008 -0800

    added some blame and merge stuff
```

探していたのは、`1c002dd....` で始まるコミットです。`git show` でこのコミットを見るときは、次のどのコマンドでも同じ結果になります (短いバージョンで、重複するコミットはないものとします)。

```
$ git show 1c002dd4b536e7479fe34593e72e6c6c1819e53b
$ git show 1c002dd4b536e7479f
$ git show 1c002d
```

一意に特定できる範囲での SHA-1 の短縮形を Git に見つけさせることもできます。 `git log` コマンドで `--abbrev-commit` を指定すると、コミットを一意に特定できる範囲の省略形で出力します。デフォルトでは 7 文字ぶん表示しますが、それだけで SHA-1 を特定できない場合はさらに長くなります。

```
$ git log --abbrev-commit --pretty=oneline
ca82a6d changed the version number
085bb3b removed unnecessary test code
a11bef0 first commit
```

ひとつのプロジェクト内での一意性を確保するには、普通は 8 文字から 10 文字もあれば十分すぎることでしょう。

参考までに数字を挙げておきます。Linux カーネルはコミット数45万、オブジェクト数360万という巨大プロジェクトですが、SHA-1 の最初の12桁が同じになるオブジェクトは存在しません。

*SHA-1 に関するちょっとしたメモ*

「リポジトリ内のふたつのオブジェクトがたまたま同じ SHA-1 ハッシュ値を持ってしまったらどうするの?」と心配する人も多いでしょう。 実際、どうなるのでしょう?

すでにリポジトリに存在するオブジェクトと同じ SHA-1 値を持つオブジェクトをコミットしてした場合、Git はすでにそのオブジェクトがデータベースに格納されているものと判断します。 そのオブジェクトを後からどこかで取得しようとすると、常に最初のオブジェクトのデータが手元にやってきます (訳注: つまり、後からコミットした内容は存在しないことになってしまう)。

**NOTE**

しかし、そんなことはまず起こりえないということを知っておくべきでしょう。SHA-1 ダイジェストの大きさは 20 バイト (160 ビット) です。ランダムなハッシュ値がつけられた中で、たった一つの衝突が 50% の確率で発生するために必要なオブジェクトの数は約 2^80 となります (衝突の可能性の計算式は `p = (n(n-1)/2) * (1/2^160)` です)。 2^80 は、ほぼ 1.2 x 10^24 、つまり一兆二千億のそのまた一兆倍です。 これは、地球上にあるすべての砂粒の数の千二百倍にあたります。

SHA-1 の衝突を見るにはどうしたらいいのか、ひとつの例をごらんに入れましょう。 地球上の人類 65 億人が全員プログラムを書いていたとします。そしてその全員が、Linux カーネルのこれまでの開発履歴 (360 万の Git オブジェクト) と同等のコードを一秒で書き上げ、馬鹿でかい単一の Git リポジトリにプッシュしていくとします。これを2年ほど続けると、SHA-1 オブジェクトの衝突がひとつでも発生する可能性がやっと 50% になります。 それよりも「あなたの所属する開発チームの全メンバーが、同じ夜にそれぞれまったく無関係の事件で全員オオカミに殺されてしまう」可能性のほうがよっぽど高いことでしょう。

## ブランチの参照

特定のコミットを参照するのに一番直感的なのは、そのコミットを指すブランチがある場合です。 コミットオブジェクトや SHA-1 値を指定する場面ではどこでも、その代わりにブランチ名を指定することができます。 たとえば、あるブランチ上の最新のコミットを表示したい場合は次のふたつのコマンドが同じ意味となります (`topic1` ブランチが `ca82a6d` を指しているものとします)。

```
$ git show ca82a6dff817ec66f44342007202690a93763949
$ git show topic1
```

あるブランチがいったいどの SHA を指しているのか、あるいはその他の例の内容が結局のところどの SHA に行き着くのかといったことを知るには、Git の調査用ツールである `rev-parse` を使います。 こういった調査用ツールのより詳しい情報は Gitの内側 で説明します。`rev-parse` は低レベルでの操作用のコマンドであり、日々の操作で使うためのものではありません。 しかし、今実際に何が起こっているのかを知る必要があるときなどには便利です。 ブランチ上で `rev-parse` を実行すると、このようになります。

```
$ git rev-parse topic1
ca82a6dff817ec66f44342007202690a93763949
```

### 参照ログの短縮形

あなたがせっせと働いている間に Git が裏でこっそり行っていることのひとつが、“参照ログ” (reflog) の管理です。これは、HEAD とブランチの参照が過去数ヶ月間どのように動いてきたかをあらわすものです。

参照ログを見るには `git reflog` を使います。

```
$ git reflog
734713b HEAD@{0}: commit: fixed refs handling, added gc auto, updated
d921970 HEAD@{1}: merge phedders/rdocs: Merge made by recursive.
1c002dd HEAD@{2}: commit: added some blame and merge stuff
1c36188 HEAD@{3}: rebase -i (squash): updating HEAD
95df984 HEAD@{4}: commit: # This is a combination of two commits.
1c36188 HEAD@{5}: rebase -i (squash): updating HEAD
7e05da5 HEAD@{6}: rebase -i (pick): updating HEAD
```

何らかの理由でブランチの先端が更新されるたびに、Git はその情報をこの一時履歴に格納します。そして、このデータを使って過去のコミットを指定することもできます。リポジトリの HEAD の五つ前の状態を知りたい場合は、先ほど見た reflog の出力のように `@{n}` 形式で参照することができます。

```
$ git show HEAD@{5}
```

この構文を使うと、指定した期間だけさかのぼったときに特定のブランチがどこを指していたかを知ることもできます。 たとえば `master` ブランチの昨日の状態を知るには、このようにします。

```
$ git show master@{yesterday}
```

こうすると、そのブランチの先端が昨日どこを指していたかを表示します。 この技が使えるのは参照ログにデータが残っている間だけなので、直近数ヶ月よりも前のコミットについては使うことができません。

参照ログの情報を `git log` の出力風の表記で見るには `git log -g` を実行します。

```
$ git log -g master
commit 734713bc047d87bf7eac9674765ae793478c50d3
Reflog: master@{0} (Scott Chacon <schacon@gmail.com>)
Reflog message: commit: fixed refs handling, added gc auto, updated
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri Jan 2 18:32:33 2009 -0800

    fixed refs handling, added gc auto, updated tests

commit d921970aadf03b3cf0e71becdaab3147ba71cdef
Reflog: master@{1} (Scott Chacon <schacon@gmail.com>)
Reflog message: merge phedders/rdocs: Merge made by recursive.
Author: Scott Chacon <schacon@gmail.com>
Date:   Thu Dec 11 15:08:43 2008 -0800

    Merge commit 'phedders/rdocs'
```

参照ログの情報は、完全にローカルなものであることに気をつけましょう。これは、あなた自身が自分のリポジトリで何をしたのかを示す記録です。 つまり、同じリポジトリをコピーした別の人の参照ログとは異なる内容になります。また、最初にリポジトリをクローンした直後の参照ログは空となります。まだリポジトリ上であなたが何もしていないからです。 `git show HEAD@{2.months.ago}` が動作するのは、少なくとも二ヶ月以上前にそのリポジトリをクローンした場合のみで、もしつい 5 分前にクローンしたばかりなら何も結果を返しません。

### 家系の参照

コミットを特定する方法として他によく使われるのが、その家系をたどっていく方法です。 参照の最後に `^` をつけると、Git はそれを「指定したコミットの親」と解釈します。 あなたのプロジェクトの歴史がこのようになっていたとしましょう。

```
$ git log --pretty=format:'%h %s' --graph
* 734713b fixed refs handling, added gc auto, updated tests
*   d921970 Merge commit 'phedders/rdocs'
|\
| * 35cfb2b Some rdoc changes
* | 1c002dd added some blame and merge stuff
|/
* 1c36188 ignore *.gem
* 9b29157 add open3_detach to gemspec file list
```

直前のコミットを見るには `HEAD^` を指定します。これは “HEAD の親” という意味になります。

```
$ git show HEAD^
commit d921970aadf03b3cf0e71becdaab3147ba71cdef
Merge: 1c002dd... 35cfb2b...
Author: Scott Chacon <schacon@gmail.com>
Date:   Thu Dec 11 15:08:43 2008 -0800

    Merge commit 'phedders/rdocs'
```

^ の後に数字を指定することもできます。たとえば d921970^2 は “d921970 の二番目の親” という意味になります。 これが役立つのはマージコミット (親が複数存在する) のときくらいでしょう。 最初の親はマージを実行したときにいたブランチとなり、二番目の親は取り込んだブランチ上のコミットとなります。

```
$ git show d921970^
commit 1c002dd4b536e7479fe34593e72e6c6c1819e53b
Author: Scott Chacon <schacon@gmail.com>
Date:   Thu Dec 11 14:58:32 2008 -0800

    added some blame and merge stuff

$ git show d921970^2
commit 35cfb2b795a55793d7cc56a6cc2060b4bb732548
Author: Paul Hedderly <paul+git@mjr.org>
Date:   Wed Dec 10 22:22:03 2008 +0000

    Some rdoc changes
```

家系の指定方法としてもうひとつよく使うのが ~ です。 これも最初の親を指します。つまり HEAD~ と HEAD^ は同じ意味になります。 違いが出るのは、数字を指定したときです。 HEAD~2 は「最初の親の最初の親」、つまり「祖父母」という意味になります。指定した数だけ、順に最初の親をさかのぼっていくことになります。 たとえば、先ほど示したような歴史上では HEAD~3 は次のようになります。

```
$ git show HEAD~3
commit 1c3618887afb5fbcbea25b7c013f4e2114448b8d
Author: Tom Preston-Werner <tom@mojombo.com>
Date:   Fri Nov 7 13:47:59 2008 -0500

    ignore *.gem
```

これは HEAD^^^ のようにあらわすこともできます。これは「最初の親の最初の親の最初の親」という意味になります。

```
$ git show HEAD^^^
commit 1c3618887afb5fbcbea25b7c013f4e2114448b8d
Author: Tom Preston-Werner <tom@mojombo.com>
Date:   Fri Nov 7 13:47:59 2008 -0500

    ignore *.gem
```

これらふたつの構文を組み合わせることもできます。直近の参照 (マージコミットだったとします) の二番目の親を取得するには HEAD~3^2 などとすればいいのです。

## コミットの範囲指定

個々のコミットを指定できるようになったので、次はコミットの範囲を指定する方法を覚えていきましょう。これは、ブランチをマージするときに便利です。たくさんのブランチがある場合など、「で、このブランチの作業のなかでまだメインブランチにマージしていないのはどれだったっけ?」といった疑問を解決するために範囲指定を使えます。

### ダブルドット

範囲指定の方法としてもっとも一般的なのが、ダブルドット構文です。 これは、ひとつのコミットからはたどれるけれどもうひとつのコミットからはたどれないというコミットの範囲を Git に調べさせるものです。 範囲指定選択用の歴史の例 のようなコミット履歴を例に考えましょう。

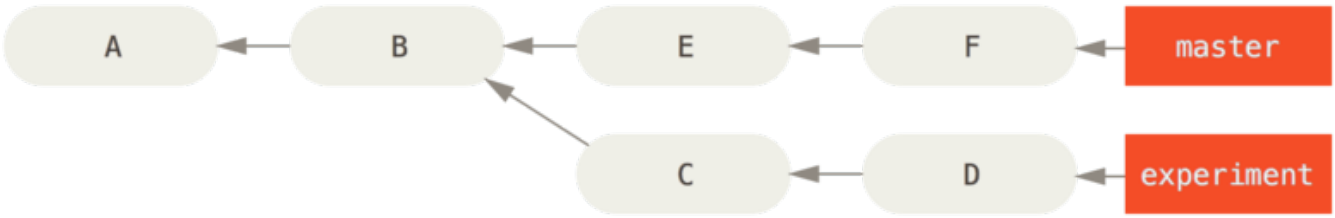



*Figure 136. 範囲指定選択用の歴史の例*

experiment ブランチの内容のうち、まだ master ブランチにマージされていないものを調べることになりました。 対象となるコミットのログを見るには、Git に master..experiment と指示します。これは「experiment からはたどれるけれど、master からはたどれないすべてのコミット」という意味です。 説明を短く簡潔にするため、実際のログの出力のかわりに上の図の中でコミットオブジェクトをあらわす文字を使うことにします。

```
$ git log master..experiment
D
C
```

もし逆に、master には存在するけれども experiment には存在しないすべてのコミットが知りたいのなら、ブランチ名を逆にすればいいのです。 experiment..master とすれば、master のすべてのコミットのうち experiment からたどれないものを取得できます。

```
$ git log experiment..master
F
E
```

これは、experiment ブランチを最新の状態に保つために何をマージしなければならないのかを知るのに便利です。 もうひとつ、この構文をよく使う例としてあげられるのが、これからリモートにプッシュしようとしている内容を知りたいときです。

```
$ git log origin/master..HEAD
```

このコマンドは、現在のブランチ上でのコミットのうち、リモート origin の master ブランチに存在しないものをすべて表示します。 現在のブランチが origin/master を追跡しているときに git push を実行すると、git log origin/master..HEAD で表示されたコミットがサーバーに転送されます。 この構文で、どちらか片方を省略することもできます。その場合、Git は省略したほうを HEAD とみなします。 たとえば、git log origin/master.. と入力すると先ほどの例と同じ結果が得られます。Git は、省略した側を HEAD に置き換えて処理を進めるのです。

#### 複数のポイント

ダブルドット構文は、とりあえず使うぶんには便利です。しかし、二つよりもっと多くのブランチを指定してリビジョンを特定したいこともあるでしょう。複数のブランチの中から現在いるブランチには存在しないコミットを見つける場合などです。 Git でこれを行うには ^ 文字を使うか、あるいはそこからたどりつけるコミットが不要な参照の前に --not をつけます。 これら三つのコマンドは、同じ意味となります。

```
$ git log refA..refB
$ git log ^refA refB
$ git log refB --not refA
```

これらの構文が便利なのは、二つよりも多くの参照を使って指定できるというところです。ダブルドット構文では二つの参照しか指定できませんでした。 たとえば、refA と refB のどちらかからはたどれるけれども refC からはたどれないコミットを取得したい場合は、次のいずれかを実行します。

```
$ git log refA refB ^refC
$ git log refA refB --not refC
```

この非常に強力なリビジョン問い合わせシステムを使えば、今あなたのブランチに何があるのかを知るのに非常に役立つことでしょう。

#### トリプルドット

範囲指定選択の主な構文であとひとつ残っているのがトリプルドット構文です。これは、ふたつの参照のうちどちらか一方からのみたどれるコミット (つまり、両方からたどれるコミットは含まない) を指定します。 範囲指定選択用の歴史の例 で示したコミット履歴の例を振り返ってみましょう。 master あるいは

experiment
に存在するコミットのうち、両方に存在するものを除いたコミットを知りたい場合は次のようにします。

```
$ git log master...experiment
F
E
D
C
```

これは通常の log の出力と同じですが、これら四つのコミットについての情報しか表示しません。表示順は、従来どおりコミット日時順となります。

この場合に log コマンドでよく使用するスイッチが --left-right です。このスイッチは、それぞれのコミットがどちら側に存在するのかを表示します。 これを使うとデータをより活用しやすくなるでしょう。

```
$ git log --left-right master...experiment
< F
< E
> D
> C
```

これらのツールを使えば、より簡単に「どれを調べたいのか」を Git に伝えられるようになります。

## 対話的なステージング

Git には、コマンドラインでの作業をしやすくするためのスクリプトがいくつか付属しています。 ここでは、対話コマンドをいくつか紹介しましょう。これらを使うと、コミットの内容に細工をして特定のコミットだけとかファイルの中の一部だけとかを含めるようにすることが簡単にできるようになります。 大量のファイルを変更した後に、それをひとつの馬鹿でかいコミットにしてしまうのではなくテーマごとの複数のコミットに分けて処理したい場合などに非常に便利です。 このようにして各コミットを論理的に独立した状態にしておけば、同僚によるレビューも容易になります。 git add に -i あるいは --interactive というオプションをつけて実行すると、Git は対話シェルモードに移行し、このように表示されます。

```
$ git add -i
           staged     unstaged path
  1:    unchanged        +0/-1 TODO
  2:    unchanged        +1/-1 index.html
  3:    unchanged        +5/-1 lib/simplegit.rb

*** Commands ***
  1: status     2: update      3: revert     4: add untracked
  5: patch      6: diff        7: quit       8: help
What now>
```

このコマンドは、ステージングエリアに関する情報を違った観点で表示します。`git status` で得られる情報と基本的には同じですが、より簡潔で有益なものとなっています。 ステージした変更が左側、そしてステージしていない変更が右側に表示されます。

Commands セクションでは、さまざまなことができるようになっています。 ファイルをステージしたりステージングエリアから戻したり、ファイルの一部だけをステージしたりまだ追跡されていないファイルを追加したり、あるいは何がステージされたのかを diff で見たりといったことが可能です。

`What now>` プロンプトで `2` または `u` と入力すると、どのファイルをステージするかを聞いてきます。

```
What now> 2
           staged     unstaged path
  1:    unchanged        +0/-1 TODO
  2:    unchanged        +1/-1 index.html
  3:    unchanged        +5/-1 lib/simplegit.rb
Update>>
```

TODO と index.html をステージするには、その番号を入力します。

```
Update>> 1,2
           staged     unstaged path
* 1:    unchanged        +0/-1 TODO
* 2:    unchanged        +1/-1 index.html
  3:    unchanged        +5/-1 lib/simplegit.rb
Update>>
```

ファイル名の横に `*` がついていれば、そのファイルがステージ対象として選択されたことを意味します。`Update>>` プロンプトで何も入力せずに Enter を押すと、選択されたすべてのファイルを Git がステージします。

```
Update>>
updated 2 paths

*** Commands ***
  1: status     2: update      3: revert     4: add untracked
  5: patch      6: diff        7: quit       8: help
What now> 1
           staged     unstaged path
  1:        +0/-1      nothing TODO
  2:        +1/-1      nothing index.html
  3:    unchanged        +5/-1 lib/simplegit.rb
```

TODO と index.html がステージされ、simplegit.rb はまだステージされていないままです。ここで仮に TODO ファイルのステージを取り消したくなったとしたら、`3` あるいは `r` (revert の r) を選択します。

```
*** Commands ***
  1: status     2: update      3: revert     4: add untracked
  5: patch      6: diff        7: quit       8: help
What now> 3
           staged     unstaged path
  1:        +0/-1      nothing TODO
  2:        +1/-1      nothing index.html
  3:    unchanged        +5/-1 lib/simplegit.rb
Revert>> 1
           staged     unstaged path
* 1:        +0/-1      nothing TODO
  2:        +1/-1      nothing index.html
  3:    unchanged        +5/-1 lib/simplegit.rb
Revert>> [enter]
reverted one path
```

もう一度 Git のステータスを見ると、TODO ファイルのステージが取り消されていることがわかります。

```
*** Commands ***
  1: status     2: update      3: revert     4: add untracked
  5: patch      6: diff        7: quit       8: help
What now> 1
           staged     unstaged path
  1:    unchanged        +0/-1 TODO
  2:        +1/-1      nothing index.html
  3:    unchanged        +5/-1 lib/simplegit.rb
```

ステージした変更の diff を見るには、`6` あるいは `d` (diff の d) を使用します。 このコマンドは、ステージしたファイルの一覧を表示します。その中から、ステージされた diff を見たいファイルを選択します。 これは、コマンドラインで `git diff --cached` を使用するのと同じようなことです。

```
*** Commands ***
  1: status     2: update      3: revert     4: add untracked
  5: patch      6: diff        7: quit       8: help
What now> 6
           staged     unstaged path
  1:        +1/-1      nothing index.html
Review diff>> 1
diff --git a/index.html b/index.html
index 4d07108..4335f49 100644
--- a/index.html
+++ b/index.html
@@ -16,7 +16,7 @@ Date Finder

 <p id="out">...</p>

-<div id="footer">contact : support@github.com</div>
+<div id="footer">contact : email.support@github.com</div>

 <script type="text/javascript">
```

これらの基本的なコマンドを使えば、ステージングエリアでの対話的な追加モードを多少簡単に扱えるようになるでしょう。

### パッチのステージ

Git では、ファイルの特定の箇所だけをステージして他の部分はそのままにしておくということもできます。たとえば、simplegit.rb のふたつの部分を変更したけれど、そのうちの一方だけをステージしたいという場合があります。Git なら、そんなことも簡単です。 対話モードのプロンプトで 5 あるいは p (patch の p) と入力しましょう。 Git は、どのファイルを部分的にステージしたいのかを聞いてきます。その後、選択したファイルのそれぞれについて diff のハンクを順に表示し、ステージするかどうかをひとつひとつたずねます。

```
diff --git a/lib/simplegit.rb b/lib/simplegit.rb
index dd5ecc4..57399e0 100644
--- a/lib/simplegit.rb
+++ b/lib/simplegit.rb
@@ -22,7 +22,7 @@ class SimpleGit
   end

   def log(treeish = 'master')
-    command("git log -n 25 #{treeish}")
+    command("git log -n 30 #{treeish}")
   end

   def blame(path)
Stage this hunk [y,n,a,d,/,j,J,g,e,?]?
```

ここでは多くの選択肢があります。 何ができるのかを見るには ? を入力しましょう。

```
Stage this hunk [y,n,a,d,/,j,J,g,e,?]? ?
y - stage this hunk
n - do not stage this hunk
a - stage this and all the remaining hunks in the file
d - do not stage this hunk nor any of the remaining hunks in the file
g - select a hunk to go to
/ - search for a hunk matching the given regex
j - leave this hunk undecided, see next undecided hunk
J - leave this hunk undecided, see next hunk
k - leave this hunk undecided, see previous undecided hunk
K - leave this hunk undecided, see previous hunk
s - split the current hunk into smaller hunks
e - manually edit the current hunk
? - print help
```

たいていは、y か n で各ハンクをステージするかどうかを指定していくでしょう。しかし、それ以外にも「このファイルの残りのハンクをすべてステージする」とか「このハンクをステージするかどうかの判断を先送りする」などというオプションも便利です。 あるファイルのひとつの箇所だけをステージして残りはそのままにした場合、ステータスの出力はこのようになります。

```
What now> 1
           staged     unstaged path
  1:    unchanged        +0/-1 TODO
  2:        +1/-1      nothing index.html
  3:        +1/-1        +4/-0 lib/simplegit.rb
```

simplegit.rb のステータスがおもしろいことになっています。 ステージされた行もあれば、ステージされていない行もあるという状態です。 つまり、このファイルを部分的にステージしたというわけです。 この時点で対話的追加モードを抜けて `git commit` を実行すると、ステージした部分だけをコミットすることができます。

ファイルを部分的にステージするだけなら、対話的な追加モードに入る必要すらありません。`git add -p` や `git add --patch` をコマンドラインから実行すれば、同じ機能を呼び出せます。

また、このパッチモードを使って、ファイルの一部分だけをリセットすることもできます。その場合のコマンドは `reset --patch` です。同様に、部分的なチェックアウトは `checkout --patch` コマンドを、部分的に退避するなら `stash save --patch` コマンドを使います。各コマンドの詳細は、より高度な使い方に触れるときに併せて紹介します。

## 作業の隠しかたと消しかた

何らかのプロジェクトの一員として作業している場合にありがちなのですが、ある作業が中途半端な状態になっているときに、ブランチを切り替えてちょっとだけ別の作業をしたくなることがあります。 中途半端な状

態をコミットしてしまうのはいやなので、できればコミットせずにしておいて後でその状態から作業を再開したいものです。 そんなときに使うのが `git stash` コマンドです。

これは、作業ディレクトリのダーティな状態 (追跡しているファイルのうち変更されたもの、そしてステージされた変更) を受け取って未完了の作業をスタックに格納し、あとで好きなときに再度それを適用できるようにするものです。

## 自分の作業を隠す

例を見てみましょう。自分のプロジェクトでいくつかのファイルを編集し、その中のひとつをステージしたとします。 ここで `git status` を実行すると、ダーティな状態を確認することができます。

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

    modified:   index.html

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

    modified:   lib/simplegit.rb
```

ここで別のブランチに切り替えることになりましたが、現在の作業内容はまだコミットしたくありません。そこで、変更をいったん隠すことにします。 新たにスタックに隠すには `git stash` か `git stash save` を実行します。

```
$ git stash
Saved working directory and index state \
  "WIP on master: 049d078 added the index file"
HEAD is now at 049d078 added the index file
(To restore them type "git stash apply")
```

これで、作業ディレクトリはきれいな状態になりました。

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

これで、簡単にブランチを切り替えて別の作業をできるようになりました。これまでの変更内容はスタックに格納されています。 今までに格納した内容を見るには `git stash list` を使います。

```
$ git stash list
stash@{0}: WIP on master: 049d078 added the index file
stash@{1}: WIP on master: c264051 Revert "added file_size"
stash@{2}: WIP on master: 21d80a5 added number to log
```

この例では、以前にも二回ほど作業を隠していたようです。そこで、三種類の異なる作業にアクセスできるようになっています。 先ほど隠した変更を再度適用するには、stash コマンドの出力に書かれていたように `git stash apply` コマンドを実行します。 それよりもっと前に隠したものを適用したい場合は `git stash apply stash@{2}` のようにして名前を指定することもできます。 名前を指定しなければ、Git は直近に隠された変更を再適用します。

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

    modified:   index.html
    modified:   lib/simplegit.rb

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

このコマンドによって、さきほど隠したファイルが元に戻ったことがわかるでしょう。 今回は、作業ディレクトリがきれいな状態で変更を書き戻しました。また、変更を隠したときと同じブランチに書き戻しています。しかし、隠した内容を再適用するためにこれらが必須条件であるというわけではありません。 あるブランチの変更を隠し、別のブランチに移動して移動先のブランチにそれを書き戻すこともできます。また、隠した変更を書き戻す際に、現在のブランチに未コミットの変更があってもかまいません。もしうまく書き戻せなかった場合は、マージ時のコンフリクトと同じようになります。

さて、ファイルへの変更はもとどおりになりましたが、以前にステージしていたファイルはステージされていません。 これを行うには、`git stash apply` コマンドに `--index` オプションをつけて実行し、変更のステージ処理も再適用するよう指示しなければなりません。 先ほどのコマンドのかわりにこれを実行すると、元の状態に戻ります。

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

    modified:   index.html

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

    modified:   lib/simplegit.rb
```

apply オプションは、スタックに隠した作業を再度適用するだけで、スタックにはまだその作業が残ったままになります。 スタックから削除するには、`git stash drop` に削除したい作業の名前を指定して実行します。

```
$ git stash list
stash@{0}: WIP on master: 049d078 added the index file
stash@{1}: WIP on master: c264051 Revert "added file_size"
stash@{2}: WIP on master: 21d80a5 added number to log
$ git stash drop stash@{0}
Dropped stash@{0} (364e91f3f268f0900bc3ee613f9f733e82aaed43)
```

あるいは `git stash pop` を実行すれば、隠した内容を再適用してその後スタックからも削除してくれます。

### ファイルを隠す機能の応用

ファイルの隠しかたは何パターンかあり、役立つものがあるかもしれません。まずひとつ目、`stash save` コマンドの `--keep-index` オプションです。これはよく使われているオプションで、`git add` コマンドでインデックスに追加した内容を隠したくないときに用います。

あれこれと変更したうちの一部だけをコミットして、残りは後ほど処置したい場合、この機能が役立つでしょう。

```
$ git status -s
M  index.html
 M lib/simplegit.rb

$ git stash --keep-index
Saved working directory and index state WIP on master: 1b65b17 added the
index file
HEAD is now at 1b65b17 added the index file

$ git status -s
M  index.html
```

変更を隠すときの要望といえば、追跡しているファイルとそうでないファイルをどちらも一緒に隠してしまいたい、というのもあるでしょう。デフォルトでは、`git stash` コマンドが保存するのは追跡しているファイルだけです。けれど、`--include-untracked`（短縮形は `-u`）オプションを使うと、追跡していないファイルも一緒に保管して隠してくれます。

```
$ git status -s
M  index.html
 M lib/simplegit.rb
?? new-file.txt

$ git stash -u
Saved working directory and index state WIP on master: 1b65b17 added the
index file
HEAD is now at 1b65b17 added the index file

$ git status -s
$
```

もうひとつの応用例として、`--patch` オプションを挙げておきましょう。これを使うと、変更内容をすべて隠してしまうのではなく、隠したい変更を対話的に選択できるようになります。この場合、選択されなかった変更は作業ディレクトリに残ることになります。

```
$ git stash --patch
diff --git a/lib/simplegit.rb b/lib/simplegit.rb
index 66d332e..8bb5674 100644
--- a/lib/simplegit.rb
+++ b/lib/simplegit.rb
@@ -16,6 +16,10 @@ class SimpleGit
         return `#{git_cmd} 2>&1`.chomp
       end
     end
+
+    def show(treeish = 'master')
+      command("git show #{treeish}")
+    end

 end
 test
Stash this hunk [y,n,q,a,d,/,e,?]? y

Saved working directory and index state WIP on master: 1b65b17 added the
index file
```

### 隠した変更からのブランチの作成

作業をいったん隠し、しばらくそのブランチで作業を続けていると、隠した内容を再適用するときに問題が発生する可能性があります。 隠した後に何らかの変更をしたファイルに変更を再適用しようとすると、マージ時にコンフリクトが発生してそれを解決しなければならなくなるでしょう。 もう少しお手軽な方法で以前の作業を確認したい場合は `git stash branch` を実行します。このコマンドは、まず新しいブランチを作成し、作業をスタックに隠したときのコミットをチェックアウトし、スタックにある作業を再適用し、それに成功すればスタックからその作業を削除します。

```
$ git stash branch testchanges
M   index.html
M   lib/simplegit.rb
Switched to a new branch 'testchanges'
On branch testchanges
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   index.html

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

    modified:   lib/simplegit.rb

Dropped refs/stash@{0} (29d385a81d163dfd45a452a2ce816487a6b8b014)
```

これを使うと、保存していた作業をお手軽に復元して新しいブランチで作業をすることができます。

### 作業ディレクトリの掃除

最後に、作業ディレクトリにある変更内容やファイルを隠すのではなく、取り除いてしまいたい場合の話をしましょう。これは、`git clean` コマンドを使えば実現できます。

このコマンドが役立つのは、マージの結果、あるいは外部ツールによって生成された不要物を取り除いたり、ビルド結果を削除してクリーンな状態でビルドを実行したいときです。

このコマンドを実行するときは十分注意してください。作業ディレクトリにあって追跡されていないファイルは削除されるようになっているからです。後で気が変わっても、削除してしまったデータを取り戻すのは難しいでしょう。代わりに `git stash --all` を実行して、すべてを隠してしまうほうが安全です。

不要物を本当に取り除きたい、作業ディレクトリを掃除したい、という場合は、`git clean` を実行しましょう。作業ディレクトリの追跡されていないファイルをすべて削除するには、`git clean -f -d` を実行します。そうすれば、ファイルをすべて削除し、サブディレクトリを空にしてくれます。`-f` オプションは *force* の省略形で、「本当にそうしたい」という意味です。

このコマンドを実行するとどうなるか知りたいなら、`-n` オプションがいいでしょう。これを使うと、「リハーサルをして、何が消される *はず* だったのかを教えて」と Git に指示してくれます。

```
$ git clean -d -n
Would remove test.o
Would remove tmp/
```

`git clean` コマンドがデフォルトで削除するのは、追跡されていなくて、かつ無視されてもいないファイル

だけです。.gitignore
ファイルなどの無視設定に合致するファイルは削除されません。そういったファイルも消したい場合は、clean コマンドに -x オプションを追加するといいでしょう。完全にクリーンなビルドを行うため、以前のビルドで生成された .o ファイルをすべて削除したい、というような場合に使えます。

```
$ git status -s
 M lib/simplegit.rb
?? build.TMP
?? tmp/

$ git clean -n -d
Would remove build.TMP
Would remove tmp/

$ git clean -n -d -x
Would remove build.TMP
Would remove test.o
Would remove tmp/
```

git clean コマンドが何を削除するのかわからず不安なら、-n オプションを常につけるようにしましょう。何が削除されるかを前もって確認してから、 -n オプションを -f に変えてファイルを実際に削除すればよいのです。また、このコマンドを慎重に実行するもうひとつの方法として、-i、「対話モード」オプションというのもあります。

これを使えば、clean コマンドを対話モードで実行できます。

```
$ git clean -x -i
Would remove the following items:
  build.TMP  test.o
*** Commands ***
    1: clean                2: filter by pattern    3: select by numbers
4: ask each             5: quit
    6: help
What now>
```

この方法であれば、ファイルを個別に選んだり、パターンマッチさせるなど対話モードで範囲を絞り込んだうえでファイルを削除できます。

# 作業内容への署名

Git の仕組みは暗号学の点から見れば堅牢です。しかし、容易には得られません。インターネットを使って貢献を受け付けているとしましょう。受け付けた内容が信頼できる筋からのものかどうか調べたいときに、署名の付与・検証をGPG を使っておこなう複数の仕組みが Git にはあります。

## GPG とは

まずはじめに、何かを署名するには、GPG を設定し、個人鍵をインストールしなければなりません。

```
$ gpg --list-keys
/Users/schacon/.gnupg/pubring.gpg
---------------------------------
pub   2048R/0A46826A 2014-06-04
uid                  Scott Chacon (Git signing key) <schacon@gmail.com>
sub   2048R/874529A9 2014-06-04
```

鍵をインストールしていないのなら、`gpg --gen-key` を使って生成できます。

```
gpg --gen-key
```

署名付与用の秘密鍵ができたら、Git の設定項目 `user.signingkey` に鍵の内容を設定します。

```
git config --global user.signingkey 0A46826A
```

こうしておけば、タグやコミットに署名を付与するとき、Git はデフォルトでこの鍵を使うようになります。

## タグへの署名

GPG 秘密鍵の設定を終えていれば、その鍵を使ってタグの作成時に署名できます。 その場合は `-a` の代わりに `-s` を指定すればいいだけです。

```
$ git tag -s v1.5 -m 'my signed 1.5 tag'

You need a passphrase to unlock the secret key for
user: "Ben Straub <ben@straub.cc>"
2048-bit RSA key, ID 800430EB, created 2014-05-04
```

このタグに対して `git show` を実行すると、あなたの GPG 署名が表示されます。

```
$ git show v1.5
tag v1.5
Tagger: Ben Straub <ben@straub.cc>
Date:   Sat May 3 20:29:41 2014 -0700

my signed 1.5 tag
-----BEGIN PGP SIGNATURE-----
Version: GnuPG v1

iQEcBAABAgAGBQJTZbQlAAoJEF0+sviABDDrZbQH/09PfE51KPVPlanr6q1v4/Ut
LQxfojUWiLQdg2ESJItkcuweYg+kc3HCyFejeDIBw9dpXt00rY26p05qrpnG+85b
hM1/PswpPLuBSr+oCIDj5GMC2r2iEKsfv2fJbNW8iWAXVLoWZRF8B0MfqX/YTMbm
ecorc4iXzQu7tupRihslbNkfvfciMnSDeSvzCpWAHl7h8Wj6hhqePmLm9lAYqnKp
8S5B/1SSQuEAjRZgI4IexpZoeKGVDptPHxLLS38fozsyi0QyDyzEgJxcJQVMXxVi
RUysgqjcpT8+iQM1PblGfHR4XAhuOqN5Fx06PSaFZhqvWFezJ28/CLyX5q+oIVk=
=EFTF
-----END PGP SIGNATURE-----

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

    changed the version number
```

## タグの検証

署名付きのタグを検証するには `git tag -v [tag-name]` を使用します。 このコマンドは、GPG を使って署名を検証します。 これを正しく実行するには、署名者の公開鍵があなたの鍵リングに含まれている必要があります。

```
$ git tag -v v1.4.2.1
object 883653babd8ee7ea23e6a5c392bb739348b1eb61
type commit
tag v1.4.2.1
tagger Junio C Hamano <junkio@cox.net> 1158138501 -0700

GIT 1.4.2.1

Minor fixes since 1.4.2, including git-mv and git-http with alternates.
gpg: Signature made Wed Sep 13 02:08:25 2006 PDT using DSA key ID F3119B9A
gpg: Good signature from "Junio C Hamano <junkio@cox.net>"
gpg:                 aka "[jpeg image of size 1513]"
Primary key fingerprint: 3565 2A26 2040 E066 C9A7  4A7D C0C6 D9A4 F311
9B9A
```

署名者の公開鍵を持っていない場合は、このようなメッセージが表示されます。

```
gpg: Signature made Wed Sep 13 02:08:25 2006 PDT using DSA key ID F3119B9A
gpg: Can't check signature: public key not found
error: could not verify the tag 'v1.4.2.1'
```

### コミットへの署名

最近のバージョン（v1.7.9 以上）では、Git を使ってコミットに署名できるようになりました。 タグだけでなく、コミットにも署名したい場合は、`git commit` コマンドの `-S` オプションを使いましょう。

```
$ git commit -a -S -m 'signed commit'

You need a passphrase to unlock the secret key for
user: "Scott Chacon (Git signing key) <schacon@gmail.com>"
2048-bit RSA key, ID 0A46826A, created 2014-06-04

[master 5c3386c] signed commit
 4 files changed, 4 insertions(+), 24 deletions(-)
 rewrite Rakefile (100%)
 create mode 100644 lib/git.rb
```

また、署名の確認・検証を行うための `--show-signature` オプションが `git log` コマンドに用意されています。

```
$ git log --show-signature -1
commit 5c3386cf54bba0a33a32da706aa52bc0155503c2
gpg: Signature made Wed Jun  4 19:49:17 2014 PDT using RSA key ID 0A46826A
gpg: Good signature from "Scott Chacon (Git signing key)
<schacon@gmail.com>"
Author: Scott Chacon <schacon@gmail.com>
Date:   Wed Jun 4 19:49:17 2014 -0700

    signed commit
```

さらに、`git log` コマンドに署名の有無を出力させることもできます。書式設定で `%G?` を使いましょう。

```
$ git log --pretty="format:%h %G? %aN  %s"

5c3386c G Scott Chacon  signed commit
ca82a6d N Scott Chacon  changed the version number
085bb3b N Scott Chacon  removed unnecessary test code
a11bef0 N Scott Chacon  first commit
```

そうすれば、この例であれば最新のコミットのみが署名付き、しかもそれが有効であることがわかります。

バージョン 1.8.3 以降の Git であれば、マージやプルのときにコミットを拒否することもできます。`--verify-signatures` オプションを使うとコミットが検証され、有効な GPG 署名がない場合はマージやプルが拒否されます。

このオプションをブランチをマージするときに使うと、署名がない、もしくは有効でないコミットが含まれているブランチのマージは失敗します。

```
$ git merge --verify-signatures non-verify
fatal: Commit ab06180 does not have a GPG signature.
```

逆に、マージ対象のコミットすべてに有効な署名が施されていれば、検証された署名がすべて表示され、マージが実行に移されます。

```
$ git merge --verify-signatures signed-branch
Commit 13ad65e has a good GPG signature by Scott Chacon (Git signing key)
<schacon@gmail.com>
Updating 5c3386c..13ad65e
Fast-forward
 README | 2 ++
 1 file changed, 2 insertions(+)
```

また、`git merge` コマンドの `-S` オプションを使うと、マージコミットにも署名できます。以下のマージの例では、マージ対象コミットの署名を検証し、さらにマージコミットに署名を施しています。

```
$ git merge --verify-signatures -S  signed-branch
Commit 13ad65e has a good GPG signature by Scott Chacon (Git signing key)
<schacon@gmail.com>

You need a passphrase to unlock the secret key for
user: "Scott Chacon (Git signing key) <schacon@gmail.com>"
2048-bit RSA key, ID 0A46826A, created 2014-06-04

Merge made by the 'recursive' strategy.
 README | 2 ++
 1 file changed, 2 insertions(+)
```

### 署名付与は全員で

タグやコミットに署名を付与するのは素晴らしい試みです。ただし、作業手順のひとつとして採用するのであれば、メンバー全員がやり方を知っているかどうか前もって確認しておくべきでしょう。そうしておかないと、作成済みコミットに署名を付与する方法を説明してまわるハメになりかねません。GPG の仕組み、署名を付与することのメリットをよく理解してから、作業手順に組み込むようにしましょう。

# 検索

コード量の大小を問わず、関数の参照位置・定義やメソッドの変更履歴を確認したくなることはよくあります。Git には便利なツールがいくつも用意されていて、コードやコミット履歴の確認が簡単にできるようになっています。具体的な方法をいくつか見ていきましょう。

## Git Grep

Git に付属する `grep` コマンドを使うと、コミット済みのツリーや作業ディレクトリが簡単に検索（文字列・正規表現）できます。使い方の説明を兼ねて、Git のソースコードを覗いてみることにしましょう。

このコマンドはデフォルトでは作業ディレクトリを検索します。`-n` オプションと一緒に使うと、検索条件とマッチした行の番号も表示してくれます。

```
$ git grep -n gmtime_r
compat/gmtime.c:3:#undef gmtime_r
compat/gmtime.c:8:      return git_gmtime_r(timep, &result);
compat/gmtime.c:11:struct tm *git_gmtime_r(const time_t *timep, struct tm
*result)
compat/gmtime.c:16:     ret = gmtime_r(timep, result);
compat/mingw.c:606:struct tm *gmtime_r(const time_t *timep, struct tm
*result)
compat/mingw.h:162:struct tm *gmtime_r(const time_t *timep, struct tm
*result);
date.c:429:             if (gmtime_r(&now, &now_tm))
date.c:492:             if (gmtime_r(&time, tm)) {
git-compat-util.h:721:struct tm *git_gmtime_r(const time_t *, struct tm
*);
git-compat-util.h:723:#define gmtime_r git_gmtime_r
```

その他にも、興味深いオプションがこのコマンドにはいくつも用意されています。

上記の実行例とは違い、コマンド出力を Git に要約させることもできます。例えば、検索にマッチしたファイルの名前とマッチ回数を表示させるには、`--count` オプションを使います。

```
$ git grep --count gmtime_r
compat/gmtime.c:4
compat/mingw.c:1
compat/mingw.h:1
date.c:2
git-compat-util.h:2
```

検索にマッチした結果からメソッドや関数と思われるものだけを確認したい場合は、`-p` オプションを使いましょう。

```
$ git grep -p gmtime_r *.c
date.c=static int match_multi_number(unsigned long num, char c, const char
*date, char *end, struct tm *tm)
date.c:         if (gmtime_r(&now, &now_tm))
date.c=static int match_digit(const char *date, struct tm *tm, int
*offset, int *tm_gmt)
date.c:         if (gmtime_r(&time, tm)) {
```

この例では、gmtime_r が date.c ファイルにある関数 match_multi_number と match_digit から呼び出されていることがわかります。

また、文字列の複雑な組み合わせを探したい場合は --and オプションを使いましょう。検索条件がすべて同一行に含まれている行だけを返すためのオプションです。例として、文字列 “LINK” か “BUF_MAX” を含む定数が記述されている行を、Git の古いバージョン 1.8.0 から探してみます。

なお、この例では --break と --heading のオプションも使っています。出力を分割して読みやすくするためです。

```
$ git grep --break --heading \
    -n -e '#define' --and \( -e LINK -e BUF_MAX \) v1.8.0
v1.8.0:builtin/index-pack.c
62:#define FLAG_LINK (1u<<20)

v1.8.0:cache.h
73:#define S_IFGITLINK  0160000
74:#define S_ISGITLINK(m)       (((m) & S_IFMT) == S_IFGITLINK)

v1.8.0:environment.c
54:#define OBJECT_CREATION_MODE OBJECT_CREATION_USES_HARDLINKS

v1.8.0:strbuf.c
326:#define STRBUF_MAXLINK (2*PATH_MAX)

v1.8.0:symlinks.c
53:#define FL_SYMLINK  (1 << 2)

v1.8.0:zlib.c
30:/* #define ZLIB_BUF_MAX ((uInt)-1) */
31:#define ZLIB_BUF_MAX ((uInt) 1024 * 1024 * 1024) /* 1GB */
```

grep や ack のような他の検索用コマンドと比較すると、git grep コマンドには利点がふたつあります。とても早く動作することと、作業ディレクトリだけでなくコミット済みの全ツリーが検索対象であることです。上記の例ではその利点を示すために、検索対象を古いバージョンの Git のソースコードとし、チェックアウトされたバージョンのものにはしませんでした。

## Git ログの検索

場合によっては、探しているのは語句の **所在** ではなく、語句が存在した・追加された **時期**、ということもあるでしょう。`git log` コマンドの強力なオプションを使うと、コミットメッセージの内容やコミットごとの差分をもとに、特定のコミットを絞り込めます。

ここでは、定数 `ZLIB_BUF_MAX` が追加された時期を調べてみましょう。その文字列が追加、あるいは削除されたコミットだけを表示するには、`-S` オプションを用います。

```
$ git log -SZLIB_BUF_MAX --oneline
e01503b zlib: allow feeding more than 4GB in one go
ef49a7a zlib: zlib can only process 4GB at a time
```

これらのコミットの差分を見てみると、コミット `ef49a7a` でこの定数が追加され、コミット `e01503b` でそれが変更されたことがわかります。

より詳しく調べたいのなら、`-G` オプションをつけましょう。検索に正規表現が使えるようになります。

### ログの行指向検索

一歩進んだログ検索の方法をもうひとつ見ておきましょう。履歴を行指向で検索するという、ものすごく便利な方法です。最近になって Git に追加された機能であまり知られていませんが、本当に便利です。`git log` コマンドに `-L` オプションをつけると行指向検索が有効になり、指定した行（関数など）の履歴を確認できます。

ここでは仮に、`zlib.c` ファイルにある `git_deflate_bound` 関数の変更履歴を確認したいとしましょう。用いるコマンドは `git log -L :git_deflate_bound:zlib.c` です。これを実行すると、指定された関数の定義範囲がまずは推測されます。そして、その範囲の全変更履歴をパッチの形でひとつずつ、関数が追加されたときの履歴にまでさかのぼって表示します。

```
$ git log -L :git_deflate_bound:zlib.c
commit ef49a7a0126d64359c974b4b3b71d7ad42ee3bca
Author: Junio C Hamano <gitster@pobox.com>
Date:   Fri Jun 10 11:52:15 2011 -0700

    zlib: zlib can only process 4GB at a time

diff --git a/zlib.c b/zlib.c
--- a/zlib.c
+++ b/zlib.c
@@ -85,5 +130,5 @@
-unsigned long git_deflate_bound(z_streamp strm, unsigned long size)
+unsigned long git_deflate_bound(git_zstream *strm, unsigned long size)
 {
-       return deflateBound(strm, size);
+       return deflateBound(&strm->z, size);
 }


commit 225a6f1068f71723a910e8565db4e252b3ca21fa
Author: Junio C Hamano <gitster@pobox.com>
Date:   Fri Jun 10 11:18:17 2011 -0700

    zlib: wrap deflateBound() too

diff --git a/zlib.c b/zlib.c
--- a/zlib.c
+++ b/zlib.c
@@ -81,0 +85,5 @@
+unsigned long git_deflate_bound(z_streamp strm, unsigned long size)
+{
+       return deflateBound(strm, size);
+}
+
```

検索対象のコードで用いられているプログラミング言語によっては、 Git が関数やメソッドの定義範囲を絞り込めないことがあります。そんな場合は、正規表現を使いましょう。上記の例でいえば `git log -L '/unsigned long git_deflate_bound/',/^}/:zlib.c` はまったく同じ結果を出力します。また、行番号で検索対象を指定（単一行の指定、複数行で範囲指定の両方が可能）しても、同じような結果が得られます。

## 歴史の書き換え

Git を使って作業をしていると、何らかの理由でコミットの歴史を書き換えたくなることが多々あります。 Git のすばらしい点のひとつは、何をどうするかの決断をぎりぎりまで先送りできることです。 どのファイルをどのコミットに含めるのかは、ステージングエリアの内容をコミットする直前まで変更することができますし、既に作業した内容でも stash コマンドを使えばまだ作業していないことにできます。また、すでにコミッ

トしてしまった変更についても、それを書き換えてまるで別の方法で行ったかのようにすることもできます。コミットの順序を変更したり、コミットメッセージやコミットされるファイルを変更したり、複数のコミットをひとつにまとめたりひとつのコミットを複数に分割したり、コミットそのものをなかったことにしたり……といった作業を、変更内容を他のメンバーに公開する前ならいつでもすることができます。

このセクションでは、これらの便利な作業の方法について扱います。これで、あなたのコミットの歴史を思い通りに書き換えてから他の人と共有できるようになります。

## 直近のコミットの変更

直近のコミットを変更するというのは、歴史を書き換える作業のうちもっともよくあるものでしょう。 直近のコミットに対して手を加えるパターンとしては、コミットメッセージを変更したりそのコミットで記録されるスナップショットを変更 (ファイルを追加・変更あるいは削除) したりといったものがあります。

単に直近のコミットメッセージを変更したいだけの場合は非常にシンプルです。

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

これを実行するとテキストエディタが開きます。すでに直近のコミットメッセージが書き込まれた状態になっており、それを変更することができます。 変更を保存してエディタを終了すると、変更後のメッセージを含む新しいコミットを作成して直近のコミットをそれで置き換えます。

いったんコミットしたあとで、そこにさらにファイルを追加したり変更したりしたくなったとしましょう。「新しく作ったファイルを追加し忘れた」とかがありそうですね。この場合の手順も基本的には同じです。ファイルを編集して `git add` したり追跡中のファイルを `git rm` したりしてステージングエリアをお好みの状態にしたら、続いて `git commit --amend` を実行します。すると、現在のステージングエリアの状態を次回のコミット用のスナップショットにします。

この技を使う際には注意が必要です。この処理を行うとコミットの SHA-1 が変わるからです。 いわば、非常に小規模なリベースのようなものです。すでにプッシュしているコミットは書き換えないようにしましょう。

## 複数のコミットメッセージの変更

さらに歴史をさかのぼったコミットを変更したい場合は、もう少し複雑なツールを使わなければなりません。Git には歴史を修正するツールはありませんが、リベースツールを使って一連のコミットを (別の場所ではなく) もともとあった場所と同じ HEAD につなげるという方法を使うことができます。 対話的なリベースツールを使えば、各コミットについてメッセージを変更したりファイルを追加したりお望みの変更をすることができます。 対話的なリベースを行うには、`git rebase` に `-i` オプションを追加します。 どこまでさかのぼってコミットを書き換えるかを指示するために、どのコミットにリベースするかを指定しなければなりません。

直近の三つのコミットメッセージあるいはそのいずれかを変更したくなった場合、変更したい最古のコミットの親を `git rebase -i` の引数に指定します。ここでは `HEAD~2^` あるいは `HEAD~3` となります。 直近の三つのコミットを編集しようと考えているのだから、`~3` のほうが覚えやすいでしょう。しかし、実際のところは四つ前 (変更したい最古のコミットの親) のコミットを指定していることに注意しましょう。

```
$ git rebase -i HEAD~3
```

これはリベースコマンドであることを認識しておきましょう。 HEAD~3..HEAD に含まれるすべてのコミットは、実際にメッセージを変更したか否かにかかわらずすべて書き換えられます。 すでに中央サーバーにプッシュしたコミットをここに含めてはいけません。含めてしまうと、同じ変更が別のバージョンで見えてしまうことになって他の開発者が混乱します。

このコマンドを実行すると、テキストエディタが開いてコミットの一覧が表示され、このようになります。

```
pick f7f3f6d changed my name a bit
pick 310154e updated README formatting and added blame
pick a5f4a0d added cat-file

# Rebase 710f0f8..a5f4a0d onto 710f0f8
#
# Commands:
#  p, pick = use commit
#  r, reword = use commit, but edit the commit message
#  e, edit = use commit, but stop for amending
#  s, squash = use commit, but meld into previous commit
#  f, fixup = like "squash", but discard this commit's log message
#  x, exec = run command (the rest of the line) using shell
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#
# Note that empty commits are commented out
```

このコミット一覧の表示順は、log コマンドを使ったときの通常の表示順とは逆になることに注意しましょう。 log を実行すると、このようになります。

```
$ git log --pretty=format:"%h %s" HEAD~3..HEAD
a5f4a0d added cat-file
310154e updated README formatting and added blame
f7f3f6d changed my name a bit
```

逆順になっていますね。 対話的なリベースを実行するとスクリプトが出力されるので、それをあとで実行することになります。 このスクリプトはコマンドラインで指定したコミット (HEAD~3) から始まり、それ以降のコミットを古い順に再現していきます。 最新のものからではなく古いものから表示されているのは、最初に再現するのがいちばん古いコミットだからです。

このスクリプトを編集し、手を加えたいコミットのところでスクリプトを停止させるようにします。そのためには、各コミットのうちスクリプトを停止させたいものについて「pick」を「edit」に変更します。たとえば、三番目のコミットメッセージだけを変更したい場合はこのようにファイルを変更します。

```
edit f7f3f6d changed my name a bit
pick 310154e updated README formatting and added blame
pick a5f4a0d added cat-file
```

これを保存してエディタを終了すると、Git はそのリストの最初のコミットまで処理を巻き戻し、次のようなメッセージとともにコマンドラインを返します。

```
$ git rebase -i HEAD~3
Stopped at f7f3f6d... changed my name a bit
You can amend the commit now, with

       git commit --amend

Once you’re satisfied with your changes, run

       git rebase --continue
```

この指示が、まさにこれからすべきことを教えてくれています。

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

と打ち込んでコミットメッセージを変更してからエディタを終了し、次に

```
$ git rebase --continue
```

を実行します。このコマンドはその他のふたつのコミットも自動的に適用するので、これで作業は終了です。複数行で「pick」を「edit」に変更した場合は、これらの作業を各コミットについてくりかえすことになります。 それぞれの場面で Git が停止するので、amend でコミットを書き換えて continue で処理を続けます。

## コミットの並べ替え

対話的なリベースで、コミットの順番を変更したり完全に消し去ってしまったりすることもできます。“added cat-file” のコミットを削除して残りの二つのコミットの適用順を反対にしたい場合は、リベーススクリプトを

```
pick f7f3f6d changed my name a bit
pick 310154e updated README formatting and added blame
pick a5f4a0d added cat-file
```

から

```
pick 310154e updated README formatting and added blame
pick f7f3f6d changed my name a bit
```

のように変更します。これを保存してエディタを終了すると、Git はまずこれらのコミットの親までブランチを巻き戻してから `310154e` を適用し、その次に `f7f3f6d` を適用して停止します。 これで、効率的にコミット順を変更して “added cat-file” のコミットは完全に取り除くことができました。

## コミットのまとめ

一連のコミット群をひとつのコミットにまとめて押し込んでしまうことも、対話的なリベースツールで行うことができます。リベースメッセージの中に、その手順が出力されています。

```
#
# Commands:
#  p, pick = use commit
#  r, reword = use commit, but edit the commit message
#  e, edit = use commit, but stop for amending
#  s, squash = use commit, but meld into previous commit
#  f, fixup = like "squash", but discard this commit's log message
#  x, exec = run command (the rest of the line) using shell
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#
# Note that empty commits are commented out
```

「pick」や「edit」のかわりに「squash」を指定すると、Git はその変更と直前の変更をひとつにまとめて新たなコミットメッセージを書き込めるようにします。 つまり、これらの三つのコミットをひとつのコミットにまとめたい場合は、スクリプトをこのように変更します。

```
pick f7f3f6d changed my name a bit
squash 310154e updated README formatting and added blame
squash a5f4a0d added cat-file
```

これを保存してエディタを終了すると、Git は三つの変更をすべて適用してからエディタに戻るので、そこでコミットメッセージを変更します。

```
# This is a combination of 3 commits.
# The first commit's message is:
changed my name a bit

# This is the 2nd commit message:

updated README formatting and added blame

# This is the 3rd commit message:

added cat-file
```

これを保存すると、さきほどの三つのコミットの内容をすべて含んだひとつのコミットができあがります。

## コミットの分割

コミットの分割は、いったんコミットを取り消してから部分的なステージとコミットを繰り返して行います。たとえば、先ほどの三つのコミットのうち真ん中のものを分割することになったとしましょう。“updated README formatting and added blame” のコミットを、“updated README formatting” と “added blame” のふたつに分割します。 そのためには、`rebase -i` スクリプトを実行してそのコミットの指示を「edit」に変更します。

```
pick f7f3f6d changed my name a bit
edit 310154e updated README formatting and added blame
pick a5f4a0d added cat-file
```

続いて、コマンドラインに戻ってコミットをリセットし、その内容を使ってコミットを複数に分割していきます。 まず、変更を保存してエディタを終了すると、Git はリストの最初のコミットの親まで処理を巻き戻します。そして最初のコミット (`f7f3f6d`) と二番目のコミット (`310154e`) を適用してからコンソールに戻ります。 コミットをリセットするには `git reset HEAD^` を実行します。これはコミット自体を取り消し、変更されたファイルはステージしていない状態にします。 そして、その状態から一連のコミットを作ったら、以下のように`git rebase --continue` を実行しましょう。

```
$ git reset HEAD^
$ git add README
$ git commit -m 'updated README formatting'
$ git add lib/simplegit.rb
$ git commit -m 'added blame'
$ git rebase --continue
```

Git はスクリプトの最後のコミット (`a5f4a0d`) を適用し、歴史はこのようになります。

```
$ git log -4 --pretty=format:"%h %s"
1c002dd added cat-file
9b29157 added blame
35cfb2b updated README formatting
f3cc40e changed my name a bit
```

念のためにもう一度言いますが、この変更はリスト内のすべてのコミットの SHA を変更します。すでに共有リポジトリにプッシュしたコミットは、このリストに表示させないようにしましょう。

## 最強のオプション: filter-branch

歴史を書き換える方法がもうひとつあります。これは、大量のコミットの書き換えを機械的に行いたい場合 (メールアドレスを一括変更したりすべてのコミットからあるファイルを削除したりなど) に使うものです。そのためのコマンドが `filter-branch` です。これは歴史を大規模にばさっと書き換えることができるものなので、プロジェクトを一般に公開した後や書き換え対象のコミットを元にしてだれかが作業を始めている場合はまず使うことはありません。 しかし、これは非常に便利なものでもあります。 一般的な使用例をいくつか説明するので、それをもとにこの機能を使いこなせる場面を考えてみましょう。

### 全コミットからのファイルの削除

これは、相当よくあることでしょう。 誰かが不注意で `git add .` をした結果、巨大なバイナリファイルが間違えてコミットされてしまったとしましょう。これを何とか削除してしまいたいものです。 あるいは、間違ってパスワードを含むファイルをコミットしてしまったとしましょう。このプロジェクトをオープンソースにしたいと思ったときに困ります。 `filter-branch` は、こんな場合に歴史全体を洗うために使うツールです。 `passwords.txt` というファイルを歴史から完全に抹殺してしまうには、`filter-branch` の `--tree-filter` オプションを使います。

```
$ git filter-branch --tree-filter 'rm -f passwords.txt' HEAD
Rewrite 6b9b3cf04e7c5686a9cb838c3f36a8cb6a0fc2bd (21/21)
Ref 'refs/heads/master' was rewritten
```

`--tree-filter` オプションは、プロジェクトの各チェックアウトに対して指定したコマンドを実行し、結果を再コミットします。 この場合は、すべてのスナップショットから `passwords.txt` というファイルを削除します。 間違えてコミットしてしまったエディタのバックアップファイルを削除するには、`git filter-branch --tree-filter 'rm -f *~' HEAD` のように実行します。

Git がツリーを書き換えてコミットし、ブランチのポインタを末尾に移動させる様子がごらんいただけるでしょう。 この作業は、まずはテスト用ブランチで実行してから結果をよく吟味し、それから master ブランチに適用することをおすすめします。 `filter-branch` をすべてのブランチで実行するには、このコマンドに `--all` を渡します。

### サブディレクトリを新たなルートへ

別のソース管理システムからのインポートを終えた後、無意味なサブディレクトリ (`trunk`、`tags`など) が残っている状態を想定しましょう。 すべてのコミットの `trunk` ディレクトリを新たなプロジェクトルート

としたい場合にも、filter-branch が助けになります。

```
$ git filter-branch --subdirectory-filter trunk HEAD
Rewrite 856f0bf61e41a27326cdae8f09fe708d679f596f (12/12)
Ref 'refs/heads/master' was rewritten
```

これで、新たなプロジェクトルートはそれまで trunk ディレクトリだった場所になります。 Git は、このサブディレクトリに影響を及ぼさないコミットを自動的に削除します。

#### メールアドレスの一括変更

もうひとつよくある例としては、「作業を始める前に git config で名前とメールアドレスを設定することを忘れていた」とか「業務で開発したプロジェクトをオープンソースにするにあたって、職場のメールアドレスをすべて個人アドレスに変更したい」などがあります。 どちらの場合についても、複数のコミットのメールアドレスを一括で変更することになりますが、これも filter-branch でることができます。 注意して、あなたのメールアドレスのみを変更しなければなりません。そこで、--commit-filter を使います。

```
$ git filter-branch --commit-filter '
        if [ "$GIT_AUTHOR_EMAIL" = "schacon@localhost" ];
        then
                GIT_AUTHOR_NAME="Scott Chacon";
                GIT_AUTHOR_EMAIL="schacon@example.com";
                git commit-tree "$@";
        else
                git commit-tree "$@";
        fi' HEAD
```

これで、すべてのコミットであなたのアドレスを新しいものに書き換えます。 コミットにはその親の SHA-1 値が含まれるので、このコマンドは (マッチするメールアドレスが存在するものだけではなく) すべてのコミットを書き換えます。

# リセットコマンド詳説

専門的なツールを説明する前に、reset と checkout について触れておきます。 いざ使うことになると、一番ややこしい部類の Git コマンドです。 出来ることがあまりに多くて、ちゃんと理解したうえで正しく用いることなど夢のまた夢のようにも思えてしまいます。 よって、ここでは単純な例えを使って説明していきます。

## 3つのツリー

reset と checkout を単純化したいので、Git を「3つのツリーのデータを管理するためのツール」と捉えてしまいましょう。 なお、ここでいう「ツリー」とはあくまで「ファイルの集まり」であって、データ構造は含みません。 （Git のインデックスがツリーとは思えないようなケースもありますが、ここでは単純にするため、「ツリー＝ファイルの集まり」で通していきます。）

いつものように Git を使っていくと、以下のツリーを管理・操作していくことになります。

| ツリー | 役割 |
|---|---|
| HEAD | 最新コミットのスナップショットで、次は親になる |
| インデックス | 次のコミット候補のスナップショット |
| 作業ディレクトリ | サンドボックス |

### HEAD

現在のブランチを指し示すポインタは HEAD と呼ばれています。HEAD は、そのブランチの最新コミットを指し示すポインタでもあります。 ということは、HEAD が指し示すコミットは新たに追加されていくコミットの親になる、ということです。 HEAD のことを **最新のコミット** のスナップショットと捉えておくとわかりやすいでしょう。

では、スナップショットの内容を確認してみましょう。実に簡単です。 ディレクトリ構成と SHA-1 チェックサムを HEAD のスナップショットから取得するには、以下のコマンドを実行します。

```
$ git cat-file -p HEAD
tree cfda3bf379e4f8dba8717dee55aab78aef7f4daf
author Scott Chacon  1301511835 -0700
committer Scott Chacon  1301511835 -0700

initial commit

$ git ls-tree -r HEAD
100644 blob a906cb2a4a904a152...   README
100644 blob 8f94139338f9404f2...   Rakefile
040000 tree 99f1a6d12cb4b6f19...   lib
```

`cat-file` や `ls-tree` は「配管」コマンドなので、日々の作業で使うことはないはずでしょう。ただし、今回のように詳細を把握するには便利です。

### インデックス

インデックスとは、**次のコミット候補** のことを指します。Git の「ステージングエリア」と呼ばれることもあります。`git commit` を実行すると確認される内容だからです。

インデックスの中身は、前回のチェックアウトで作業ディレクトリに保存されたファイルの一覧になっています。保存時のファイルの状態も記録されています。 ファイルに変更を加え、`git commit` コマンドを実行すると、ツリーが作成され新たなコミットとなります。

```
$ git ls-files -s
100644 a906cb2a4a904a152e80877d4088654daad0c859 0   README
100644 8f94139338f9404f26296befa88755fc2598c289 0   Rakefile
100644 47c6340d6459e05787f644c2447d2595f5d3a54b 0   lib/simplegit.rb
```

この例で使った `ls-files` コマンドも縁の下の力持ち的なコマンドです。インデックスの状態を表示してくれます。

なお、インデックスは厳密にはツリー構造ではありません。実際には、階層のない構造になっています。ただ、理解する上ではツリー構造と捉えて差し支えありません。

#### 作業ディレクトリ

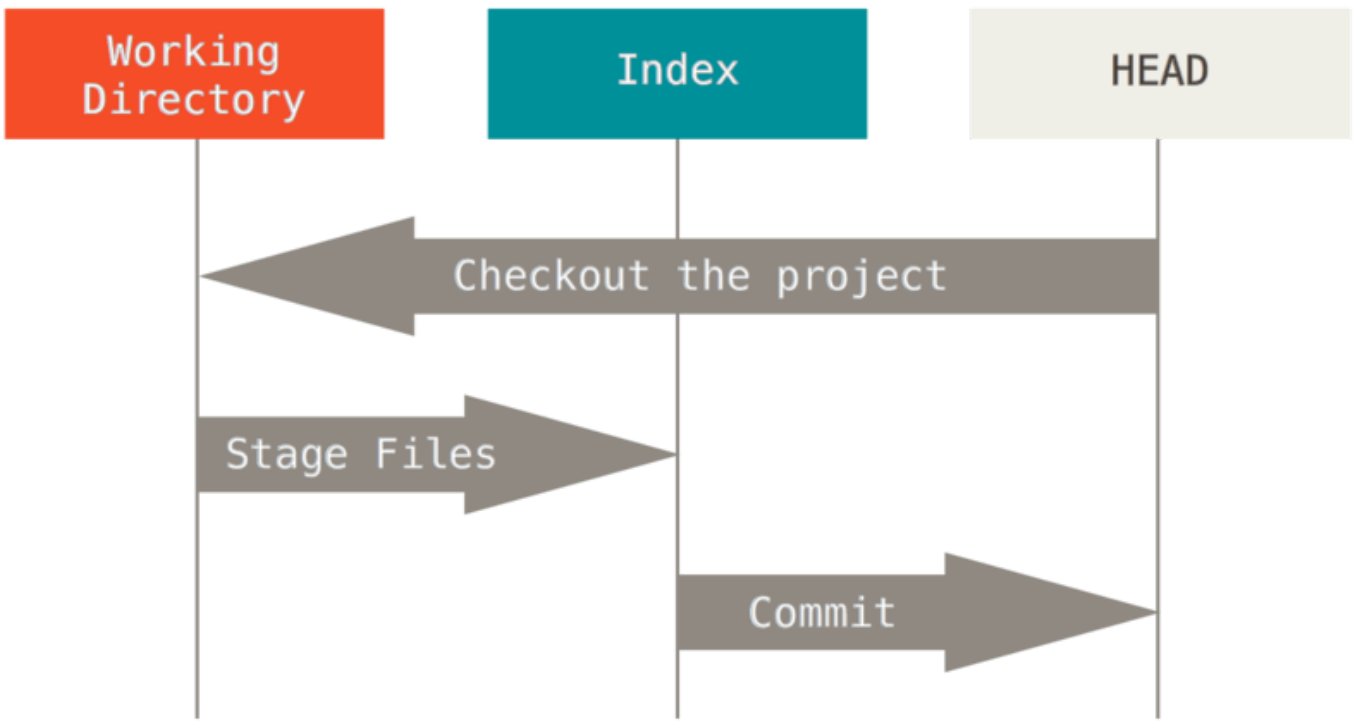
3つのツリーの最後は作業ディレクトリです。 他のツリーは、データを `.git` ディレクトリ内に処理しやすい形で格納してしまうため、人間が取り扱うには不便でした。 一方、作業ディレクトリにはデータが実際のファイルとして展開されます。とても取り扱いやすい形です。 作業ディレクトリのことは **サンドボックス** だと思っておいてください。そこでは、自由に変更を試せます。変更が完了したらステージングエリア（インデックス）に追加し、さらにコミットして歴史に追加するのです。

```
$ tree
.
├── README
├── Rakefile
└── lib
    └── simplegit.rb

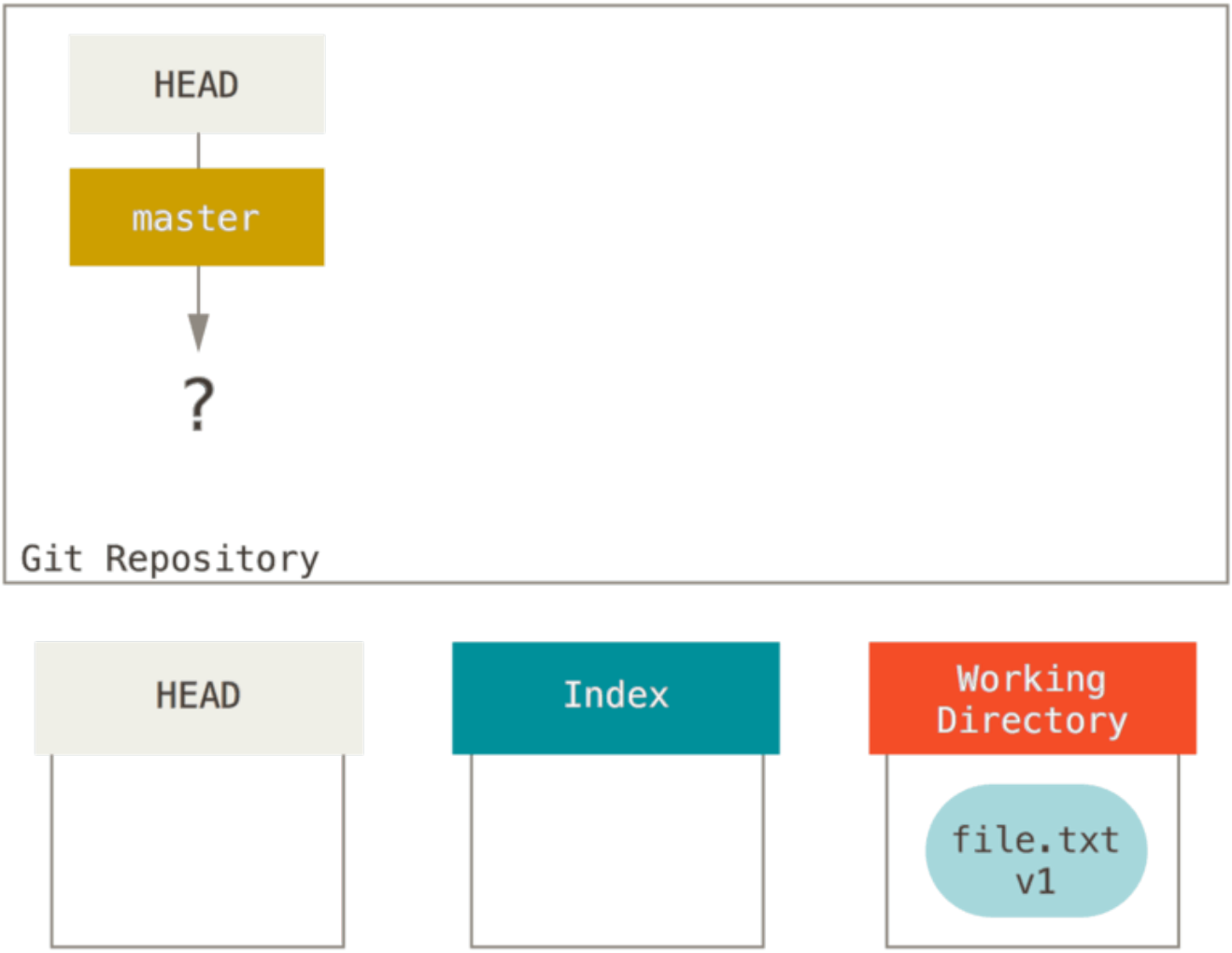
1 directory, 3 files
```

### 作業手順

Git を使う主目的は、プロジェクトのスナップショットを健全な状態で取り続けることです。そのためには、3つのツリーを操作する必要があります。

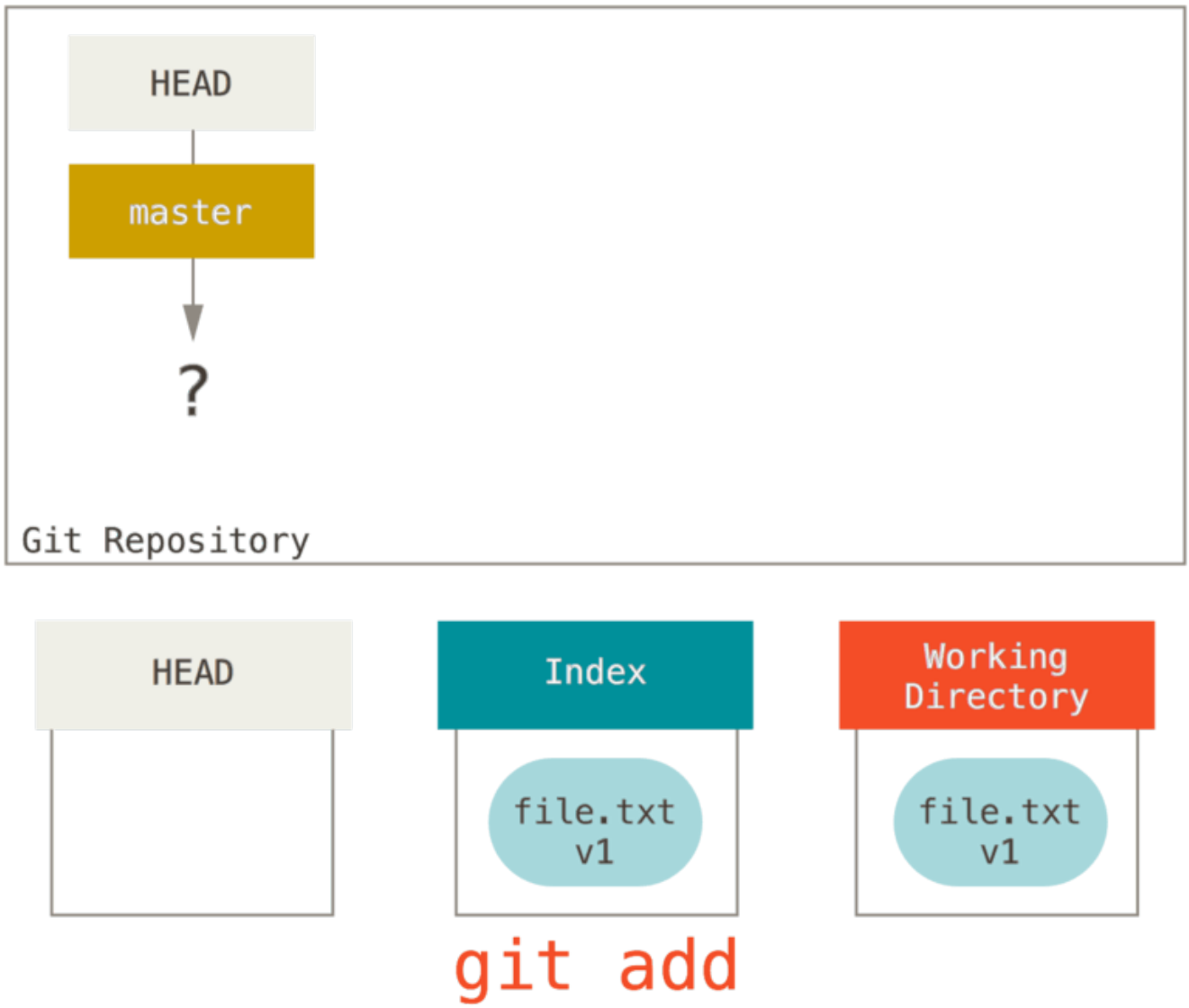


その手順を頭を使って説明しましょう。まず、新しいディレクトリを作って、テキストファイルをひとつ保存したとします。 現段階でのこのファイルを **v1** としましょう（図では青塗りの部分）。 次に `git init` を実行して Git リポジトリを生成します。このときの HEAD は、これから生成される予定のブランチを指し示すことになります（`master` はまだ存在しません）。

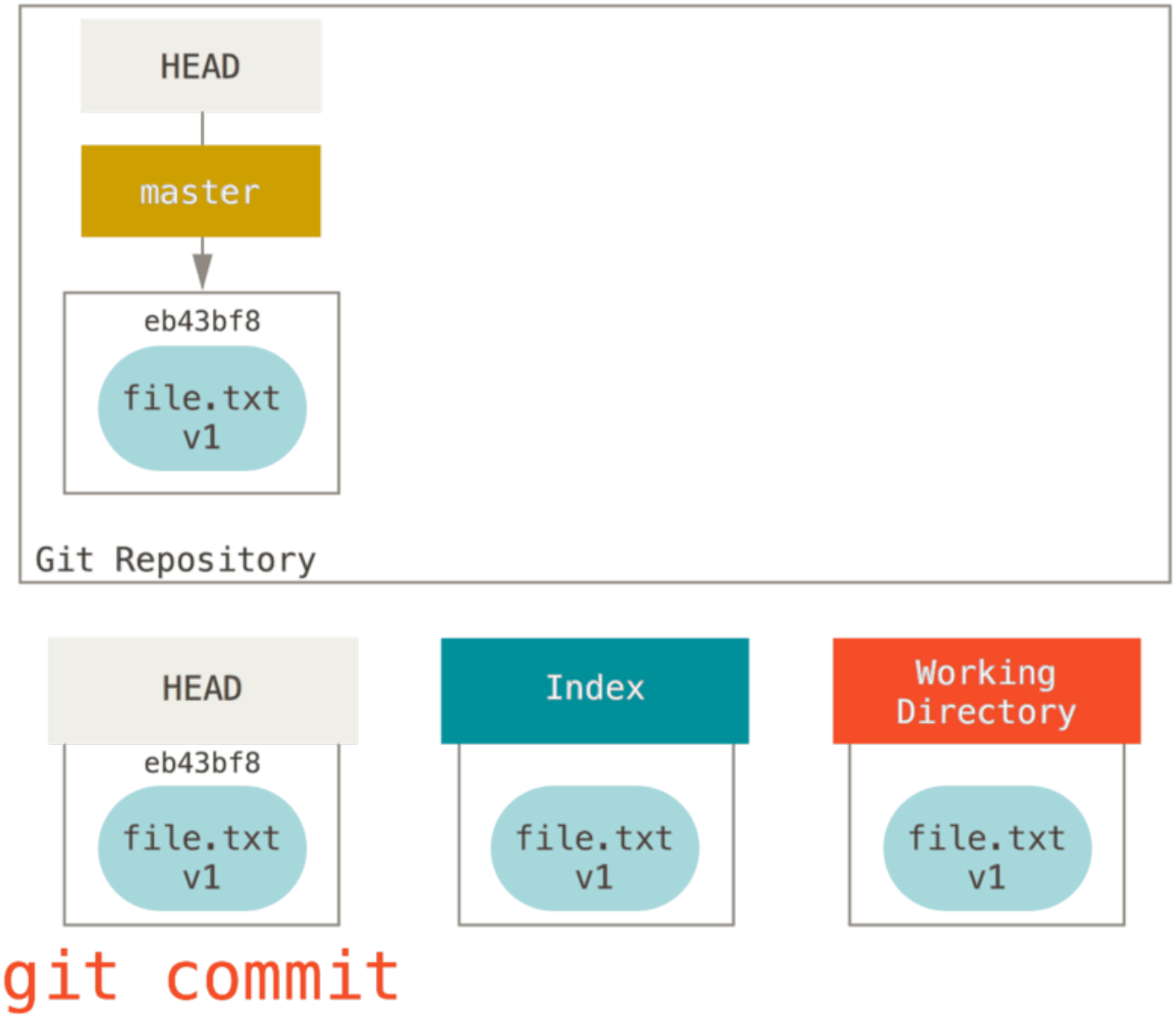



この時点では、作業ディレクトリにしかテキストファイルのデータは存在しません。

では、このファイルをコミットしてみましょう。まずは `git add` を実行して、作業ディレクトリ上のデータをインデックスにコピーします。

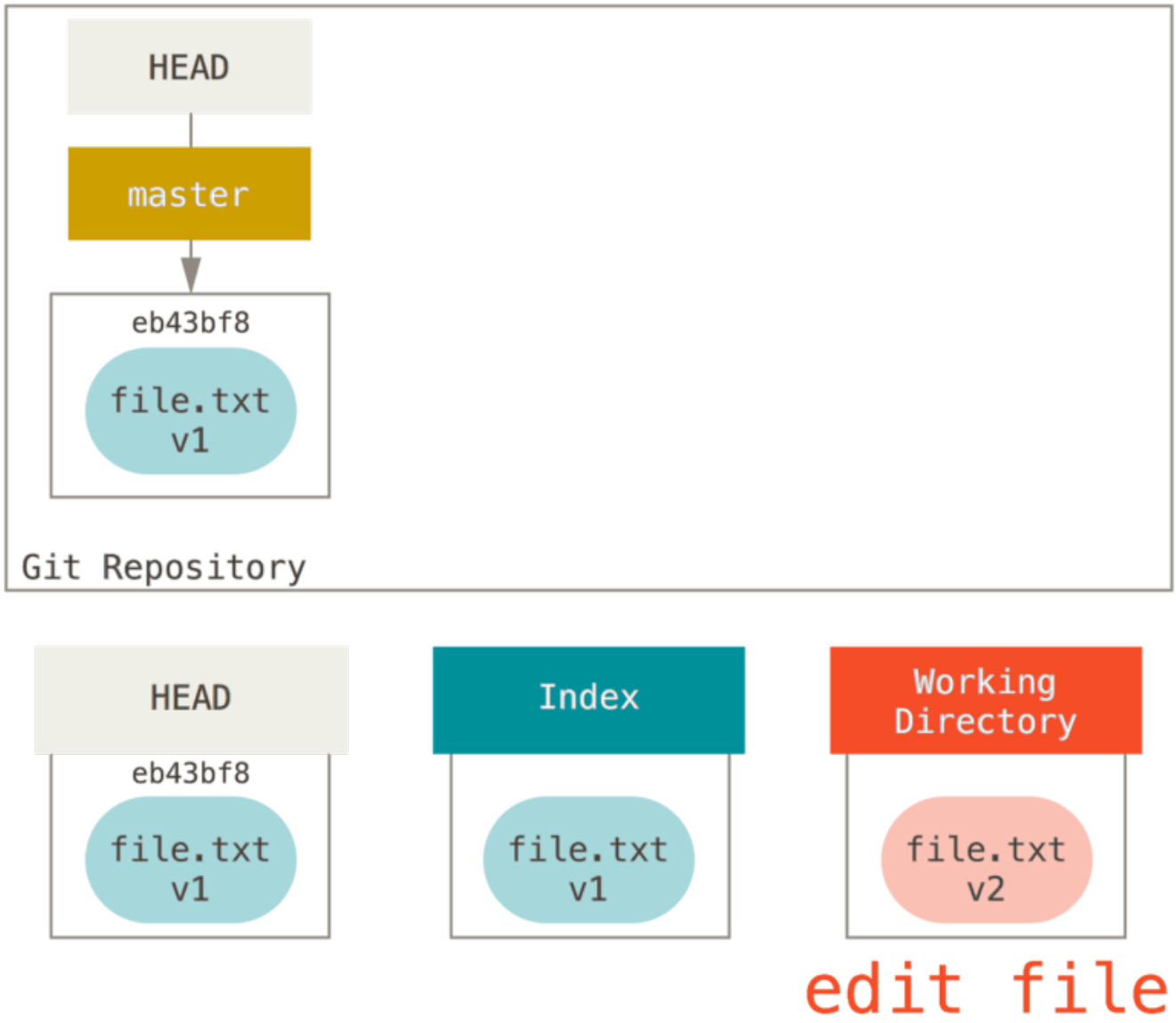



さらに、`git commit` を実行し、インデックスの内容でスナップショットを作成します。そうすると、作成したスナップショットをもとにコミットオブジェクトが作成され、`master` がそのコミットを指し示すようになります。




この段階で `git status` を実行しても、何も変更点は出てきません。3つのツリーが同じ状態になっているからです。

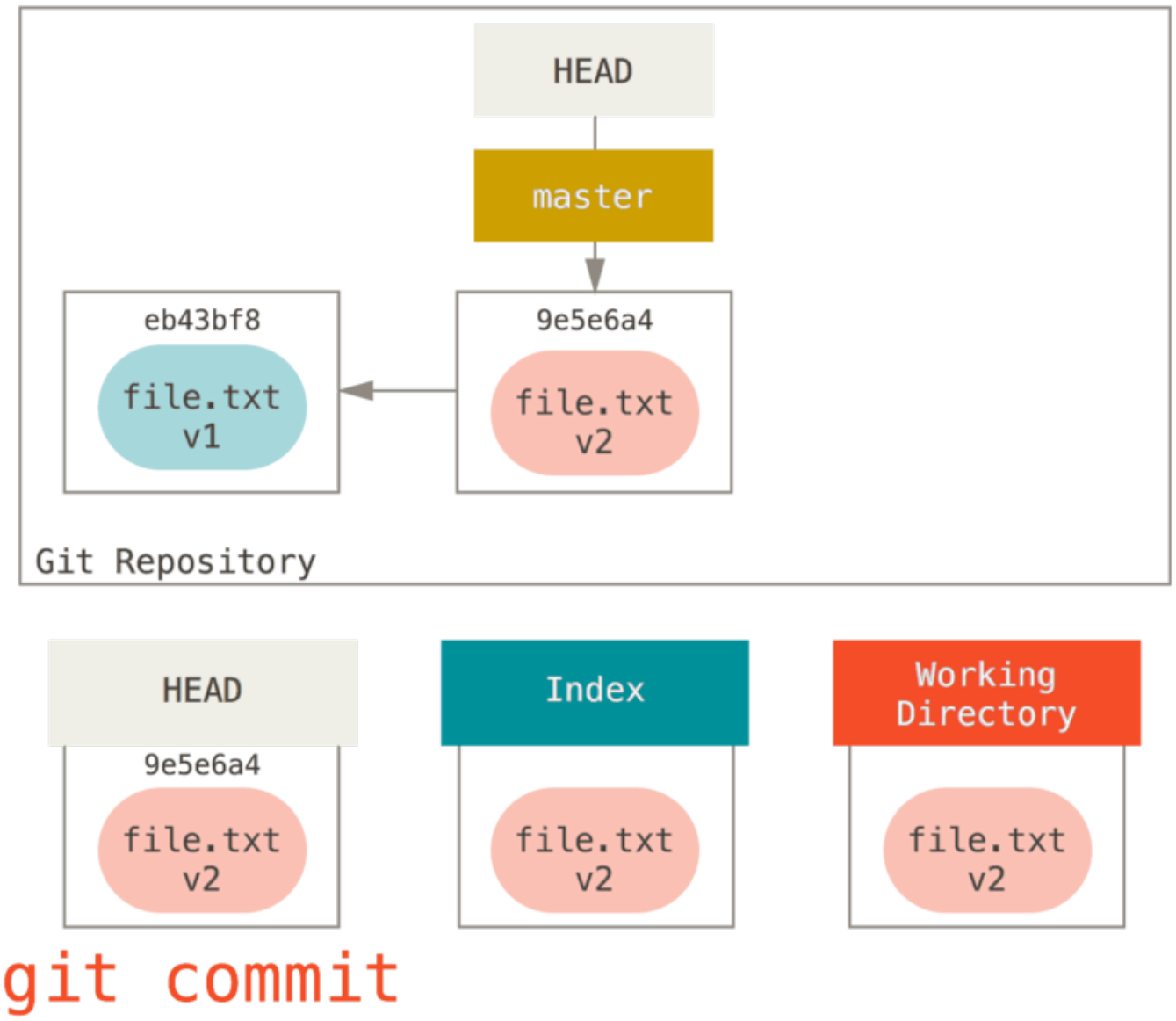
続いて、このテキストファイルの内容を変更してからコミットしてみましょう。 手順はさきほどと同じです。まずは、作業ディレクトリにあるファイルを変更します。 変更した状態のファイルを **v2** としましょう（図では赤塗りの部分）。




`git status` をここで実行すると、コマンド出力の “Changes not staged for commit” 欄に赤塗り部分のファイルが表示されます。作業ディレクトリ上のそのファイルの状態が、インデックスの内容とは異なっているからです。 では、`git add` を実行して変更をインデックスに追加してみましょう。



この状態で `git status` を実行すると、以下の図で緑色の枠内にあるファイルがコマンド出力の “Changes to be committed” 欄 に表示されます。インデックスと HEAD の内容に差分があるからです。次のコミット候補と前回のコミットの内容に差異が生じた、とも言えます。 では、`git commit` を実行してコミット内容を確定させましょう。



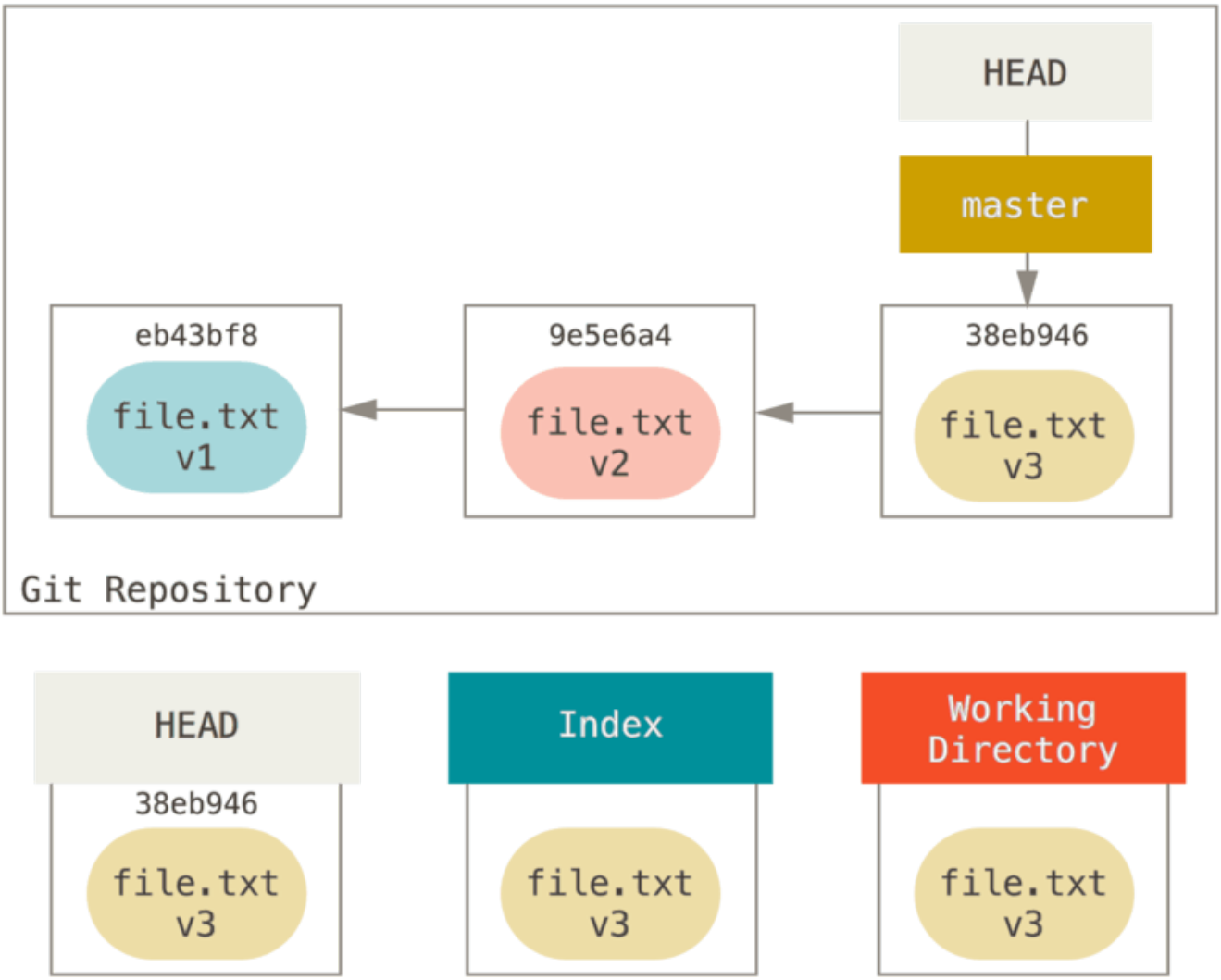

ここで `git status` を実行しても何も出力されません。3つのツリーが同じ状態に戻ったからです。

なお、ブランチを切り替えたりリモートブランチをクローンしても同じような処理が走ります。 ブランチをチェックアウトしたとしましょう。そうすると、**HEAD** はそのブランチを指すようになります。さらに、HEAD コミットのスナップショットで **インデックス** が上書きされ、そのデータが **作業ディレクトリ** にコピーされます。

### リセットの役割

これから説明する内容に沿って考えれば、`reset` コマンドの役割がわかりやすくなるはずです。

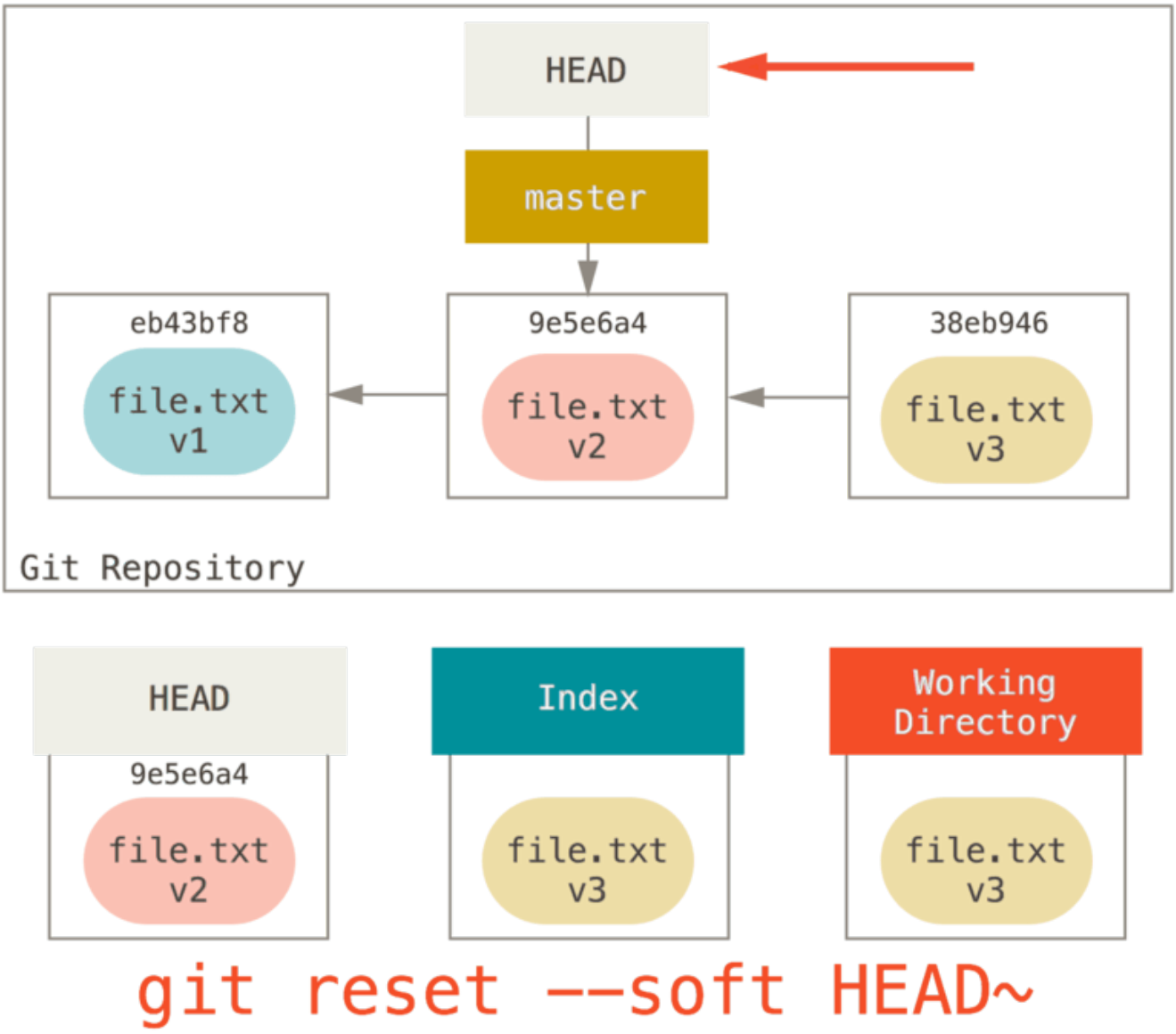
説明で使う例として、さきほど使った `file.txt` をまた編集し、コミットしたと仮定します。その場合、このリポジトリの歴史は以下のようになります。




では、`reset` コマンドの処理の流れを順を追って見ていきましょう。単純な方法で3つのツリーが操作されていきます。 一連の処理は、最大で3つになります。

**処理1 HEAD の移動**

`reset` コマンドを実行すると、HEAD に指し示されているものがまずは移動します。 これは、`checkout` のときのような、HEAD そのものを書き換えてしまう処理ではありません。HEAD が指し示すブランチの方が移動する、ということです。 つまり、仮に HEAD が `master` ブランチを指している（`master` ブランチをチェックアウトした状態）場合、`git reset 9e5e64a` を実行すると `master` ブランチがコミット `9e5e64a` を指すようになります。



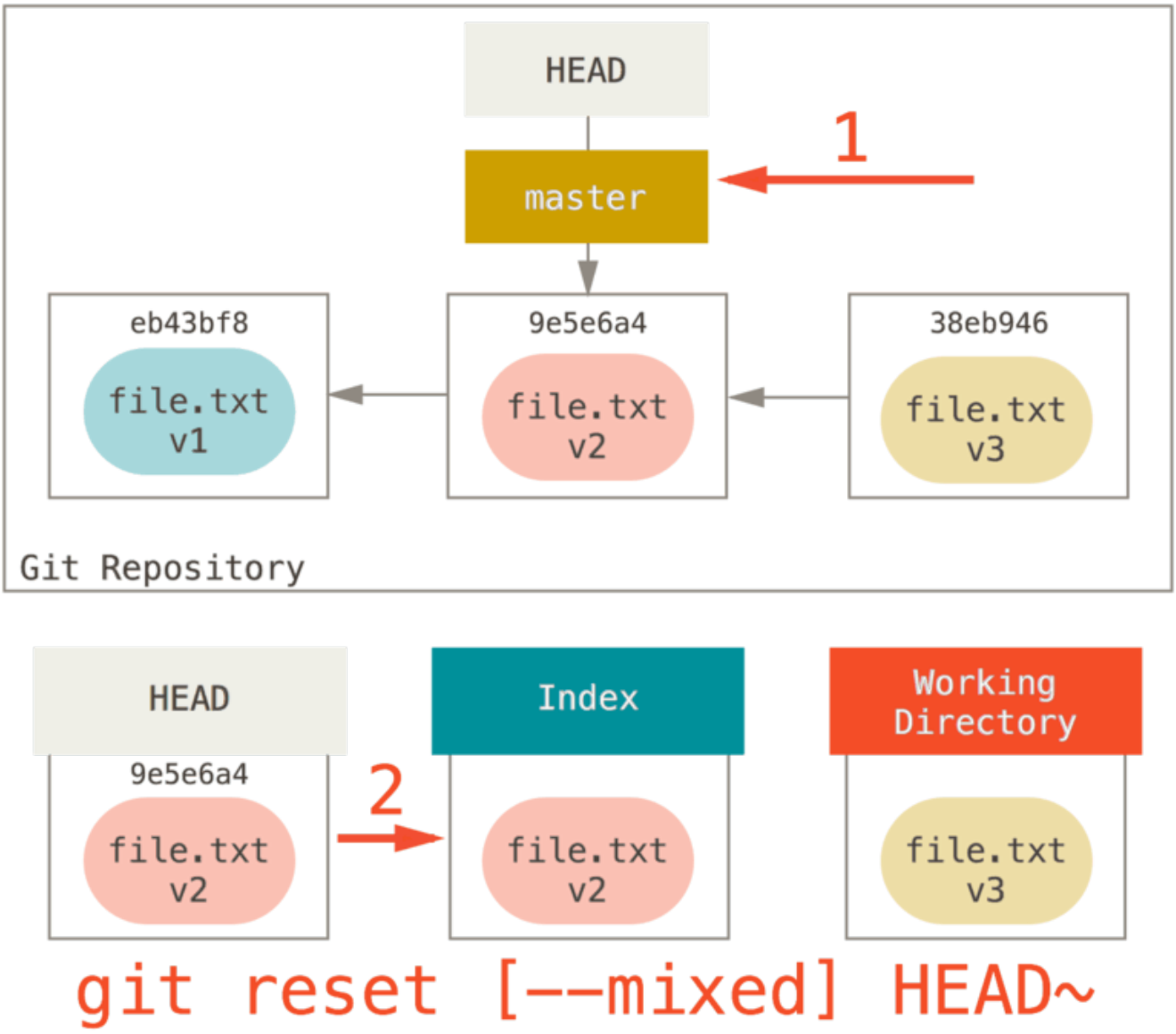

付与されたオプションがなんであれ、コミットを指定して `reset` コマンド実行すると、必ず上記の処理が走ります。 `reset --soft` オプションを使った場合は、コマンドはここで終了します。

そして、改めて図を見てみると、直近の `git commit` コマンドが取り消されていることがわかると思います。 通常であれば、`git commit` を実行すると新しいコミットが作られ、HEAD が指し示すブランチはそのコミットまで移動します。 また、`reset` を実行して `HEAD~`（HEAD の親）までリセットすれば、ブランチは以前のコミットまで巻き戻されます。この際、インデックスや作業ディレクトリは変更されません。 なお、この状態でインデックスを更新して `git commit` を実行すれば、`git commit --amend` を行った場合と同じ結果が得られます（詳しくは 直近のコミットの変更 を参照してください)。

**処理2 インデックスの更新 (--mixed)**

ここで `git status` を実行すると、インデックスの内容と変更された HEAD の内容との差分がわかることを覚えておきましょう。

第2の処理では、`reset` は HEAD が指し示すスナップショットでインデックスを置き換えます。

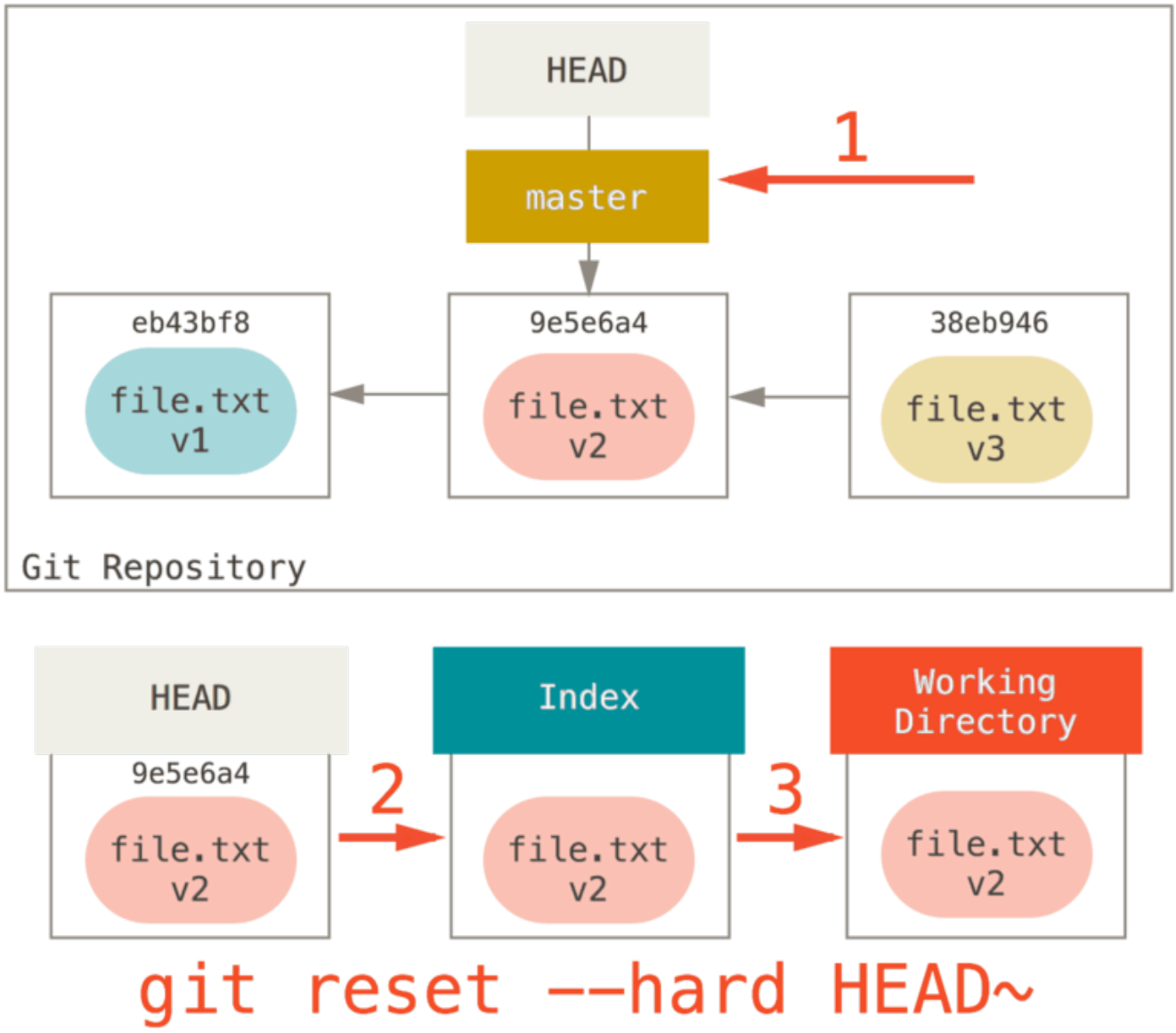



`--mixed` オプションを使うと、`reset` はここで終了します。 また、このオプションはデフォルトになっています。ここでの例の `git reset HEAD~` のようにオプションなしでコマンドを実行しても、`reset` はここで終了します。

では、もう一度図を見てみましょう。直近の `commit` がさきほどと同様に取り消されており、さらにインデックスの内容も *取り消された* ことがわかります。 `git add` でインデックスに追加し、`git commit` でコミットとして確定させた内容が取り消されたということです。

**処理3 作業ディレクトリの更新 (--hard)**

`reset` の第3の処理は、作業ディレクトリをインデックスと同じ状態にすることです。 `--hard` オプションを使うと、処理はこの段階まで進むことになります。




第3の処理が走ると何が起こるのでしょうか。 まず、直近のコミットが巻き戻されます。`git add` と `git commit` で確定した内容も同様です。**さらに**、作業ディレクトリの状態も巻き戻されてしまいます。

`--hard` オプションを使った場合に限り、`reset` コマンドは危険なコマンドになってしまうことを覚えておいてください。Git がデータを完全に削除してしまう、数少ないパターンです。 `reset` コマンドの実行結果は簡単に取り消せますが、`--hard` オプションに限ってはそうはいきません。作業ディレクトリを強制的に上書きしてしまうからです。 ここに挙げた例では、**v3** バージョンのファイルは Git のデータベースにコミットとしてまだ残っていて、`reflog` を使えば取り戻せます。ただしコミットされていない内容については、上書きされてしまうため取り戻せません。

**要約**

`reset` コマンドを使うと、3つのツリーを以下の順で上書きしていきます。どこまで上書きするかはオプション次第です。

1. HEAD が指し示すブランチを移動する *(`--soft` オプションを使うと処理はここまで)*

2. インデックスの内容を HEAD と同じにする *(`--hard` オプションを使わなければ処理はここまで)*
3. 作業ディレクトリの内容をインデックスと同じにする

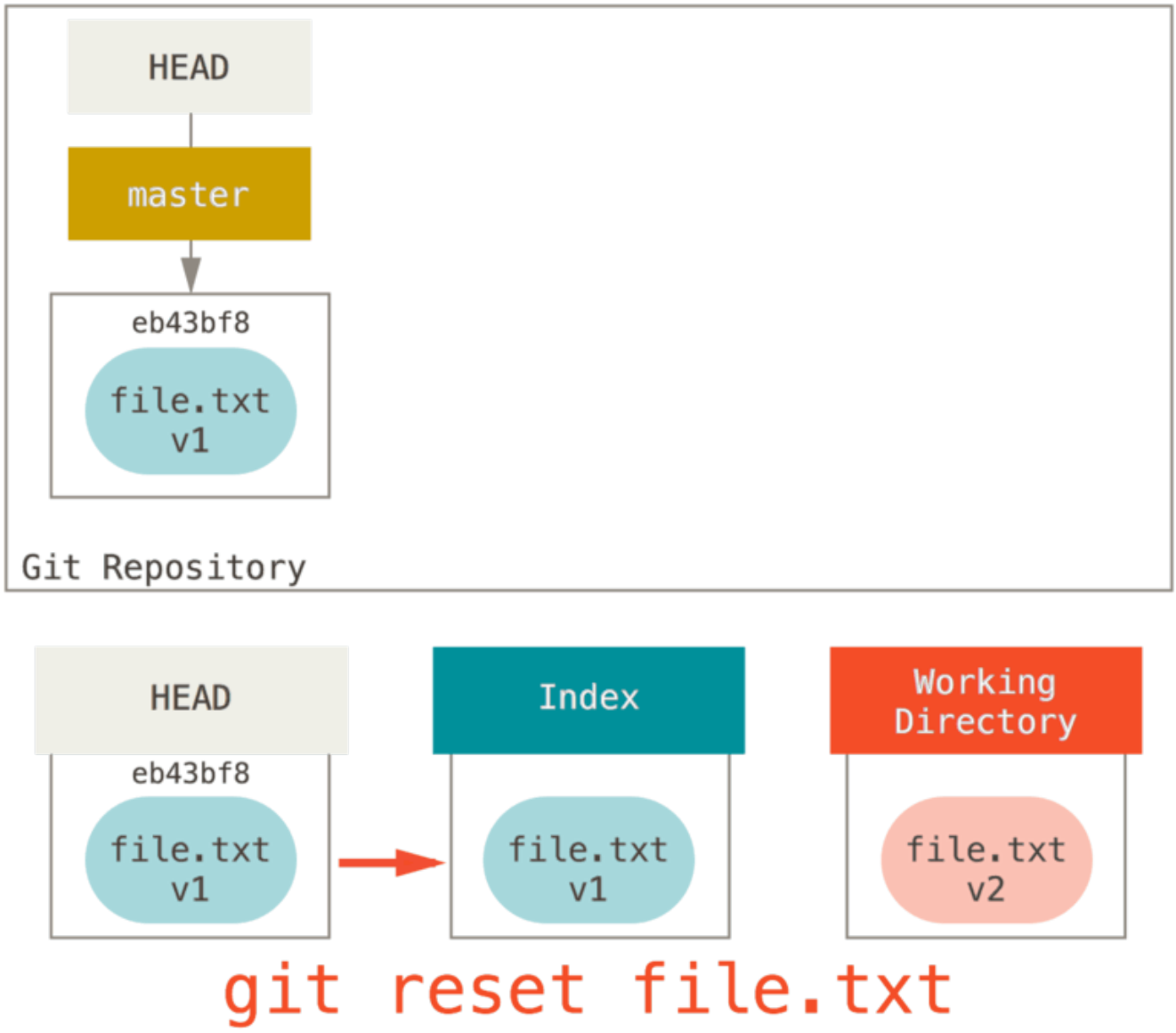
## パスを指定したリセット

ここまでで、`reset` の基礎と言える部分を説明してきました。次に、パスを指定して実行した場合の挙動について説明します。 パスを指定して `reset` を実行すると、処理1は省略されます。また、処理2と3については、パスで指定された範囲（ファイル郡）に限って実行されます。 このように動作するのはもっともな話です。処理1で操作される HEAD はポインタにすぎず、指し示せるコミットは一つだけだからです（こちらのコミットのこの部分と、あちらのコミットのあの部分、というようには指し示せません）。 一方、インデックスと作業ディレクトリを一部分だけ更新することは *可能* です。よって、リセットの処理2と3は実行されます。

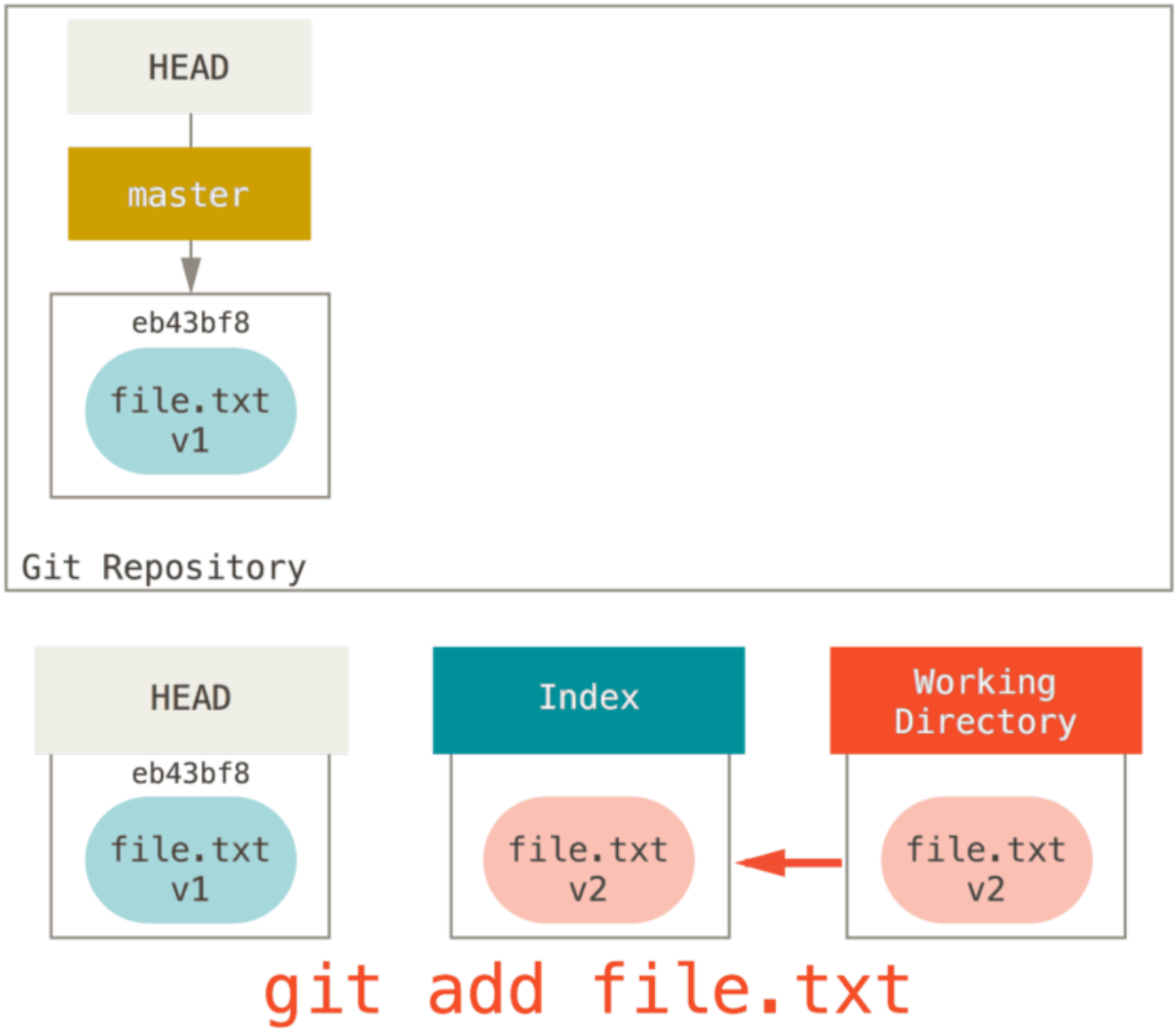
実際の例として、 `git reset file.txt` を実行したらどうなるか見ていきましょう。 このコマンドは `git reset --mixed HEAD file.txt` のショートカット版（ブランチやコミットの SHA-1 の指定がなく、`--soft` or `--hard` の指定もないため）です。実行すると、

1. HEAD が指し示すブランチを移動する *(この処理は省略)*
2. HEAD の内容でインデックスを上書きする *(処理はここまで)*

が行われます。要は、HEAD からインデックスに `file.txt` がコピーされるということです。

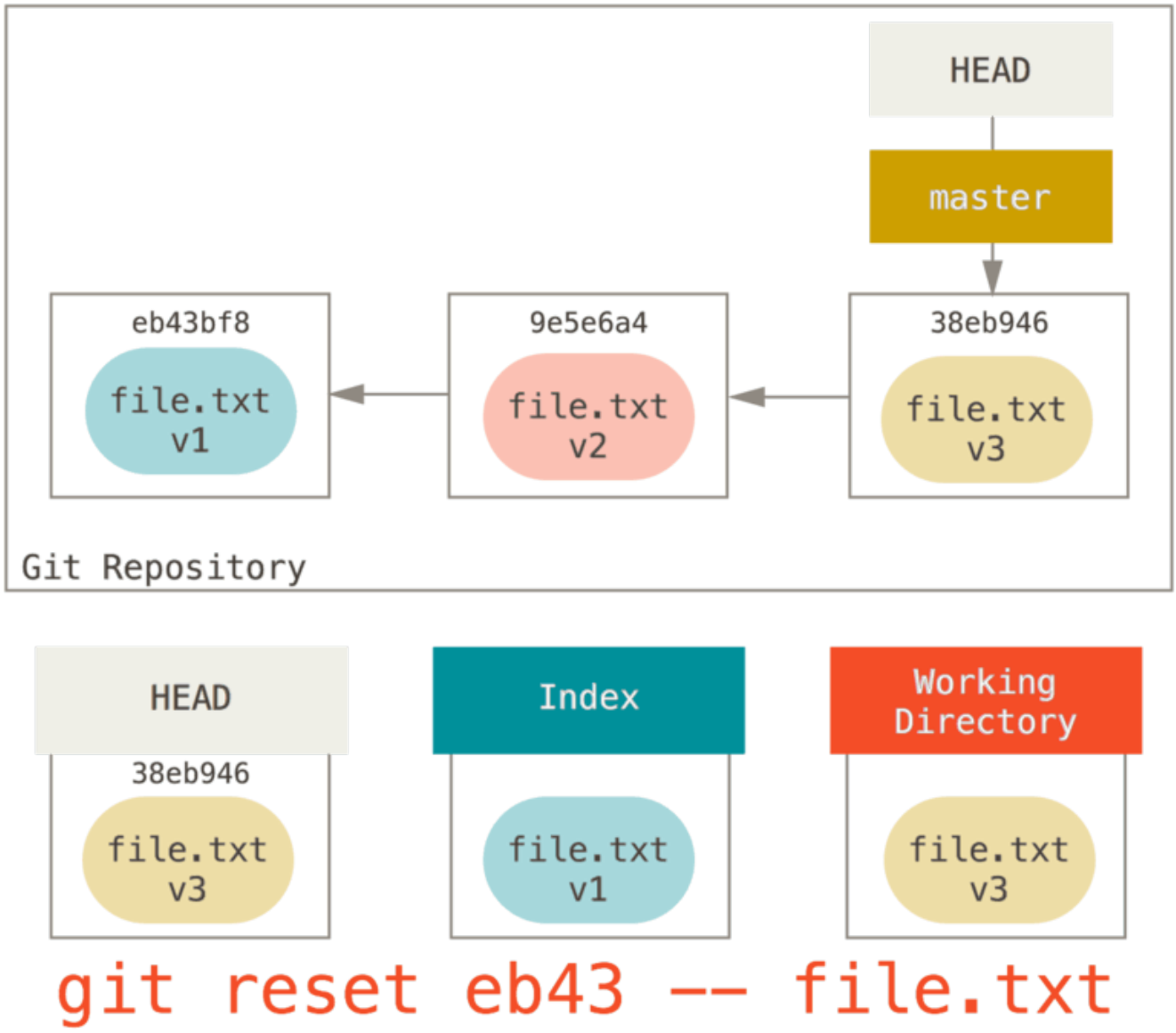



同時に、このコマンドは指定したファイルをステージされていない状態に戻す（*unstage*）、ということでもあります。 上の図（リセットコマンドを図示したもの）を念頭におきつつ、`git add` の挙動を考えてみてください。真逆であることがわかるはずです。




なお、ファイルをステージされていない状態に戻したいときはこのリセットコマンドを実行するよう、`git status` コマンドの出力には書かれています。その理由は、リセットコマンドが上述のような挙動をするからなのです。（詳細は ステージしたファイルの取り消し を確認してください）。

「HEAD のデータが欲しい」という前提で処理が行われるのを回避することもできます。とても簡単で、必要なデータを含むコミットを指定するだけです。`git reset eb43bf file.txt` のようなコマンドになります。




これを実行すると、作業ディレクトリ上の `file.txt` が **v1** の状態に戻り、`git add` が実行されたあと、作業ディレクトリの状態が **v3** に戻る、のと同じことが起こります（実際にそういった手順で処理されるわけではありませんが）。 さらに `git commit` を実行してみましょう。すると、作業ディレクトリ上の状態をまた **v1** に戻したわけではないのに、該当のファイルを **v1** に戻す変更がコミットされます。

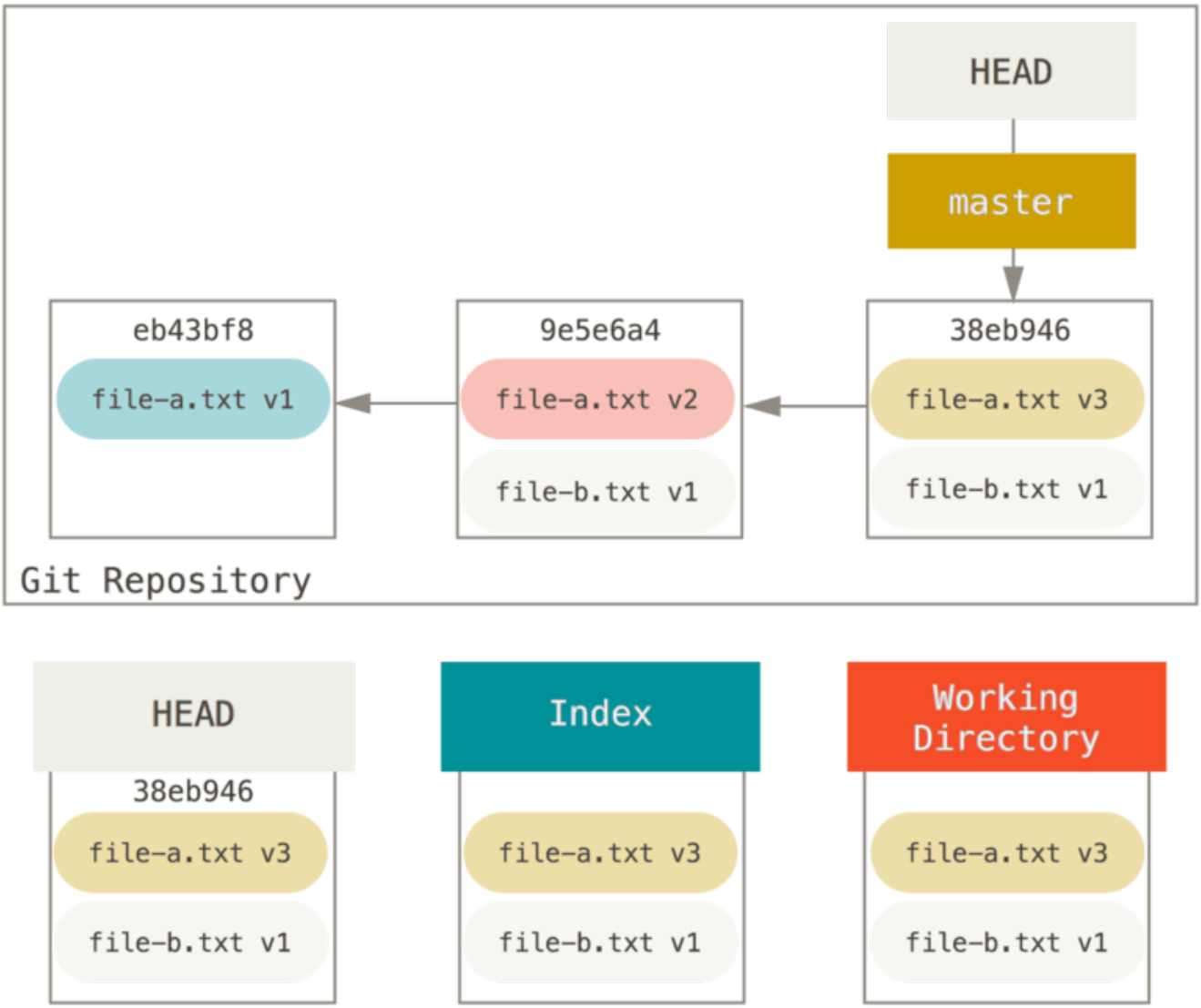
もうひとつ、覚えておくべきことを紹介します。 `git add` などと同じように、`reset` コマンドにも `--patch` オプションがあります。これを使うと、ステージした内容を塊ごとに作業ディレクトリに戻せます。 つまり、一部分だけを作業ディレクトリに戻したり以前の状態に巻き戻したりできるわけです。

## `reset` を使ったコミットのまとめ

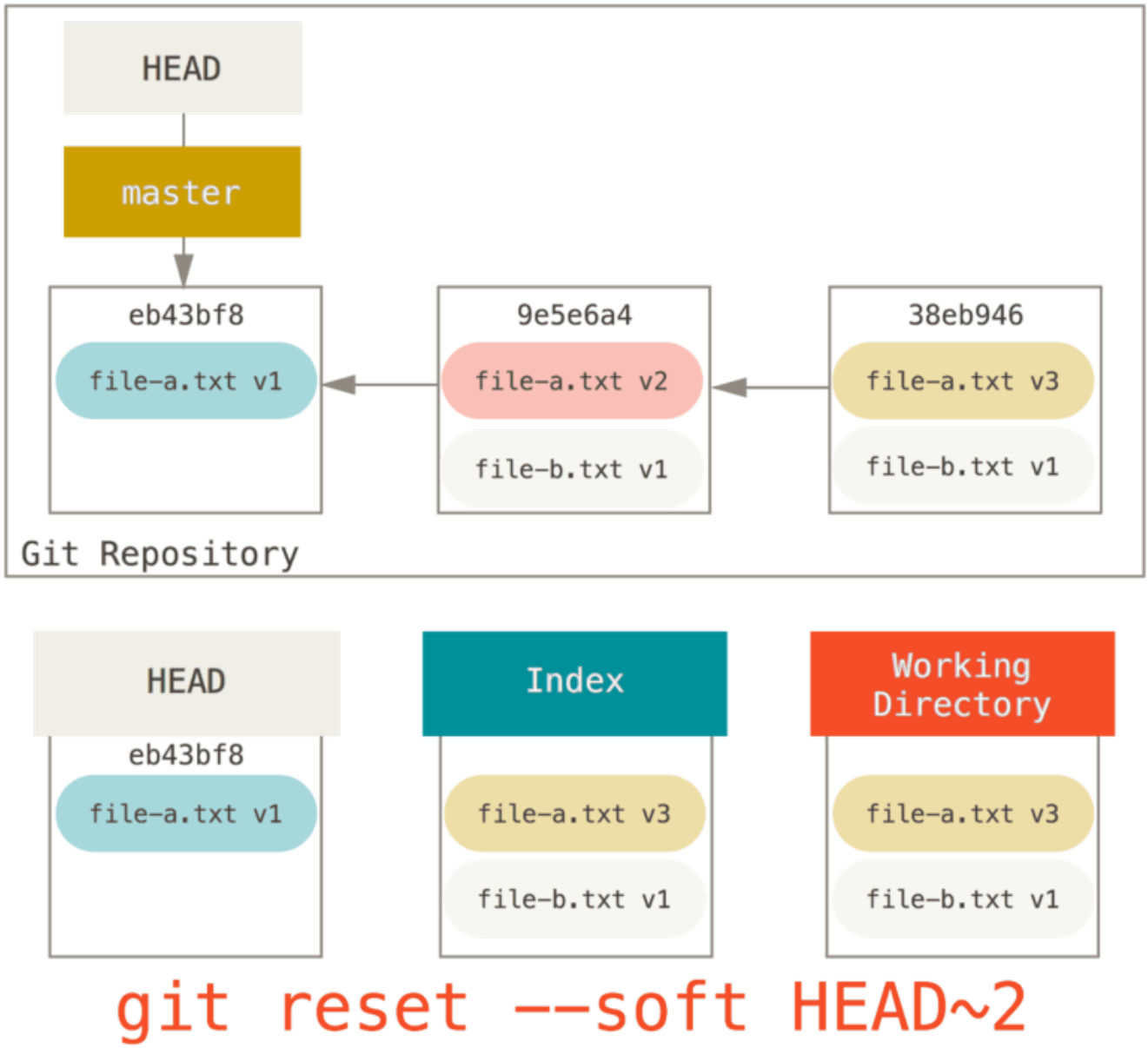
本節で学んだ方法を使う、気になる機能を紹介します。コミットのまとめ機能です。

「凡ミス」「WIP」「ファイル追加忘れ」のようなメッセージのコミットがいくつも続いたとします。 そんなときは `reset` を使いましょう。すっきりと一つにまとめられます（別の手段を コミットのまとめ で紹介していますが、今回の例では `reset` の方がわかりやすいと思います）。

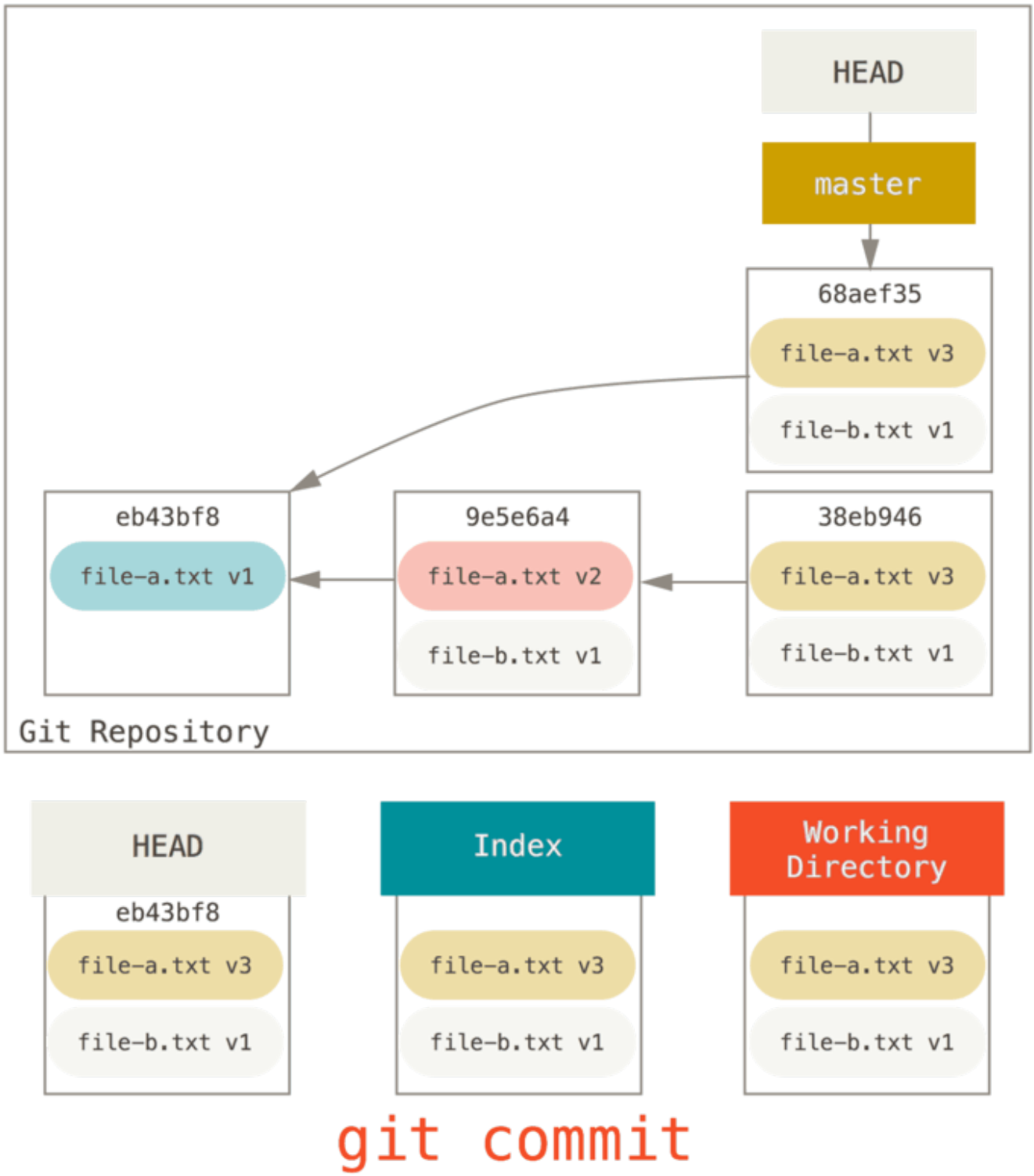
ここで、最初のコミットはファイル数が1、次のコミットでは最初からあったファイルの変更と新たなファイルの追加、その次のコミットで最初からあったファイルをまた変更、というコミット履歴を経てきたプロジェクトがあったとします。 二つめのコミットは作業途中のもの（WIP）だったので、どこかにまとめてしまいましょう。




まず、`git reset --soft HEAD~2` を実行して HEAD を過去のコミット（消したくはないコミットのうち古い方）へと移動させます。




そうしたら、あとは `git commit` を実行するだけです。




こうしてしまえば、1つめのコミットで `file-a.txt` v1 が追加され、2つめのコミットで `file-a.txt` が v3 に変更され `file-b.txt` が追加された、というコミット履歴が到達可能な歴史（プッシュすることになる歴史）になります。`file-a.txt` を v2 に変更したコミットを歴史から取り除くことができました。

### チェックアウトとの違い

最後に、`checkout` と `reset` の違いについて触れておきます。 3つのツリーを操作する、という意味では

`checkout` は `reset`
と同様です。けれど、コマンド実行時にファイルパスを指定するかどうかによって、少し違いがでてきます。

**パス指定なしの場合**

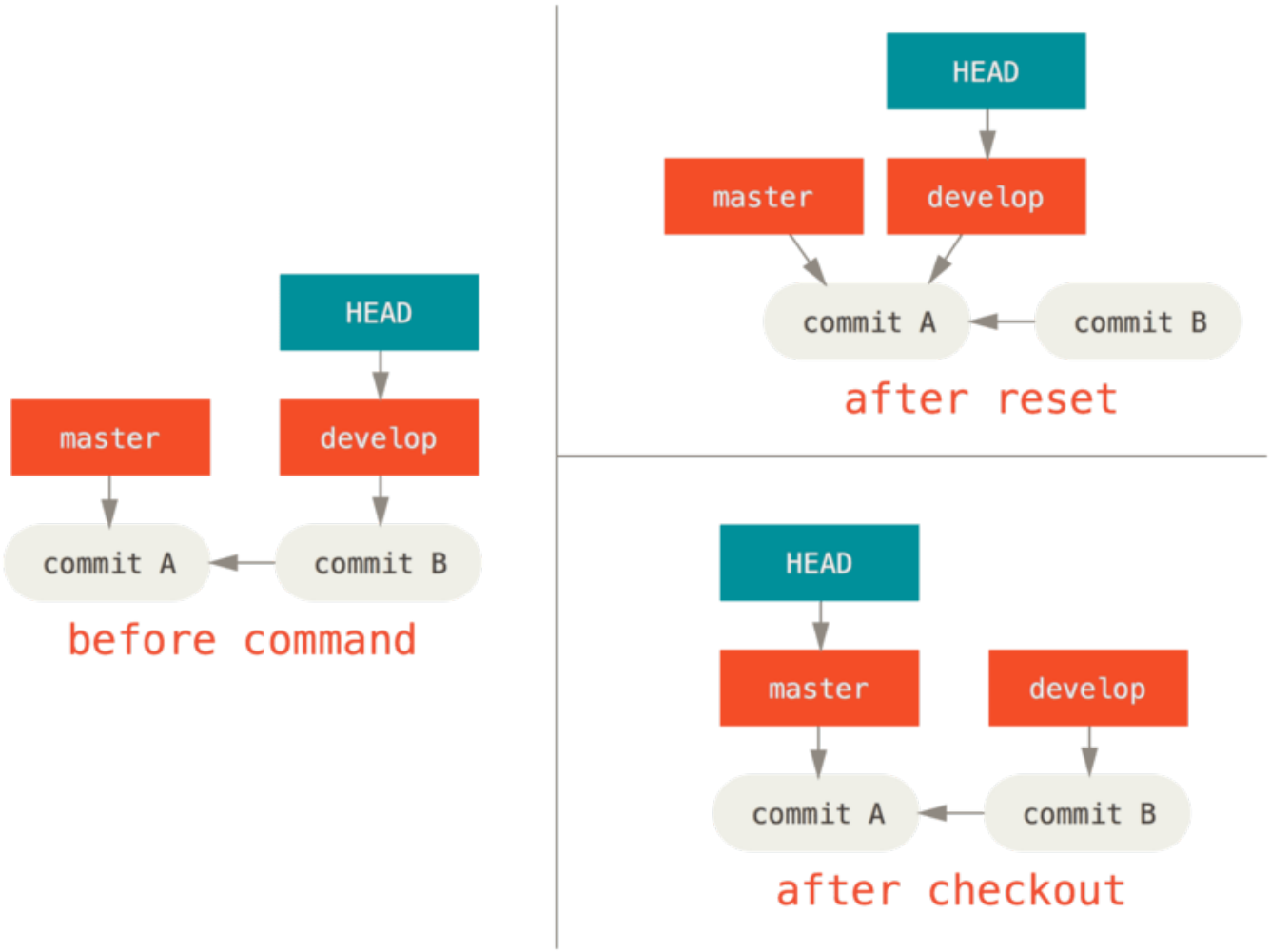
`git checkout [branch]` と `git reset --hard [branch]` の挙動は似ています。どちらのコマンドも、3つのツリーを `[branch]` の状態に変更するからです。ただし、大きな違いが2点あります。

まず、`reset --hard` とは違い、`checkout` は作業ディレクトリを守ろうとします。作業ディレクトリの内容を上書きしてしまう前に、未保存の変更がないかをチェックしてくれるのです。 さらに詳しく見てみると、このコマンドはもっと親切なことがわかります。作業ディレクトリのファイルに対し、“trivial” なマージを試してくれるのです。うまくいけば、*未変更* のファイルはすべて更新されます。 一方、`reset --hard` の場合、このようなチェックは行わずにすべてが上書きされます。

もうひとつの違いは、HEAD の更新方法です。 `reset` の場合はブランチの方が移動するのに対し、`checkout` の場合は HEAD のそのものが別ブランチに移動します。

具体例を挙げて説明しましょう。`master` と `develop` の2つのブランチが異なるコミットを指し示していて、`develop` の方をチェックアウトしているとします（HEAD は後者の方を向いた状態です）。 ここで `git reset master` を実行すると、`master` ブランチの最新のコミットを `develop` ブランチも指し示すようになります。 ですが、代わりに `git checkout master` を実行しても、`develop` ブランチは移動しません。HEAD が移動するのです。 その結果、HEAD は `master` の方を指し示すようになります。

どちらの場合でも HEAD がコミット A を指すようになるという意味では同じですが、*どのように* それが行われるかはずいぶん違います。 `reset` の場合は HEAD が指し示すブランチの方が移動するのに対し、`checkout` の場合は HEAD そのものが移動するのです。




**パス指定ありの場合**

`checkout` はパスを指定して実行することも出来ます。その場合、`reset` と同様、HEAD が動くことはありません。 実行されると指定したコミットの指定したファイルでインデックスの内容を置き換えます。`git reset [branch] file` と同じ動きです。しかし、`checkout` の場合は、さらに作業ディレクトリのファイルも置き換えます。 `git reset --hard [branch] file` を実行しても、まったく同じ結果になるでしょう（実際には `reset` ではこういうオプションの指定はできません）。作業ディレクトリを保護してはくれませんし、HEAD が動くこともありません。

また、`checkout` にも `git reset` や `git add` のように `--patch` オプションがあります。これを使えば、変更点を部分ごとに巻き戻していけます。

## まとめ

これまでの説明で `reset` コマンドについての不安は解消されたでしょうか。`checkout` との違いがまだまだ曖昧かもしれません。実行の仕方が多すぎて、違いを覚えるのは無理と言っても言い過ぎではないはずです。

どのコマンドがどのツリーを操作するか、以下の表にまとめておきました。“HEAD” の列は、該当のコマンドが HEAD が指し示すブランチの位置を動かす場合は “REF”、動くのが HEAD そのものの場合は “HEAD” としてあります。 「作業ディレクトリ保護の有無」の列はよく見ておいてください。その列が **いいえ** の場合は、実行結果をよくよく踏まえてからコマンドを実行するようにしてください。

| | HEAD | インデックス | 作業ディレクトリ | 作業ディレクトリ保護の有無 |
|---|---|---|---|---|
| **Commit Level** | | | | |
| `reset --soft [commit]` | REF | いいえ | いいえ | はい |
| `reset [commit]` | REF | はい | いいえ | はい |
| `reset --hard [commit]` | REF | はい | はい | **いいえ** |
| `checkout [commit]` | HEAD | はい | はい | はい |
| **File Level** | | | | |
| `reset (commit) [file]` | いいえ | はい | いいえ | はい |
| `checkout (commit) [file]` | いいえ | はい | はい | **いいえ** |

# 高度なマージ手法

Git を使うと、大抵の場合マージは簡単です。違うブランチを何度もマージすることも簡単なので、一度作ったブランチで延々と作業を続けながら、常に最新の状態に保っておけます。そうすれば、マージのたびに少しずつコンフリクトを解消することになるので、作業の最後で一度だけマージする場合のように、膨大なコンフリクトにあっけにとられることもなくなるでしょう。

とはいえ、ややこしいコンフリクトは発生してしまうものです。他のバージョン管理システムとは違い、Git は無理をしてまでコンフリクトを解消しようとはしません。Git は、マージの内容が明確かどうか正確に判断できるよう作られています。しかし、コンフリクトが発生した場合は、わかったつもりになってコンフリクトを解消してしまうようなことはしません。すぐに乖離してしまうようなブランチをいつまでもマージしないでおくと、問題になる場合があります。

この節では、どういった問題が起こりうるのか、そしてそういった状況を解決するのに役立つ Git のツールを見ていきます。また、いつもとは違う方法でマージを行うにはどうすればいいか、マージした内容を元に戻すにはどうすればいいかも見ていきましょう。

## マージのコンフリクト

マージのコンフリクトをどのように解消するか、基本的なところを マージ時のコンフリクト で紹介しました。ここでは、複雑なコンフリクトの場合に、状況を把握しコンフリクトを上手に解消するための Git ツールを紹介します。

まず、可能な限り、作業ディレクトリがクリーンな状態であることを確認しましょう。コンフリクトを起こす可能性のあるマージを実行するのはその後です。作業中の内容があるのなら、一時保存用のブランチを作ってコミットするか stash に隠してしまいましょう。こうしておけば、**何が** 起こってもやり直しがききます。以下で説明するヒントのなかには、作業ディレクトリの変更を保存せずにマージを行うと未保存の作業が消えてしまうものもあります。

では、わかりやすい例を見てみましょう。*hello world* と出力する単純な Ruby スクリプトです。

```
#! /usr/bin/env ruby

def hello
  puts 'hello world'
end

hello()
```

このスクリプトが保存されているリポジトリに `whitespace` というブランチを作ったら、ファイルの改行コードを Unix から DOS に変更します。これで、空白文字だけが全行分変更されました。次に、“hello world” という行を “hello mundo” に変更してみます。

```
$ git checkout -b whitespace
Switched to a new branch 'whitespace'

$ unix2dos hello.rb
unix2dos: converting file hello.rb to DOS format ...
$ git commit -am 'converted hello.rb to DOS'
[whitespace 3270f76] converted hello.rb to DOS
 1 file changed, 7 insertions(+), 7 deletions(-)

$ vim hello.rb
$ git diff -b
diff --git a/hello.rb b/hello.rb
index ac51efd..e85207e 100755
--- a/hello.rb
+++ b/hello.rb
@@ -1,7 +1,7 @@
 #! /usr/bin/env ruby

 def hello
-  puts 'hello world'
+  puts 'hello mundo'^M
 end

 hello()

$ git commit -am 'hello mundo change'
[whitespace 6d338d2] hello mundo change
 1 file changed, 1 insertion(+), 1 deletion(-)
```

ここで `master` ブランチに切り替えて、コメントで機能を説明しておきましょう。

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

$ vim hello.rb
$ git diff
diff --git a/hello.rb b/hello.rb
index ac51efd..36c06c8 100755
--- a/hello.rb
+++ b/hello.rb
@@ -1,5 +1,6 @@
 #! /usr/bin/env ruby

+# prints out a greeting
 def hello
   puts 'hello world'
 end

$ git commit -am 'document the function'
[master bec6336] document the function
 1 file changed, 1 insertion(+)
```

では、`whitespace` ブランチをマージしてみましょう。空白文字を変更したため、コンフリクトが発生してしまいます。

```
$ git merge whitespace
Auto-merging hello.rb
CONFLICT (content): Merge conflict in hello.rb
Automatic merge failed; fix conflicts and then commit the result.
```

**マージの中止**

コンフリクトには、対応方法がいくつかあります。まず、現状から抜け出す方法を説明します。コンフリクトが起こるとは思っていなかった、今はまだ処理したくない、といった場合、`git merge --abort` を実行すればマージ後の状況から抜け出せます。

```
$ git status -sb
## master
UU hello.rb

$ git merge --abort

$ git status -sb
## master
```

`git merge --abort` が実行されると、マージを実行する前の状態に戻ろうとします。これがうまくいかな

い可能性があるのが、作業ディレクトリの変更を隠しておらず、コミットもしていない状態でこのコマンドが実行された場合です。それ以外で失敗することはないでしょう。

また、一度やり直したいときは、`git reset --hard HEAD`（もしくは戻りたいコミットを指定）を実行してもよいでしょう。最新コミットの状態にリポジトリを戻してくれます。 ただし、コミットしていない内容が消去されてしまうことだけは覚えておいてください。変更内容をなかったことにしたいときだけ、このコマンドを実行するようにしましょう。

**空白文字の除外**

この例では、コンフリクトは空白文字が原因で起こっていました。例が簡単なのでそれが明確ですが、実際の場合でも見分けるのは簡単です。というのも、コンフリクトの内容が、一方で全行を削除しつつもう一方では全行を追加した形になっているからです。Git のデフォルトでは、これは「全行が変更された」と見なされ、マージは行えません。

ただし、デフォルトのマージ戦略で指定できる引数には、空白文字を適切に除外できるものもあります。大量の空白文字が原因でマージがうまくいかない場合は、一度中止して最初からやり直してみましょう。その際は、`-Xignore-all-space` か `-Xignore-space-change` のオプションを使ってください。前者は既存の空白文字に関する変更を **すべて** 無視し、後者は2文字以上の空白文字が連続している場合にそれを同一であるとみなして処理します。

```
$ git merge -Xignore-space-change whitespace
Auto-merging hello.rb
Merge made by the 'recursive' strategy.
 hello.rb | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```

この例ではファイルの実際の変更にコンフリクトがないので、空白文字の変更を無視してしまえば、あとはすんなりとマージできます。

チームのメンバーにスペースをタブに変えたがる（もしくはその反対）人がいたりすると、このオプションはきっと大助かりだと思います。

**マージの手動再実行**

Git は空白文字を前もって上手に処理してくれます。ただし、自動で処理するのは難しいけれど、変更の内容によっては処理をスクリプトに落とし込める場合があります。ここでは例として、空白文字がうまく処理されず、手動でコンフリクトを解消することになったとしましょう。

その場合、マージしようとしているファイルを前もって `dos2unix` プログラムで処理しておく必要があります。どうすればいいでしょうか。

手順はこうです。まずはじめに、実際にコンフリクトを発生させます。次に、コンフリクトしているファイルを、自分たちの分・相手側（マージしようとしているブランチ）の分・共通（両方のブランチの共通の祖先）の分の3バージョン用意します。最後に、自分たちか相手側、どちらかのファイルを修正し、該当のファイル1つだけを改めてマージします。

なお、この手順で使う3バージョンは簡単に用意できます。Git は、これらのバージョンを “stages” というイ

ンデックスに番号付きで保存してくれているのです。Stage 1 は共通の祖先、stage 2 は自分たちの分、Stage 3は MERGE_HEAD （マージしようとしている、“theirs” にあたる）の分になります。

コンフリクトが発生したファイルの3バージョンを用意するには、git show コマンドを以下のように指定して実行します。

```
$ git show :1:hello.rb > hello.common.rb
$ git show :2:hello.rb > hello.ours.rb
$ git show :3:hello.rb > hello.theirs.rb
```

そんな簡単なの？と拍子抜けしたのなら、Git の配管コマンドである ls-files -u を使ってみましょう。各ファイルの blob の SHA-1 を表示してくれます。

```
$ git ls-files -u
100755 ac51efdc3df4f4fd328d1a02ad05331d8e2c9111 1   hello.rb
100755 36c06c8752c78d2aff89571132f3bf7841a7b5c3 2   hello.rb
100755 e85207e04dfdd5eb0a1e9febbc67fd837c44a1cd 3   hello.rb
```

このとおり、:1:hello.rb は blob の SHA を調べるための簡易記法です。

3バージョン分のデータを作業ディレクトリに取り出せたので、相手側のファイルにある空白文字の問題を解消して、マージを再実行してみましょう。マイナーなコマンドですが、まさにこういったときのために使える git merge-file というコマンドを用います。

```
$ dos2unix hello.theirs.rb
dos2unix: converting file hello.theirs.rb to Unix format ...

$ git merge-file -p \
    hello.ours.rb hello.common.rb hello.theirs.rb > hello.rb

$ git diff -b
diff --cc hello.rb
index 36c06c8,e85207e..0000000
--- a/hello.rb
+++ b/hello.rb
@@@ -1,8 -1,7 +1,8 @@@
  #! /usr/bin/env ruby

 +# prints out a greeting
  def hello
-   puts 'hello world'
+   puts 'hello mundo'
  end

  hello()
```

こうすれば、コンフリクトしたファイルをきれいにマージできます。この方法を使うと、空白文字の問題は無視されずにマージ前にきちんと解決されるので、`ignore-space-change` オプションを使うよりも便利です。実際、`ignore-space-change` でマージを行ったら改行コードが DOS の行が数行残っており、改行コードが混在した状態になってしまっていました。

なお、自分たち（もしくは相手側）のファイルがどのように変更されたかを、ここでの変更をコミットする前に確認したい場合は、`git diff` コマンドを使います。そうすれば、作業ディレクトリにあるコミット予定のファイルを、上述の3ステージと比較できるのです。実際にやってみましょう。

まず、マージ前のブランチの状態を手元の現状と比較する（マージが何をどう変更したのか確認する）には、`git diff --ours` を実行します。

```
$ git diff --ours
* Unmerged path hello.rb
diff --git a/hello.rb b/hello.rb
index 36c06c8..44d0a25 100755
--- a/hello.rb
+++ b/hello.rb
@@ -2,7 +2,7 @@

 # prints out a greeting
 def hello
-  puts 'hello world'
+  puts 'hello mundo'
 end

 hello()
```

そうすると、作業中のブランチがどう変更されたか（マージすることでこのファイルがどう変更されるか）がすぐわかります。この例では、変更されるのは1行だけです。

次に、相手側のファイルがマージ前後でどう変わったかを確認するには、`git diff --theirs` を使います。なお、この例と次の例では、空白文字を除外するために `-b` オプションを使用しています。これから比較するのは空白文字が処理済みの手元のファイル `hello.theirs.rb` ではなく、Git のデータベースに格納されているデータだからです。

```
$ git diff --theirs -b
* Unmerged path hello.rb
diff --git a/hello.rb b/hello.rb
index e85207e..44d0a25 100755
--- a/hello.rb
+++ b/hello.rb
@@ -1,5 +1,6 @@
 #! /usr/bin/env ruby

+# prints out a greeting
 def hello
   puts 'hello mundo'
 end
```

そして、自分と相手、両側から変更を確認する場合は `git diff --base` を使いましょう。

```
$ git diff --base -b
* Unmerged path hello.rb
diff --git a/hello.rb b/hello.rb
index ac51efd..44d0a25 100755
--- a/hello.rb
+++ b/hello.rb
@@ -1,7 +1,8 @@
 #! /usr/bin/env ruby

+# prints out a greeting
 def hello
-  puts 'hello world'
+  puts 'hello mundo'
 end

 hello()
```

最後に、マージを手動で行うために作成したファイルは `git clean` コマンドで削除してしまいましょう。必要になることはもうありません。

```
$ git clean -f
Removing hello.common.rb
Removing hello.ours.rb
Removing hello.theirs.rb
```

#### コンフリクトのチェックアウト

ここで、さきほど試したコンフリクトの解決方法があまりよくなかった、もしくはマージ対象の一方（あるいは両方）を編集してもコンフリクトをうまく解消できず、これまでの流れを詳しく把握する必要が生じたとし

ます。

これを解説するには、先程の例を少し変更しておくほうがいいでしょう。今回は両方のブランチそれぞれにコミットが数回なされており、かつマージ時にはコンフリクトが発生するような状態だと仮定します。

```
$ git log --graph --oneline --decorate --all
* f1270f7 (HEAD, master) update README
* 9af9d3b add a README
* 694971d update phrase to hola world
| * e3eb223 (mundo) add more tests
| * 7cff591 add testing script
| * c3ffff1 changed text to hello mundo
|/
* b7dcc89 initial hello world code
```

`master` ブランチにしかないコミットが3つあり、`mundo` ブランチにしかないコミットも3つある、という状態です。ここで `mundo` ブランチをマージすれば、コンフリクトが発生してしまいます。

```
$ git merge mundo
Auto-merging hello.rb
CONFLICT (content): Merge conflict in hello.rb
Automatic merge failed; fix conflicts and then commit the result.
```

どのようなコンフリクトが発生したのか確認しておきましょう。ファイルを開いてみると、以下の様な状態になっていました。

```
#! /usr/bin/env ruby

def hello
<<<<<<< HEAD
  puts 'hola world'
=======
  puts 'hello mundo'
>>>>>>> mundo
end

hello()
```

マージ対象の両サイドで同じファイルの同じ箇所に違う変更を加えた結果、コンフリクトが発生してしまったことがわかります。

こういった場合に使える、コンフリクトの発生原因を確認できるツールを紹介します。コンフリクトをどう解消すればいいかが明確だとは限りません。そういったときは、経緯を把握する必要もあるはずです。

まず1つめは、`git checkout` コマンドの `--conflict` オプションです。これを実行すると、指定したファ

イルをもう一度チェックアウトし、コンフリクトマーカーを書きなおします。コンフリクトを手で直していてうまくいかず、最初からやり直すためにリセットしたいときに便利です。

`--conflict` オプションには `diff3` か `merge` が指定できます（デフォルトは `merge`）。前者を指定すると、コンフリクトマーカーが少し変わってきます。通常のマーカーである “ours” と “theirs” に加え、“base” も表示されるのです。より詳しく状況がわかると思います。

```
$ git checkout --conflict=diff3 hello.rb
```

これを実行すると、マーカーはいつもとは違い以下のようになるはずです。

```
#! /usr/bin/env ruby

def hello
<<<<<<< ours
  puts 'hola world'
||||||| base
  puts 'hello world'
=======
  puts 'hello mundo'
>>>>>>> theirs
end

hello()
```

これをコンフリクトマーカーのデフォルトにすることもできます。この表示の方が好みであれば、設定項目 `merge.conflictstyle` を `diff3` に変更してみましょう。

```
$ git config --global merge.conflictstyle diff3
```

`git checkout` コマンドには `--ours` や `--theirs` オプションを指定することもできます。これを使うと、何かをマージする代わりに、どちらか一方を選択して簡単にチェックアウトできます。

これは、バイナリデータのコンフリクトを解消するとき（使いたい方を選べばよい）や、他のブランチから特定のファイルを取り込みたいときに便利でしょう。後者であれば、マージコマンドを実行してから該当のファイルを `--ours` か `--theirs` を指定してチェックアウトし、コミットしてください。

#### マージの履歴

もう一つ、コンフリクトの解決に使える便利なツールが `git log` です。どういった流れでコンフリクトが発生したのかを追跡するときに使えます。というのも、歴史を少し紐解いてみると、平行して進行していた2つの開発作業がなぜコードの同じ部分を編集するに至ったか、その理由を思い出せたりするからです。

マージ対象のブランチに含まれるコミットを重複分を除いて表示させるには、トリプルドット で触れた「トリプルドット」記法を使います。

```
$ git log --oneline --left-right HEAD...MERGE_HEAD
< f1270f7 update README
< 9af9d3b add a README
< 694971d update phrase to hola world
> e3eb223 add more tests
> 7cff591 add testing script
> c3ffff1 changed text to hello mundo
```

この例では、全部で6コミットがわかりやすい状態でリスト表示されています。それぞれのコミットがどちらのブランチのものかもわかるようになっています。

また、より細かく流れを把握するために、表示内容を絞り込むこともできます。`git log` コマンドの `--merge` オプションを使うと、表示されるのはコンフリクトが発生しているファイルを編集したコミットだけになるのです。

```
$ git log --oneline --left-right --merge
< 694971d update phrase to hola world
> c3ffff1 changed text to hello mundo
```

また、このコマンドに `-p` オプションを追加すると、表示される内容がコンフリクトしているファイルの差分だけになります。コンフリクトの原因を把握して賢明な方法でそれを解消するために、必要な背景や経緯をすばやく理解したいときに **とても** 役に立つでしょう。

#### Combined Diff 形式

Git でマージを行うと、うまくマージされた内容はインデックスに追加されます。つまり、マージのコンフリクトが残っている状態で `git diff` を実行すると、コンフリクトの内容だけが表示されることになります。これを使えば、残ったコンフリクトだけを確認できます。

実際に、マージのコンフリクトが発生した直後に `git diff` を実行してみましょう。特徴的な diff 形式で差分が表示されます。

```
$ git diff
diff --cc hello.rb
index 0399cd5,59727f0..0000000
--- a/hello.rb
+++ b/hello.rb
@@@ -1,7 -1,7 +1,11 @@@
  #! /usr/bin/env ruby

  def hello
++<<<<<<< HEAD
 +  puts 'hola world'
++=======
+   puts 'hello mundo'
++>>>>>>> mundo
  end

  hello()
```

これは “Combined Diff” という形式で、各行の行頭2文字を使って関連情報を表示します。具体的には、作業ディレクトリの内容とマージ元のブランチ（「ours」）の内容に差分があれば1文字目を、作業ディレクトリとマージの相手側のブランチ（「theirs」）に差分があれば2文字目が使われます。

この例では、作業ディレクトリには存在する `<<<<<<<` と `>>>>>>>` の行が、マージ対象のブランチどちらにも存在していないことがわかります。それもそのはず、これらの行はマージによって挿入されたからです。差分をわかりやすくするために挿入されたこれらの行は、手動で削除する必要があります。

このコンフリクトを解消してから `git diff` を実行しても同じような内容が表示されますが、この場合はもう少し気の利いた内容になります。

```
$ vim hello.rb
$ git diff
diff --cc hello.rb
index 0399cd5,59727f0..0000000
--- a/hello.rb
+++ b/hello.rb
@@@ -1,7 -1,7 +1,7 @@@
  #! /usr/bin/env ruby

  def hello
-   puts 'hola world'
-  puts 'hello mundo'
++  puts 'hola mundo'
  end

  hello()
```

ここから読み取れるのは、“hola world”
はマージ元のブランチにはあって作業ディレクトリには存在せず、“hello mundo” はマージ対象のブランチにはあって作業ディレクトリには存在していないこと、更に “hola mundo” はマージ対象の両ブランチには存在しないけれど作業ディレクトリには存在していることです。これを使えば、コンフリクトをどのように解決したか、マージする前に確認できます。

`git log` を使っても、同じ内容を表示させられます。マージの際にどういった変更がなされたのか、後々になって確認する際に便利です。`git show` コマンドをマージコミットに対して実行した場合か、`git log -p`（デフォルトではマージコミット以外のコミットの内容をパッチ形式で表示）に `--cc` オプションを付与した場合、この形式の差分が出力されます。

```
$ git log --cc -p -1
commit 14f41939956d80b9e17bb8721354c33f8d5b5a79
Merge: f1270f7 e3eb223
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri Sep 19 18:14:49 2014 +0200

    Merge branch 'mundo'

    Conflicts:
        hello.rb

diff --cc hello.rb
index 0399cd5,59727f0..e1d0799
--- a/hello.rb
+++ b/hello.rb
@@@ -1,7 -1,7 +1,7 @@@
  #! /usr/bin/env ruby

  def hello
-   puts 'hola world'
-  puts 'hello mundo'
++  puts 'hola mundo'
  end

  hello()
```

## マージの取消

マージの方法がわかったということは、間違ってマージしてしまう可能性も出てきた、ということになります。 Git を使うことの利点は、間違ってもいい、ということです。というのも、（大抵は簡単に）修正できるからです。

マージコミットももちろん修正可能です。 例えば、トピックブランチで作業を開始し、間違ってそのブランチを `master` にマージしてしまったとしましょう。コミット履歴は以下のようになっているはずです。

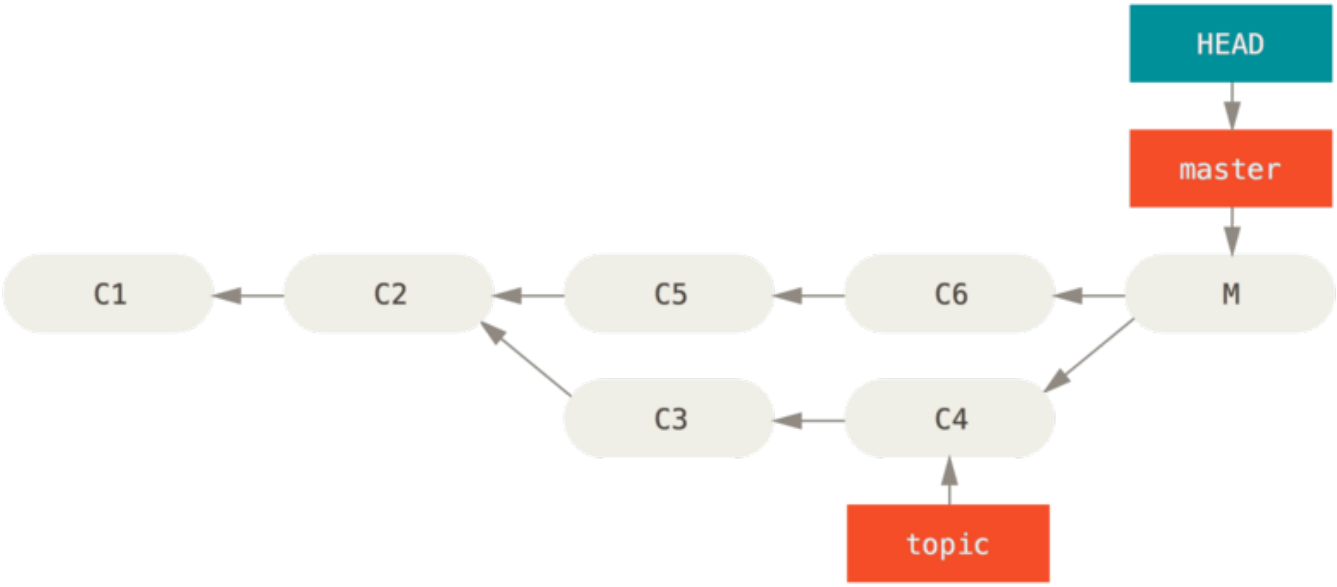



*Figure 137. 間違って作成したマージコミット*

この状況を修正するには2通りのやり方があります。どのように修正したいかに応じて使い分けましょう。

**参照の修正**

不要なマージコミットをまだプッシュしていないのなら、ブランチが指し示すコミットを変更してしまうのが一番簡単な解決方法です。 大半の場合、間違って実行した `git merge` の後に `git reset --hard HEAD~` を実行すれば、ブランチのポインタがリセットされます。実行結果は以下のようになるでしょう。

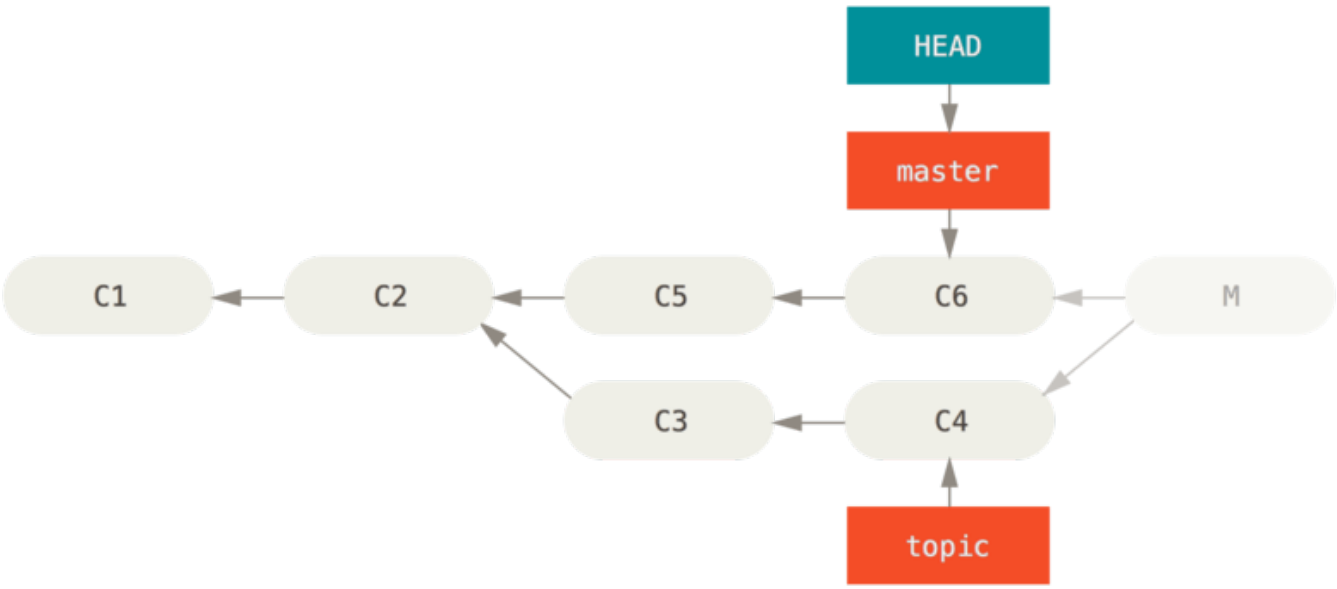



*Figure 138. `git reset --hard HEAD~` 実行後の歴史*

`reset` コマンドについては リセットコマンド詳説 で触れましたので、ここで何が起こっているか、理解するのは難しいことではないと思います。 念のためおさらいしておきましょう。`reset --hard` を実行すると、通常は以下の処理が走ります。

1. HEAD が指し示すブランチを移動する この例では、マージコミット (C6) が作成される前に master が指していたところまで戻します。
2. インデックスの内容を HEAD と同じにする
3. 作業ディレクトリの内容をインデックスと同じにする

この方法の欠点は、歴史を書き換えてしまう点です。共有リポジトリで作業していると、問題視される場合があります。 書き換えようとしているコミットをほかの人たちもプルしてしまっている場合は、reset は使わないほうが無難でしょう。理由については ほんとうは怖いリベース を確認してみてください。 また、新たなコミットがマージ以後に追加されている場合は、この方法はうまくいきません。参照を移動してしまうと、追加された内容を削除することになってしまうからです。

#### コミットの打ち消し

ブランチのポインタを動かすという上述の方法が機能しない場合、既存のコミットの内容を打ち消す新しいコミットを作ることもできます。 これは “revert” と呼ばれる操作で、今回の例では以下のようにすると呼び出せます。

```
$ git revert -m 1 HEAD
[master b1d8379] Revert "Merge branch 'topic'"
```

-m 1 オプションで、保持すべき「メイン」の親がどれであるかを指定します。 HEAD に対するマージ（git merge topic）を実行すると、マージコミットには2つの親ができます。HEAD (C6) と マージされるブランチの最新コミット (C4) です。 この例では、第2の親 (C4) をマージしたことで生じた変更をすべて打ち消しつつ、第1の親 (C6) の内容は保持したままにしてみます。

revert のコミットを含む歴史は以下のようになります。

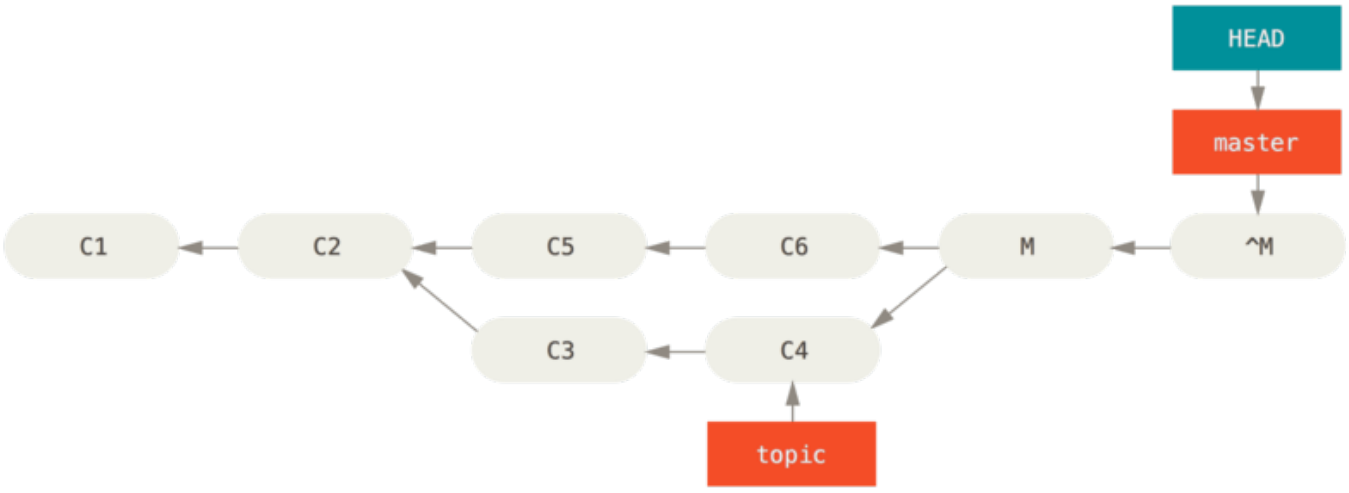



*Figure 139. git revert -m 1 の後の歴史*

新しく作成されたコミット ^M の内容はコミット C6 とまったく同じですので、歴史を今後振り返ると、マージなど一度も実施されていないかのように思えます。ただし、実際のところは HEAD の方の歴史にはマージされていないコミットが残ったままになってしまいます。 この状態で topic を master にマージしようとする

と、Git は状況を正確に判断できません。

```
$ git merge topic
Already up-to-date.
```

これは、`topic` ブランチにあるものは `master` ブランチにもすべて存在している、という状態です。 更に悪いことに、この状態の `topic` ブランチにコミットを追加してマージを行うと、revert されたマージ *後* の変更だけが取り込まれることになります。

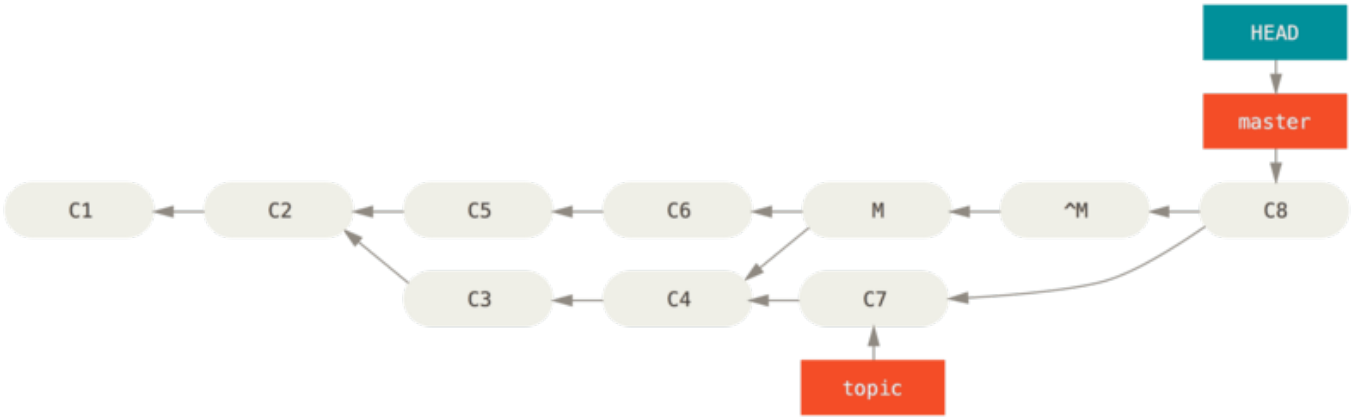



*Figure 140. よくないマージを含む歴史*

ここでは revert してしまった変更を取り戻したいわけですから、revert 済みの古いマージコミットをもう一度 revert し、 **そのうえで** 改めてマージするのが一番いいでしょう。

```
$ git revert ^M
[master 09f0126] Revert "Revert "Merge branch 'topic'""
$ git merge topic
```

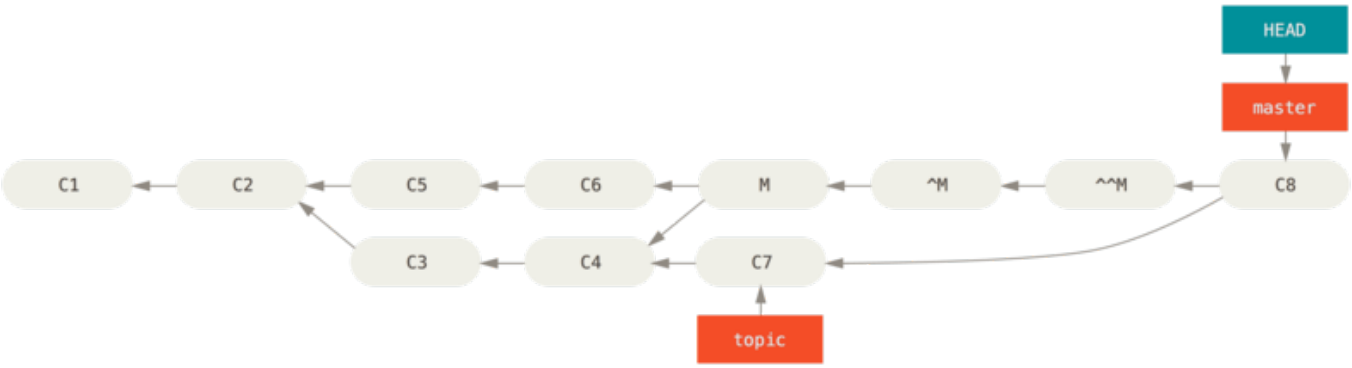



*Figure 141. revert 済みのマージコミットを再度マージした後の歴史*

そうすると、`M` と `^M` が互いを打ち消します。 `^^M` によって `C3` と `C4` の変更が取り込まれたことになりますし、`C8` のマージコミットによって `C7` が取り込まれます。これでようやっと、`topic` ブランチはすべてマージされました。

## 他のマージ手法

ここまでは2つのブランチをマージする通常の手法を見てきました。一般的には、「再帰」と呼ばれるマージ戦略によって処理されている手法です。これ以外にもブランチをマージする手法がありますので、いくつかをざっと紹介します。

### Our か Theirs の選択

1つめに紹介するのは、マージの「再帰」モードで使える便利なオプションです。`-X` と組み合わせて使う `ignore-all-space` や `ignore-space-change` といったオプションは既に紹介しました。Git ではそれ以外にも、コンフリクトが発生したときにマージ対象のどちらを優先するかを指定できます。

Git のデフォルトでは、マージしようとしているブランチ間でコンフリクトがある場合、コードにはコンフリクトを示すマーカーが挿入され、該当ファイルはコンフリクト扱いとなり、手動で解決することになります。そうではなく、マージ対象のブランチどちらかを優先して自動でコンフリクトを解消して欲しいとしましょう。その場合、`merge` コマンドに `-Xours` か `-Xtheirs` オプションを指定できます。

これらが指定されると、コンフリクトを示すマーカーは追加されません。マージ可能な差異は通常どおりマージされ、コンフリクトが発生する差異については指定された側のブランチの内容が採用されます。これはバイナリデータについても同様です。

以前使った “hello world” の例で確認してみましょう。作ったブランチをマージしようとするとコンフリクトが発生してしまいます。

```
$ git merge mundo
Auto-merging hello.rb
CONFLICT (content): Merge conflict in hello.rb
Resolved 'hello.rb' using previous resolution.
Automatic merge failed; fix conflicts and then commit the result.
```

ですが、`-Xours` か `-Xtheirs` を指定してマージすると、コンフリクトは発生しません。

```
$ git merge -Xours mundo
Auto-merging hello.rb
Merge made by the 'recursive' strategy.
 hello.rb | 2 +-
 test.sh  | 2 ++
 2 files changed, 3 insertions(+), 1 deletion(-)
 create mode 100644 test.sh
```

そうすると、“hello mundo” と “hola world” でコンフリクトが発生している部分にマーカーを挿入する代わりに、“hola world” の方が採用されます。そして、その場合でも、マージされる側のブランチにあるコンフリクトしない変更についてはすべてマージされます。

このオプションは、既に紹介した `git merge-file` コマンドでも使用可能です。`git merge-file --ours` のような形で実行すれば、ファイルを個別にマージするときに使えます。

また、同じようなことをしたいけれど、マージされる側の変更点は何一つ取り込みたくない、というようなことになったとしましょう。その場合、より強力な選択肢として “ours” というマージ *戦略* が使えます。これは “ours” を使って行う再帰的なマージ用の *オプション* とは異なります。

ではその戦略が何をするかというと、偽のマージが実行されるのです。マージ対象の両ブランチを親としたマージコミットが新たに作成されますが、マージされる側のブランチの内容については一切考慮されません。現在いるブランチの内容が、マージの結果としてそのままそっくり記録されます。

```
$ git merge -s ours mundo
Merge made by the 'ours' strategy.
$ git diff HEAD HEAD~
$
```

このとおり、マージ結果とマージ直前の状態に一切変更点がないことがわかります。

これが役に立つのは、後々になってマージを行う際に Git を勘違いさせて、ブランチをマージ済みとして取り扱わせたい場合です。具体例を挙げて説明しましょう。「リリース」ブランチを作成して作業を進めているとします。そのブランチは、いずれ “master” ブランチにマージするつもりです。ここで、“master” 上で行われたバグ修正を `release` ブランチにも取り込む必要が出てきました。そのためには、まずはバグ修正のブランチを `release` ブランチにマージし、続いて `merge -s ours` コマンドで同じブランチを `master` ブランチにもマージします（修正は既に取り込まれていますが、あえて実施します）。そうしておけば、`release` ブランチをマージする際に、バグ修正のブランチが原因でコンフリクトが発生することはありません。

**サブツリーマージ**

サブツリーマージの考え方は、ふたつのプロジェクトがあるときに一方のプロジェクトをもうひとつのプロジェクトのサブディレクトリに位置づけるというものです。 サブツリーマージを指定すると、Git は一方が他方のサブツリーであることを大抵の場合は理解して、適切にマージを行います。

これから、既存のプロジェクトに別のプロジェクトを追加し、前者のサブディレクトリとして後者をマージする例を紹介します。

まずは Rack アプリケーションをプロジェクトに追加します。 つまり、Rack プロジェクトをリモート参照として自分のプロジェクトに追加し、そのブランチにチェックアウトします。

```
$ git remote add rack_remote https://github.com/rack/rack
$ git fetch rack_remote --no-tags
warning: no common commits
remote: Counting objects: 3184, done.
remote: Compressing objects: 100% (1465/1465), done.
remote: Total 3184 (delta 1952), reused 2770 (delta 1675)
Receiving objects: 100% (3184/3184), 677.42 KiB | 4 KiB/s, done.
Resolving deltas: 100% (1952/1952), done.
From https://github.com/rack/rack
 * [new branch]      build      -> rack_remote/build
 * [new branch]      master     -> rack_remote/master
 * [new branch]      rack-0.4   -> rack_remote/rack-0.4
 * [new branch]      rack-0.9   -> rack_remote/rack-0.9
$ git checkout -b rack_branch rack_remote/master
Branch rack_branch set up to track remote branch
refs/remotes/rack_remote/master.
Switched to a new branch "rack_branch"
```

これで Rack プロジェクトのルートが `rack_branch` ブランチに取得でき、あなたのプロジェクトが `master` ブランチにある状態になりました。 まずどちらかをチェックアウトしてそれからもう一方に移ると、それぞれ別のプロジェクトルートとなっていることがわかります。

```
$ ls
AUTHORS         KNOWN-ISSUES   Rakefile      contrib         lib
COPYING         README         bin           example         test
$ git checkout master
Switched to branch "master"
$ ls
README
```

これは、考えようによっては変な話です。リポジトリにあるブランチがすべて、同一プロジェクトのブランチである必要はない、ということなのですから。めったにない話です（ちょっとやそっとのことでは役に立たないので）が、完全に異なる歴史を持つ複数のブランチを1つのリポジトリで保持するのはとても簡単なのです。

この例では、Rack プロジェクトを `master` プロジェクトのサブディレクトリとして取り込みたくなったとしましょう。そのときには、`git read-tree` を使います。`read-tree` とその仲間たちについては Gitの内側 で詳しく説明します。現時点では、とりあえず「あるブランチのルートツリーを読み込んで、それを現在のステージングエリアと作業ディレクトリに書き込むもの」だと認識しておけばよいでしょう。まず `master` ブランチに戻り、 `rack_branch` ブランチの内容を `master` ブランチの `rack` サブディレクトリに取り込みます。

```
$ git read-tree --prefix=rack/ -u rack_branch
```

これをコミットすると、Rack のファイルをすべてサブディレクトリに取り込んだようになります。そう、ま

るで tarball
からコピーしたかのような状態です。おもしろいのは、あるブランチでの変更を簡単に別のブランチにマージできるということです。もし Rack プロジェクトが更新されたら、そのブランチに切り替えてプルするだけで本家の変更を取得できます。

```
$ git checkout rack_branch
$ git pull
```

これで、変更を `master` ブランチにマージできるようになりました。`git merge -s subtree` を使えばうまく動作します。が、Git は歴史もともにマージしようとします。おそらくこれはお望みの動作ではないでしょう。変更をプルしてコミットメッセージを埋めるには、再帰的マージ戦略を指定するオプション `-Xsubtree` のほかに `--squash` オプションを使います（再帰的戦略はこの場合のデフォルトにあたりますが、使用されるオプションを明確にするためあえて記載してあります）。

```
$ git checkout master
$ git merge --squash -s recursive -Xsubtree=rack rack_branch
Squash commit -- not updating HEAD
Automatic merge went well; stopped before committing as requested
```

Rack プロジェクトでのすべての変更がマージされ、ローカルにコミットできる準備が整いました。この逆を行うこともできます。master ブランチの `rack` サブディレクトリで変更した内容を後で `rack_branch` ブランチにマージし、それをメンテナに投稿したり本家にプッシュしたりといったことも可能です。

この機能を使えば、サブモジュールを使った作業手順に似た手順（サブモジュール で紹介する予定）を、サブモジュールなしで採用できます。違うプロジェクトのデータをブランチとしてプロジェクトリポジトリ内に保持しておけますし、サブツリーマージを使ってそのブランチを取組中のプロジェクトに取り込むこともできます。これは見方によっては、例えば、すべてのコードが同じ場所にコミットされるという意味では、便利だといえるでしょう。ですが、欠点がないわけではありません。構成が複雑になり変更を取り込む際に間違いやすくなってしまうでしょう。関係ないリポジトリに誤ってプッシュしてしまうことだってあるかもしれません。

また、少し違和感を覚えるかもしれませんが、`rack` サブディレクトリの内容と `rack_branch` ブランチのコードの差分を取得する (そして、マージしなければならない内容を知る) には、通常の `diff` コマンドを使うことはできません。そのかわりに、`git diff-tree` で比較対象のブランチを指定します。

```
$ git diff-tree -p rack_branch
```

あるいは、`rack` サブディレクトリの内容と前回取得したときのサーバーの `master` ブランチとを比較するには、次のようにします。

```
$ git diff-tree -p rack_remote/master
```

# Rerere

`git rerere` コマンドはベールに包まれた機能といってもいいでしょう。これは “reuse recorded resolution” の略です。その名が示すとおり、このコマンドは、コンフリクトがどのように解消されたかを記録してくれます。そして、同じコンフリクトに次に出くわしたときに、自動で解消してくれるのです。

いくつもの場面で、この機能がとても役立つと思います。Git のドキュメントで挙げられている例は、長期にわたって開発が続いているトピックブランチを問題なくマージされるようにしておきたいけれど、そのためのマージコミットがいくつも生まれるような状況は避けたい、というものです。`rerere` を有効にした状態で、マージをときおり実行し、コンフリクトをそのたびに解消したうえで、マージを取り消してみてください。この手順を継続的に行っておけば、最終的なマージは容易なものになるはずです。`rerere` がすべてを自動で処理してくれるからです。

リベースする度に同じコンフリクトを処理することなく、ブランチをリベースされた状態に保っておくときにもこの方法が使えます。あるいは、コンフリクトをすべて解消して、ようやっとマージし終えた後に、リベースを使うことに方針を変更したとしましょう。`rerere` を使えば、同じコンフリクトを再度処理せずに済みます。

その他にも、開発中のトピックブランチをいくつもまとめてマージして、テスト可能な HEAD を生成するとき（Git 本体のプロジェクトでよく行われています）にもこのコマンドが使えます。テストが失敗したら、マージを取り消したうえで失敗の原因となったブランチを除外してからテストを再実行するわけですが、`rerere` を使えばその際にコンフリクトを解消する必要がなくなるのです。

`rerere` を有効にするには、以下の設定コマンドを実行しましょう。

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

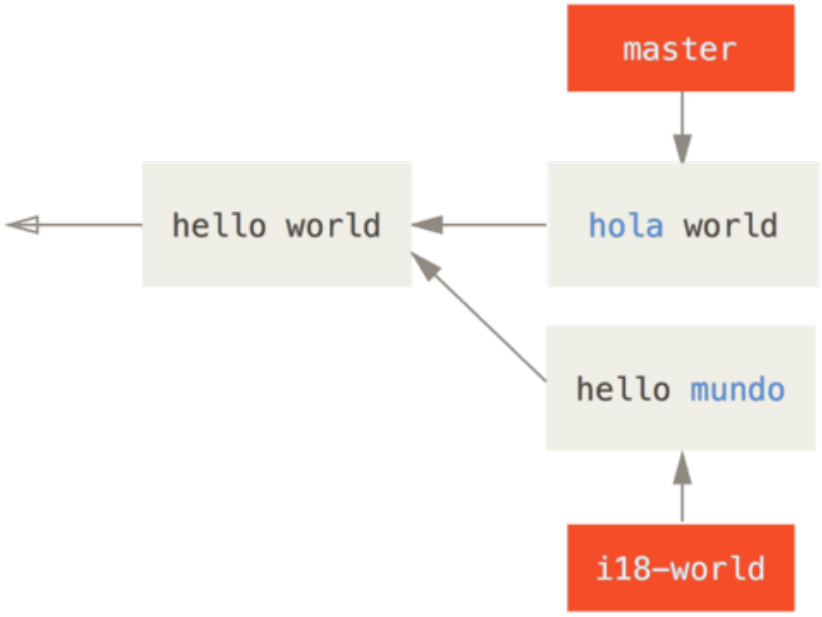
該当のリポジトリに `.git/rr-cache` というディレクトリを作成しても `rerere` は有効になりますが、設定するほうがわかりやすいでしょう。設定であれば、全リポジトリに適用することもできます。

では実際の例を見てみましょう。以前使ったような単純な例です。 `hello.rb` というファイル名の、以下のようなファイルがあったとします。

```
#! /usr/bin/env ruby

def hello
  puts 'hello world'
end
```

今いるブランチではこのファイルの “hello” という単語を “hola” に変更し、別のブランチでは “world” を “mundo” に変更したとします。前回と同様ですね。




これら2つのブランチをマージしようとすると、コンフリクトが発生します。

```
$ git merge i18n-world
Auto-merging hello.rb
CONFLICT (content): Merge conflict in hello.rb
Recorded preimage for 'hello.rb'
Automatic merge failed; fix conflicts and then commit the result.
```

コマンド出力に `Recorded preimage for FILE` という見慣れない行があるのに気づかれたでしょう。他の部分は、よくあるコンフリクトのメッセージと変わりありません。この時点で、`rerere` からわかることがいくつかあります。こういった場合、いつもであれば以下のように `git status` を実行し、何がコンフリクトしているのかを確認するものです。

```
$ git status
# On branch master
# Unmerged paths:
#   (use "git reset HEAD <file>..." to unstage)
#   (use "git add <file>..." to mark resolution)
#
#   both modified:      hello.rb
#
```

ですが、ここで `git rerere status` を実行すると、どのファイルのマージ前の状態が `git rerere` によって保存されたかがわかります。

```
$ git rerere status
hello.rb
```

更に、`git rerere diff` を実行すると、コンフリクト解消の状況がわかります。具体的には、着手前がどういう状態であったか、どういう風に解消したのか、がわかります。

```
$ git rerere diff
--- a/hello.rb
+++ b/hello.rb
@@ -1,11 +1,11 @@
 #! /usr/bin/env ruby

 def hello
-<<<<<<<
-  puts 'hello mundo'
-=======
+<<<<<<< HEAD
   puts 'hola world'
->>>>>>>
+=======
+  puts 'hello mundo'
+>>>>>>> i18n-world
 end
```

また（`rerere` 特有の話ではありませんが）、コンフリクトしているファイルと、そのファイルの3バージョン（マージ前・コンフリクトマーカー左向き・コンフリクトマーカー右向き）が `ls-files -u` を使うとわかります。

```
$ git ls-files -u
100644 39804c942a9c1f2c03dc7c5ebcd7f3e3a6b97519 1   hello.rb
100644 a440db6e8d1fd76ad438a49025a9ad9ce746f581 2   hello.rb
100644 54336ba847c3758ab604876419607e9443848474 3   hello.rb
```

さて、このコンフリクトは `puts 'hola mundo'` と修正しておきます。そして、もう一度 `rerere diff` コマンドを実行すると、rerere が記録する内容を確認できます。

```
$ git rerere diff
--- a/hello.rb
+++ b/hello.rb
@@ -1,11 +1,7 @@
 #! /usr/bin/env ruby

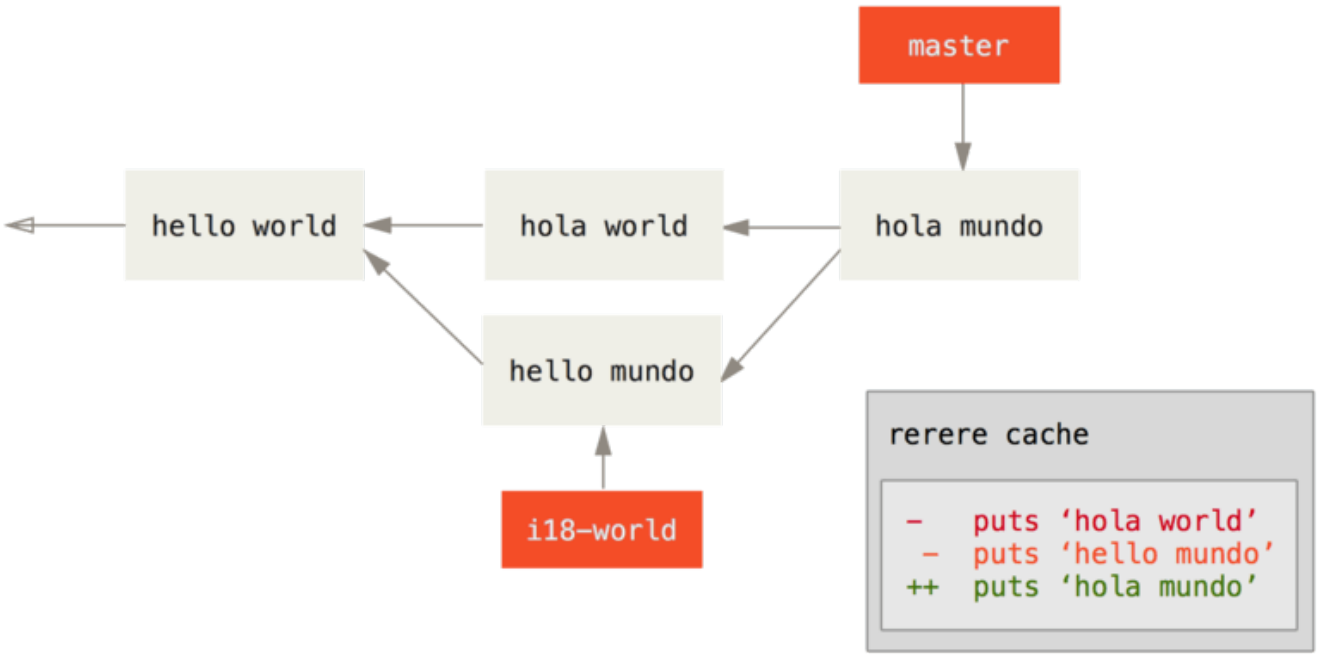
 def hello
-<<<<<<<
-  puts 'hello mundo'
-=======
-  puts 'hola world'
->>>>>>>
+  puts 'hola mundo'
 end
```

これを記録したということは、`hello.rb` に同じコンフリクト（一方は “hello mundo” でもう一方が “hola world”）が見つかった場合、自動的に “hola mundo” に修正されるということになります。

では、この変更内容をコミットしましょう。

```
$ git add hello.rb
$ git commit
Recorded resolution for 'hello.rb'.
[master 68e16e5] Merge branch 'i18n'
```

コマンド出力から、Git がコンフリクト解消方法を記録した（"Recorded resolution for FILE"）ことがわかります。




ではここで、このマージを取り消して master ブランチにリベースしてみましょう。リセットコマンド詳説 で紹介したとおり、ブランチを巻き戻すには `reset` を使います。

```
$ git reset --hard HEAD^
HEAD is now at ad63f15 i18n the hello
```

マージが取り消されました。続いてトピックブランチをリベースしてみます。

```
$ git checkout i18n-world
Switched to branch 'i18n-world'

$ git rebase master
First, rewinding head to replay your work on top of it...
Applying: i18n one word
Using index info to reconstruct a base tree...
Falling back to patching base and 3-way merge...
Auto-merging hello.rb
CONFLICT (content): Merge conflict in hello.rb
Resolved 'hello.rb' using previous resolution.
Failed to merge in the changes.
Patch failed at 0001 i18n one word
```

予想どおり、マージコンフリクトが発生しました。一方、`Resolved FILE using previous resolution` というメッセージも出力されています。該当のファイルを確認してみてください。コンフリクトはすでに解消されていて、コンフリクトを示すマーカーは挿入されていないはずです。

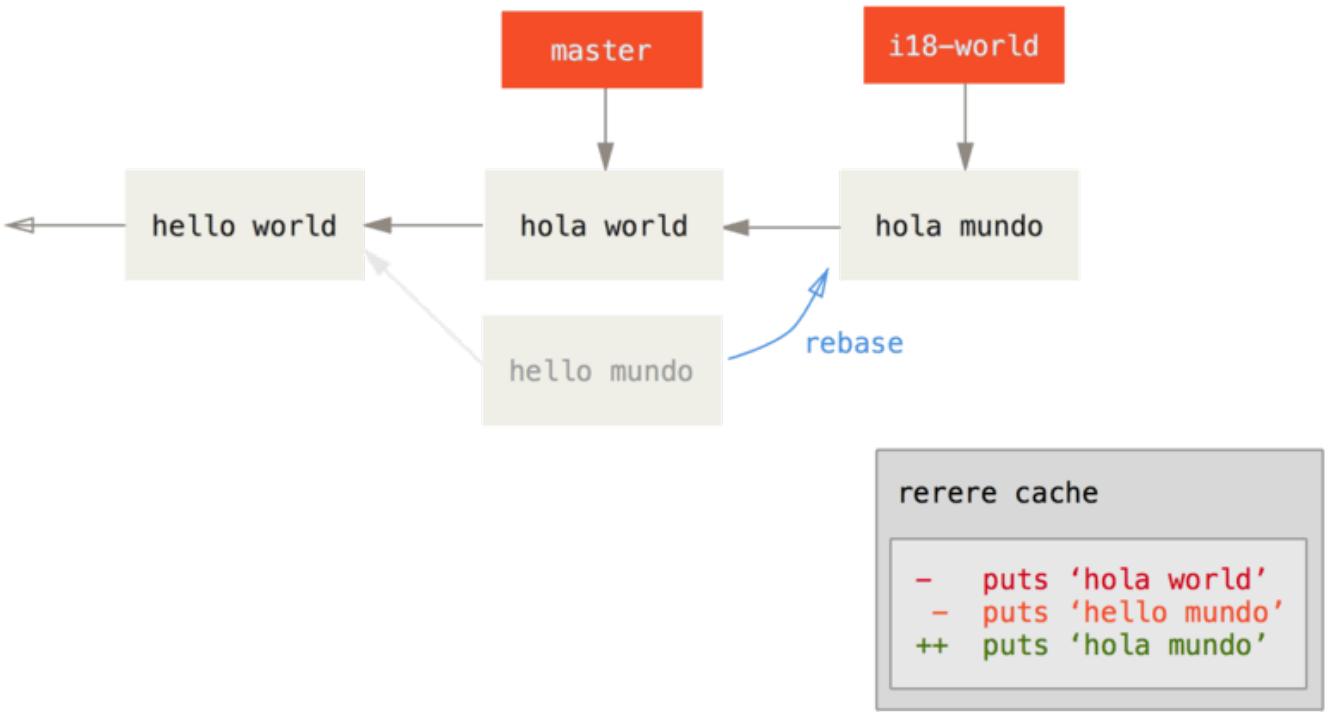
```
#! /usr/bin/env ruby

def hello
  puts 'hola mundo'
end
```

また、ここで `git diff` を実行すると、コンフリクトの再解消がどのように自動処理されたかがわかります。

```
$ git diff
diff --cc hello.rb
index a440db6,54336ba..0000000
--- a/hello.rb
+++ b/hello.rb
@@@ -1,7 -1,7 +1,7 @@@
  #! /usr/bin/env ruby

  def hello
-   puts 'hola world'
 -  puts 'hello mundo'
++  puts 'hola mundo'
  end
```




なお、`checkout` コマンドを使うと、ファイルがコンフリクトした状態を再現できます。

```
$ git checkout --conflict=merge hello.rb
$ cat hello.rb
#! /usr/bin/env ruby

def hello
<<<<<<< ours
  puts 'hola world'
=======
  puts 'hello mundo'
>>>>>>> theirs
end
```

これは 高度なマージ手法 で使用した例と同じ内容ですが、ここでは rerere を使ってコンフリクトをもう一度解消してみましょう。

```
$ git rerere
Resolved 'hello.rb' using previous resolution.
$ cat hello.rb
#! /usr/bin/env ruby

def hello
  puts 'hola mundo'
end
```

rerere がキャッシュした解消方法で、再処理が自動的に行われたようです。結果をインデックスに追加して、リベースを先に進めましょう。

```
$ git add hello.rb
$ git rebase --continue
Applying: i18n one word
```

マージの再実行を何度も行うことがある、頻繁に master ブランチをマージせずにトピックブランチを最新の状態に保ちたい、リベースをよく行う……いずれかに当てはまる場合は rerere を有効にしておきましょう。日々の生活がちょっとだけ楽になると思います。

# Git によるデバッグ

Git には、プロジェクトで発生した問題をデバッグするためのツールも用意されています。 Git はほとんどあらゆる種類のプロジェクトで使えるように設計されているので、このツールも非常に汎用的なものです。しかし、バグを見つけたり不具合の原因を探したりするための助けとなるでしょう。

## ファイルの注記

コードのバグを追跡しているときに「それが、いつどんな理由で追加されたのか」が知りたくなることがある

でしょう。そんな場合にもっとも便利なのが、ファイルの注記です。
これは、ファイルの各行について、その行を最後に更新したのがどのコミットかを表示します。 もしコードの中の特定のメソッドにバグがあることを見つけたら、そのファイルを `git blame` しましょう。そうすれば、そのメソッドの各行がいつ誰によって更新されたのかがわかります。 この例では、`-L` オプションを使って 12 行目から 22 行目までに出力を限定しています。

```
$ git blame -L 12,22 simplegit.rb
^4832fe2 (Scott Chacon  2008-03-15 10:31:28 -0700 12)  def show(tree =
'master')
^4832fe2 (Scott Chacon  2008-03-15 10:31:28 -0700 13)   command("git show
#{tree}")
^4832fe2 (Scott Chacon  2008-03-15 10:31:28 -0700 14)  end
^4832fe2 (Scott Chacon  2008-03-15 10:31:28 -0700 15)
9f6560e4 (Scott Chacon  2008-03-17 21:52:20 -0700 16)  def log(tree =
'master')
79eaf55d (Scott Chacon  2008-04-06 10:15:08 -0700 17)   command("git log
#{tree}")
9f6560e4 (Scott Chacon  2008-03-17 21:52:20 -0700 18)  end
9f6560e4 (Scott Chacon  2008-03-17 21:52:20 -0700 19)
42cf2861 (Magnus Chacon 2008-04-13 10:45:01 -0700 20)  def blame(path)
42cf2861 (Magnus Chacon 2008-04-13 10:45:01 -0700 21)   command("git blame
#{path}")
42cf2861 (Magnus Chacon 2008-04-13 10:45:01 -0700 22)  end
```

最初の項目は、その行を最後に更新したコミットの SHA-1 の一部です。 次のふたつの項目は、そのコミットから抽出した作者情報とコミット日時です。これで、いつ誰がその行を更新したのかが簡単にわかります。それに続いて、行番号とファイルの中身が表示されます。 `^4832fe2` のコミットに関する行に注目しましょう。これらの行は、ファイルが最初にコミットされたときのままであることを表します。 このコミットはファイルがプロジェクトに最初に追加されたときのものであり、これらの行はそれ以降変更されていません。これはちょっと戸惑うかも知れません。Git では、これまで紹介してきただけで少なくとも三種類以上の意味で `^` を使っていますからね。しかし、ここではそういう意味になるのです。

Git のすばらしいところのひとつに、ファイルのリネームを明示的には追跡しないということがあります。 スナップショットだけを記録し、もしリネームされていたのなら暗黙のうちにそれを検出します。 この機能の興味深いところは、ファイルのリネームだけでなくコードの移動についても検出できるということです。 `git blame` に `-C` を渡すと Git はそのファイルを解析し、別のところからコピーされたコード片がないかどうかを探します。 例えば、`GITServerHandler.m` というファイルをリファクタリングで複数のファイルに分割したとしましょう。そのうちのひとつが `GITPackUpload.m` です。 ここで `-C` オプションをつけて `GITPackUpload.m` を調べると、コードのどの部分をどのファイルからコピーしたのかを知ることができます。

```
$ git blame -C -L 141,153 GITPackUpload.m
f344f58d GITServerHandler.m (Scott 2009-01-04 141)
f344f58d GITServerHandler.m (Scott 2009-01-04 142) - (void)
gatherObjectShasFromC
f344f58d GITServerHandler.m (Scott 2009-01-04 143) {
70befddd GITServerHandler.m (Scott 2009-03-22 144)
//NSLog(@"GATHER COMMI
ad11ac80 GITPackUpload.m    (Scott 2009-03-24 145)
ad11ac80 GITPackUpload.m    (Scott 2009-03-24 146)         NSString
*parentSha;
ad11ac80 GITPackUpload.m    (Scott 2009-03-24 147)         GITCommit
*commit = [g
ad11ac80 GITPackUpload.m    (Scott 2009-03-24 148)
ad11ac80 GITPackUpload.m    (Scott 2009-03-24 149)
//NSLog(@"GATHER COMMI
ad11ac80 GITPackUpload.m    (Scott 2009-03-24 150)
56ef2caf GITServerHandler.m (Scott 2009-01-05 151)         if(commit) {
56ef2caf GITServerHandler.m (Scott 2009-01-05 152)
[refDict setOb
56ef2caf GITServerHandler.m (Scott 2009-01-05 153)
```

これはほんとうに便利です。 通常は、そのファイルがコピーされたときのコミットを知ることになります。 コピー先のファイルにおいて最初にその行をさわったのが、その内容をコピーしてきたときだからです。 Git は、その行が本当に書かれたコミットがどこであったのかを (たとえ別のファイルであったとしても) 教えてくれるのです。

## 二分探索

ファイルの注記を使えば、その問題がどの時点で始まったのかを知ることができます。 何がおかしくなったのかがわからず、最後にうまく動作していたときから何十何百ものコミットが行われている場合などは、`git bisect` に頼ることになるでしょう。 `bisect` コマンドはコミットの歴史に対して二分探索を行い、どのコミットで問題が混入したのかを可能な限り手早く見つけ出せるようにします。

自分のコードをリリースして運用環境にプッシュしたあとに、バグ報告を受け取ったと仮定しましょう。そのバグは開発環境では再現せず、なぜそんなことになるのか想像もつきません。 コードをよく調べて問題を再現させることはできましたが、何が悪かったのかがわかりません。 こんな場合に、二分探索で原因を特定することができます。 まず、`git bisect start` を実行します。そして次に `git bisect bad` を使って、現在のコミットが壊れた状態であることをシステムに伝えます。 次に、まだ壊れていなかったとわかっている直近のコミットを `git bisect good [good_commit]` で伝えます。

```
$ git bisect start
$ git bisect bad
$ git bisect good v1.0
Bisecting: 6 revisions left to test after this
[ecb6e1bc347ccecc5f9350d878ce677feb13d3b2] error handling on repo
```

Git は、まだうまく動いていたと指定されたコミット (v1.0) と現在の壊れたバージョンの間には 12 のコミットがあるということを検出しました。そして、そのちょうど真ん中にあるコミットをチェックアウトしました。 ここでテストを実行すれば、このコミットで同じ問題が発生するかどうかがわかります。 もし問題が発生したなら、実際に問題が混入したのはそれより前のコミットだということになります。そうでなければ、それ以降のコミットで問題が混入したのでしょう。 ここでは、問題が発生しなかったものとします。`git bisect good` で Git にその旨を伝え、旅を続けましょう。

```
$ git bisect good
Bisecting: 3 revisions left to test after this
[b047b02ea83310a70fd603dc8cd7a6cd13d15c04] secure this thing
```

また別のコミットがやってきました。先ほど調べたコミットと「壊れている」と伝えたコミットの真ん中にあるものです。 ふたたびテストを実行し、今度はこのコミットで問題が再現したものとします。それを Git に伝えるには `git bisect bad` を使います。

```
$ git bisect bad
Bisecting: 1 revisions left to test after this
[f71ce38690acf49c1f3c9bea38e09d82a5ce6014] drop exceptions table
```

このコミットはうまく動きました。というわけで、問題が混入したコミットを特定するための情報がこれですべて整いました。 Git は問題が混入したコミットの SHA-1 を示し、そのコミット情報とどのファイルが変更されたのかを表示します。これを使って、いったい何が原因でバグが発生したのかを突き止めます。

```
$ git bisect good
b047b02ea83310a70fd603dc8cd7a6cd13d15c04 is first bad commit
commit b047b02ea83310a70fd603dc8cd7a6cd13d15c04
Author: PJ Hyett <pjhyett@example.com>
Date:   Tue Jan 27 14:48:32 2009 -0800

    secure this thing

:040000 040000 40ee3e7821b895e52c1695092db9bdc4c61d1730
f24d3c6ebcfc639b1a3814550e62d60b8e68a8e4 M  config
```

原因がわかったら、作業を始める前に `git bisect reset` を実行して HEAD を作業前の状態に戻さなければなりません。そうしないと面倒なことになってしまいます。

```
$ git bisect reset
```

この強力なツールを使えば、何百ものコミットの中からバグの原因となるコミットを数分で見つけだせるようになります。 実際、プロジェクトが正常なときに 0 を返してどこかおかしいときに 0 以外を返すスクリプトを用意しておけば、`git bisect` を完全に自動化することもできます。 まず、先ほどと同じく、壊れているコミットと正しく動作しているコミットを指定します。 これは `bisect start` コマンドで行うこともできま

す。まず最初に壊れているコミット、そしてその後に正しく動作しているコミットを指定します。

```
$ git bisect start HEAD v1.0
$ git bisect run test-error.sh
```

こうすると、チェックアウトされたコミットに対して自動的に `test-error.sh` を実行し、壊れる原因となるコミットを見つけ出すまで自動的に処理を続けます。 `make` や `make tests`、その他自動テストを実行するためのプログラムなどをここで実行させることもできます。

# サブモジュール

あるプロジェクトで作業をしているときに、プロジェクト内で別のプロジェクトを使わなければならなくなることがよくあります。 サードパーティが開発しているライブラリや、自身が別途開発していて複数の親プロジェクトから利用しているライブラリなどがそれにあたります。 こういったときに出てくるのが「ふたつのプロジェクトはそれぞれ別のものとして管理したい。だけど、一方を他方の一部としても使いたい」という問題です。

例を考えてみましょう。ウェブサイトを制作しているあなたは、Atom フィードを作成することになりました。 Atom 生成コードを自前で書くのではなく、ライブラリを使うことに決めました。 この場合、CPAN や gem などの共有ライブラリからコードをインクルードするか、ソースコードそのものをプロジェクトのツリーに取り込むかのいずれかが必要となります。 ライブラリをインクルードする方式の問題は、ライブラリのカスタマイズが困難であることと配布が面倒になるということです。すべてのクライアントにそのライブラリを導入させなければなりません。 コードをツリーに取り込む方式の問題は、手元でコードに手を加えてしまうと本家の更新に追従しにくくなるということです。

Git では、サブモジュールを使ってこの問題に対応します。 サブモジュールを使うと、ある Git リポジトリを別の Git リポジトリのサブディレクトリとして扱うことができるようになります。 これで、別のリポジトリをプロジェクト内にクローンしても自分のコミットは別管理とすることができるようになります。

## サブモジュールの作り方

まずは単純な事例を見ていきましょう。大きな1プロジェクトを、メインの1プロジェクトとサブの複数プロジェクトに分割して開発していているとします。

開発を始めるにあたり、作業中のリポジトリのサブモジュールとして既存のリポジトリを追加します。サブモジュールを新たに追加するには `git submodule add` コマンドを実行します。追跡したいプロジェクトの URL （絶対・相対のいずれも可）を引数に指定してください。この例では、“DbConnector” というライブラリを追加してみます。

```
$ git submodule add https://github.com/chaconinc/DbConnector
Cloning into 'DbConnector'...
remote: Counting objects: 11, done.
remote: Compressing objects: 100% (10/10), done.
remote: Total 11 (delta 0), reused 11 (delta 0)
Unpacking objects: 100% (11/11), done.
Checking connectivity... done.
```

デフォルトでは、このコマンドで指定したリポジトリと同名のディレクトリに、サブプロジェクトのデータが格納されます。他のディレクトリを使いたい場合は、コマンドの末尾にパスを追加してください。

ここで `git status` を実行してみましょう。いくつか気づくことがあるはずです。

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

Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   .gitmodules
    new file:   DbConnector
```

まず気づくのが、新たに追加された `.gitmodules` ファイルです。 この設定ファイルには、プロジェクトの URL とそれを取り込んだローカルサブディレクトリの対応が格納されています。

```
[submodule "DbConnector"]
    path = DbConnector
    url = https://github.com/chaconinc/DbConnector
```

複数のサブモジュールを追加した場合は、このファイルに複数のエントリが書き込まれます。 このファイルもまた他のファイルと同様にバージョン管理下に置かれることに注意しましょう。`.gitignore` ファイルと同じことです。 プロジェクトの他のファイルと同様、このファイルもプッシュやプルの対象となります。 プロジェクトをクローンした人は、このファイルを使ってサブモジュールの取得元を知ることになります。

**NOTE**

`.gitmodules` ファイルに記述された URL を他の利用者はまずクローン/フェッチしようとします。よって、可能であればそういった人たちもアクセスできる URL を使うようにしましょう。もし、自分がプッシュする URL と他の利用者がプルする URL が違う場合は、他の利用者もアクセスできる URL をここでは使ってください。そのうえで、`git config submodule.DbConnector.url PRIVATE_URL` コマンドを使って自分用の URL を手元の環境に設定するのがいいでしょう。 可能であれば、相対 URL にしておくと便利だと思います。

また、`git status` の出力にプロジェクトフォルダも含まれています。 これに対して `git diff` を実行する

と、ちょっと興味深い結果が得られます。

```
$ git diff --cached DbConnector
diff --git a/DbConnector b/DbConnector
new file mode 160000
index 0000000..c3f01dc
--- /dev/null
+++ b/DbConnector
@@ -0,0 +1 @@
+Subproject commit c3f01dc8862123d317dd46284b05b6892c7b29bc
```

DbConnector は作業ディレクトリ内にあるサブディレクトリですが、Git はそれがサブモジュールであるとみなし、あなたがそのディレクトリにいない限りその中身を追跡することはありません。 そのかわりに、Git はこのサブディレクトリを元のプロジェクトの特定のコミットとして記録します。

差分表示をもうすこしちゃんとさせたいのなら、git diff コマンドの --submodule オプションを使いましょう。

```
$ git diff --cached --submodule
diff --git a/.gitmodules b/.gitmodules
new file mode 100644
index 0000000..71fc376
--- /dev/null
+++ b/.gitmodules
@@ -0,0 +1,3 @@
+[submodule "DbConnector"]
+       path = DbConnector
+       url = https://github.com/chaconinc/DbConnector
Submodule DbConnector 0000000...c3f01dc (new submodule)
```

コミットすると、このようになります。

```
$ git commit -am 'added DbConnector module'
[master fb9093c] added DbConnector module
 2 files changed, 4 insertions(+)
 create mode 100644 .gitmodules
 create mode 160000 DbConnector
```

DbConnector エントリのモードが 160000 となったことに注目しましょう。 これは Git における特別なモードで、サブディレクトリやファイルではなくディレクトリエントリとしてこのコミットを記録したことを意味します。

### サブモジュールを含むプロジェクトのクローン

ここでは、内部にサブモジュールを含むプロジェクトをクローンしてみます。 デフォルトでは、サブモジ

ュールを含むディレクトリは取得できますがその中にはまだ何もファイルが入っていません。

```
$ git clone https://github.com/chaconinc/MainProject
Cloning into 'MainProject'...
remote: Counting objects: 14, done.
remote: Compressing objects: 100% (13/13), done.
remote: Total 14 (delta 1), reused 13 (delta 0)
Unpacking objects: 100% (14/14), done.
Checking connectivity... done.
$ cd MainProject
$ ls -la
total 16
drwxr-xr-x   9 schacon  staff  306 Sep 17 15:21 .
drwxr-xr-x   7 schacon  staff  238 Sep 17 15:21 ..
drwxr-xr-x  13 schacon  staff  442 Sep 17 15:21 .git
-rw-r--r--   1 schacon  staff   92 Sep 17 15:21 .gitmodules
drwxr-xr-x   2 schacon  staff   68 Sep 17 15:21 DbConnector
-rw-r--r--   1 schacon  staff  756 Sep 17 15:21 Makefile
drwxr-xr-x   3 schacon  staff  102 Sep 17 15:21 includes
drwxr-xr-x   4 schacon  staff  136 Sep 17 15:21 scripts
drwxr-xr-x   4 schacon  staff  136 Sep 17 15:21 src
$ cd DbConnector/
$ ls
$
```

`DbConnector` ディレクトリは存在しますが、中身が空っぽです。 ここで、ふたつのコマンドを実行しなければなりません。まず `git submodule init` でローカルの設定ファイルを初期化し、次に `git submodule update` でプロジェクトからのデータを取得し、親プロジェクトで指定されている適切なコミットをチェックアウトします。

```
$ git submodule init
Submodule 'DbConnector' (https://github.com/chaconinc/DbConnector)
registered for path 'DbConnector'
$ git submodule update
Cloning into 'DbConnector'...
remote: Counting objects: 11, done.
remote: Compressing objects: 100% (10/10), done.
remote: Total 11 (delta 0), reused 11 (delta 0)
Unpacking objects: 100% (11/11), done.
Checking connectivity... done.
Submodule path 'DbConnector': checked out
'c3f01dc8862123d317dd46284b05b6892c7b29bc'
```

これで、サブディレクトリ `DbConnector` の中身が先ほどコミットしたときとまったく同じ状態になりました。

また、これをもうすこし簡単に済ませるには、`git clone` コマンドの `--recursive` オプションを使いましょう。そうすると、リポジトリ内のサブモジュールをすべて初期化し、データを取得してくれます。

```
$ git clone --recursive https://github.com/chaconinc/MainProject
Cloning into 'MainProject'...
remote: Counting objects: 14, done.
remote: Compressing objects: 100% (13/13), done.
remote: Total 14 (delta 1), reused 13 (delta 0)
Unpacking objects: 100% (14/14), done.
Checking connectivity... done.
Submodule 'DbConnector' (https://github.com/chaconinc/DbConnector)
registered for path 'DbConnector'
Cloning into 'DbConnector'...
remote: Counting objects: 11, done.
remote: Compressing objects: 100% (10/10), done.
remote: Total 11 (delta 0), reused 11 (delta 0)
Unpacking objects: 100% (11/11), done.
Checking connectivity... done.
Submodule path 'DbConnector': checked out
'c3f01dc8862123d317dd46284b05b6892c7b29bc'
```

## サブモジュールを含むプロジェクトでの作業

さて、サブモジュールを含むプロジェクトのデータをコピーできましたので、メインとサブ、両方のプロジェクトでの共同作業をしてみましょう。

### 上流の変更の取り込み

まずはサブモジュールの使用例で一番シンプルなモデルを見ていきます。それは、サブプロジェクトをただ単に使うだけ、というモデルです。上流の更新はときどき取り込みたいけれど、チェックアウトした内容を変更したりはしない、という使い方になります。

サブモジュールが更新されているかどうかを調べるには、サブモジュールのディレクトリで `git fetch` を実行します。併せて `git merge` で上流のブランチをマージすれば、チェックアウトしてあるコードを更新できます。

```
$ git fetch
From https://github.com/chaconinc/DbConnector
   c3f01dc..d0354fc  master     -> origin/master
$ git merge origin/master
Updating c3f01dc..d0354fc
Fast-forward
 scripts/connect.sh | 1 +
 src/db.c           | 1 +
 2 files changed, 2 insertions(+)
```

ここでメインプロジェクトのディレクトリに戻って `git diff --submodule` を実行してみてください。サブモジュールが更新されたこと、どのコミットがサブモジュールに追加されたかがわかるでしょう。なお、`git diff` の `--submodule` オプションを省略したい場合は、設定項目 `diff.submodule` の値に “log” を指定してください。

```
$ git config --global diff.submodule log
$ git diff
Submodule DbConnector c3f01dc..d0354fc:
  > more efficient db routine
  > better connection routine
```

この状態でコミットしておけば、他の人がサブモジュールを更新したときに新しい内容が取り込まれるようになります。

サブモジュールのディレクトリでのフェッチとマージを手動で行いたくない人のために、もう少し簡単な方法も紹介しておきます。`git submodule update --remote` です。これを使えば、ディレクトリに入ってフェッチしてマージして、という作業がコマンドひとつで済みます。

```
$ git submodule update --remote DbConnector
remote: Counting objects: 4, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 4 (delta 2), reused 4 (delta 2)
Unpacking objects: 100% (4/4), done.
From https://github.com/chaconinc/DbConnector
   3f19983..d0354fc  master     -> origin/master
Submodule path 'DbConnector': checked out
'd0354fc054692d3906c85c3af05ddce39a1c0644'
```

なお、このコマンドはデフォルトでは、サブモジュールのリポジトリの `master` ブランチの内容まで手元にチェックアウトした内容を更新する、という前提で動作します。ですが、そうならないよう設定することもできます。たとえば、DbConnector サブモジュールを “stable” ブランチに追従させたいとしましょう。その場合、`.gitmodules` ファイルに記述することもできますし（そうすれば、みんなが同じ設定を共有できます）、手元の `.git/config` ファイルに記述しても構いません。以下は `.gitmodules` に記述した場合の例です。

```
$ git config -f .gitmodules submodule.DbConnector.branch stable

$ git submodule update --remote
remote: Counting objects: 4, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 4 (delta 2), reused 4 (delta 2)
Unpacking objects: 100% (4/4), done.
From https://github.com/chaconinc/DbConnector
   27cf5d3..c87d55d  stable -> origin/stable
Submodule path 'DbConnector': checked out
'c87d55d4c6d4b05ee34fbc8cb6f7bf4585ae6687'
```

また、この設定コマンドから `-f .gitmodules` の部分を除くと、設定は手元の環境に対してのみ反映されます。ただ、この設定はリポジトリにコミットして追跡しておくほうがよいと思います。関係者全員が同じ設定を共有できるからです。

ここで `git status` を実行すると、「新しいコミット」（“new commits”）がサブモジュールに追加されたことがわかります。

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

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

  modified:   .gitmodules
  modified:   DbConnector (new commits)

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

さらに、設定項目 `status.submodulesummary` を指定しておけば、リポジトリ内のサブモジュールの変更点の要約も確認できます。

```
$ git config status.submodulesummary 1

$ git status
On branch master
Your branch is up-to-date with 'origin/master'.

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

    modified:   .gitmodules
    modified:   DbConnector (new commits)

Submodules changed but not updated:

* DbConnector c3f01dc...c87d55d (4):
  > catch non-null terminated lines
```

この段階で`git diff`を実行すると、`.gitmodules`ファイルが変更されていることがわかります。また、サブモジュールについては、上流からコミットがすでにいくつも取得されていて、手元のリポジトリでコミット待ちの状態になっていることがわかります。

```
$ git diff
diff --git a/.gitmodules b/.gitmodules
index 6fc0b3d..fd1cc29 100644
--- a/.gitmodules
+++ b/.gitmodules
@@ -1,3 +1,4 @@
 [submodule "DbConnector"]
        path = DbConnector
        url = https://github.com/chaconinc/DbConnector
+       branch = stable
 Submodule DbConnector c3f01dc..c87d55d:
  > catch non-null terminated lines
  > more robust error handling
  > more efficient db routine
  > better connection routine
```

手元のサブモジュールにこれから何をコミットしようとしているのかがわかるので、これはとても便利です。また、実際にコミットしたあとでも、`git log -p`を使えばこの情報は確認できます。

```
$ git log -p --submodule
commit 0a24cfc121a8a3c118e0105ae4ae4c00281cf7ae
Author: Scott Chacon <schacon@gmail.com>
Date:   Wed Sep 17 16:37:02 2014 +0200

    updating DbConnector for bug fixes

diff --git a/.gitmodules b/.gitmodules
index 6fc0b3d..fd1cc29 100644
--- a/.gitmodules
+++ b/.gitmodules
@@ -1,3 +1,4 @@
 [submodule "DbConnector"]
        path = DbConnector
        url = https://github.com/chaconinc/DbConnector
+       branch = stable
Submodule DbConnector c3f01dc..c87d55d:
  > catch non-null terminated lines
  > more robust error handling
  > more efficient db routine
  > better connection routine
```

なお、`git submodule update --remote` を実行すると、デフォルトではすべてのサブモジュールの更新が行われます。よって、サブモジュールが多い場合は更新したいものだけを指定するとよいでしょう。

**サブモジュールでの作業**

サブモジュールを使う動機を考えてみましょう。その多くは、メインプロジェクトで（あるいは複数のサブモジュールに渡って）作業をしつつ、サブモジュールのコードも変更したいから、だと思います。というのも、そうでなければ Maven や Rubygems のようなシンプルな依存関係管理の仕組みを使っているはずだからです。

ということでここでは、メインプロジェクトとサブモジュールを行ったり来たりしながら変更を加えていく方法を見ていきましょう。併せて、それらを同時にコミット/公開する方法も紹介します。

これまでの例では、`git submodule update` コマンドを実行してサブモジュールのリモートリポジトリの変更内容を取得すると、サブモジュール用ディレクトリ内のファイルは更新されますが、手元のサブモジュール用リポジトリの状態は「切り離された HEAD (detached HEAD)」になってしまっていました。つまり、作業中のブランチ（“master” など）は存在せず、変更も追跡されない、ということです。 このままでは、たとえサブモジュールになにかコミットを追加したとしても、`git submodule update`を実行したタイミングで追加した内容はなくなってしまうことになります。そういった事態を避け、サブモジュールに追加した内容をちゃんと記録するには、事前準備が必要なのです。

では、どうすればサブモジュールをハックしやすくなるでしょうか。やるべきことは2つです。まず、サブモジュール用のディレクトリで、作業用のブランチをチェックアウトしましょう。次に、何らかの変更をサブモジュールに加えたあとに `git submodule update --remote` を実行して上流から変更をプルした場合の挙動を設定します。手元の変更内容に上流の変更をマージするか、手元の変更内容を上流の変更にリベースする

かのいずれかを選択することになります。

実際にやってみましょう。まず、サブモジュール用のディレクトリに入って、作業用のブランチをチェックアウトします。

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

次の手順ですが、ここでは「マージ」することにします.実施のたびに指定するのであれば、`update` コマンド実行時に `--merge` オプションを使います。以下の例では、サーバーにあるサブモジュールのデータは変更されていて、それがマージされていることがわかります。

```
$ git submodule update --remote --merge
remote: Counting objects: 4, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 4 (delta 2), reused 4 (delta 2)
Unpacking objects: 100% (4/4), done.
From https://github.com/chaconinc/DbConnector
   c87d55d..92c7337  stable     -> origin/stable
Updating c87d55d..92c7337
Fast-forward
 src/main.c | 1 +
 1 file changed, 1 insertion(+)
Submodule path 'DbConnector': merged in
'92c7337b30ef9e0893e758dac2459d07362ab5ea'
```

DbConnector ディレクトリを見ると、上流の変更が手元の `stable` ブランチに取り込み済みであるとわかります。では次に、手元のファイルに変更を加えている間に、別の変更が上流にプッシュされたらどうなるかを説明しましょう。

```
$ cd DbConnector/
$ vim src/db.c
$ git commit -am 'unicode support'
[stable f906e16] unicode support
 1 file changed, 1 insertion(+)
```

この段階でサブモジュールを更新してみましょう。手元のファイルは変更済みで、上流にある別の変更も取り込む必要がある場合、何が起こるかがわかるはずです。

```
$ git submodule update --remote --rebase
First, rewinding head to replay your work on top of it...
Applying: unicode support
Submodule path 'DbConnector': rebased into
'5d60ef9bbebf5a0c1c1050f242ceeb54ad58da94'
```

`--rebase` や `--merge` オプションを付け忘れると、サブモジュールはサーバー上の状態で上書きされ、「切り離された HEAD」状態になります。

```
$ git submodule update --remote
Submodule path 'DbConnector': checked out
'5d60ef9bbebf5a0c1c1050f242ceeb54ad58da94'
```

ただ、こうなってしまっても慌てる必要はありません。サブモジュールのディレクトリに戻れば、変更を追加したブランチをチェックアウトできます。そのうえで、`origin/stable`（などの必要なリモートブランチ）を手動でマージなりリベースなりすればよいのです。

また、手元で加えた変更をコミットしていない状態でサブモジュールを更新したとしましょう。これは問題になりそうですが、実際はそうなりません。リモートの変更だけが取得され、サブモジュール用ディレクトリに加えた変更でコミットしていないものはそのまま残ります。

```
$ git submodule update --remote
remote: Counting objects: 4, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 4 (delta 0), reused 4 (delta 0)
Unpacking objects: 100% (4/4), done.
From https://github.com/chaconinc/DbConnector
   5d60ef9..c75e92a  stable     -> origin/stable
error: Your local changes to the following files would be overwritten by
checkout:
    scripts/setup.sh
Please, commit your changes or stash them before you can switch branches.
Aborting
Unable to checkout 'c75e92a2b3855c9e5b66f915308390d9db204aca' in submodule
path 'DbConnector'
```

手元で加えた変更が上流の変更とコンフリクトする場合は、サブモジュール更新を実施したときにわかるようになっています。 If you made changes that conflict with something changed upstream, Git will let you know when you run the update.

```
$ git submodule update --remote --merge
Auto-merging scripts/setup.sh
CONFLICT (content): Merge conflict in scripts/setup.sh
Recorded preimage for 'scripts/setup.sh'
Automatic merge failed; fix conflicts and then commit the result.
Unable to merge 'c75e92a2b3855c9e5b66f915308390d9db204aca' in submodule
path 'DbConnector'
```

そうなったら、サブモジュール用ディレクトリのファイルを編集しましょう。いつものようにコンフリクトを解消できます。

#### サブモジュールに加えた変更の公開

これまでの作業で、サブモジュール用ディレクトリの内容は変更されています。上流の変更を取り込みましたし、手元でも変更を加えました。そして、後者の存在は誰もまだ知りません。プッシュされていないからです。

```
$ git diff
Submodule DbConnector c87d55d..82d2ad3:
  > Merge from origin/stable
  > updated setup script
  > unicode support
  > remove unnecessary method
  > add new option for conn pooling
```

メインプロジェクトに変更をコミットしてプッシュしたけれど、サブモジュールの変更はプッシュしていないとします。その場合、プッシュされたリポジトリをチェックアウトしようとしてもうまくいかないでしょう。メインプロジェクトの変更が依存しているサブモジュールの変更を、取得する手段がないからです。必要とされる変更内容は、手元の環境にしかありません。

こういった状態にならないよう、サブモジュールの変更がプッシュ済みかどうかを事前に確認する方法があります。メインプロジェクトをプッシュするときに使う `git push` コマンドの、 `--recurse-submodules` オプションです。 これを “check” か “on-demand” のいずれかに設定します。“check” に設定すれば、サブモジュールの変更でプッシュされていないものがある場合、メインプロジェクトのプッシュは失敗するようになります。

```
$ git push --recurse-submodules=check
The following submodule paths contain changes that can
not be found on any remote:
  DbConnector

Please try

    git push --recurse-submodules=on-demand

or cd to the path and use

    git push

to push them to a remote.
```

ご覧のとおり、事態を解決する方法もいくつか提示されます。そのなかで一番単純なのは、全サブモジュールを個別にプッシュしてまわる方法です。サブモジュールの変更が公開された状態になれば、メインプロジェクトのプッシュもうまくいくでしょう。

他にも、このオプションを “on-demand” に設定する方法があります。そうすると、さきほど「単純」といった手順をすべて実行してくれます。

```
$ git push --recurse-submodules=on-demand
Pushing submodule 'DbConnector'
Counting objects: 9, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (8/8), done.
Writing objects: 100% (9/9), 917 bytes | 0 bytes/s, done.
Total 9 (delta 3), reused 0 (delta 0)
To https://github.com/chaconinc/DbConnector
   c75e92a..82d2ad3  stable -> stable
Counting objects: 2, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (2/2), 266 bytes | 0 bytes/s, done.
Total 2 (delta 1), reused 0 (delta 0)
To https://github.com/chaconinc/MainProject
   3d6d338..9a377d1  master -> master
```

そうです、DbConnector モジュールがプッシュされたあと、メインプロジェクトがプッシュされています。もしサブモジュールのプッシュが何らかの理由で失敗すれば、メインプロジェクトのプッシュも失敗するようになっています。

#### 変更されたサブモジュールのマージ

サブモジュールの参照を他の人と同じタイミングで変更してしまうと、問題になる場合があります。つまり、

サブモジュールの歴史が分岐してしまい、その状態が両者の手元にあるメインプロジェクトにコミットされ、ブランチも分岐した状態になってしまいます。これを解消するのは厄介です。

この場合でも、一方のコミットがもう一方のコミットの直系の先祖である場合、新しいほうのコミットがマージされます（fast-forward なマージ）。何も問題にはなりません。

ただし、“trivial” なマージすら行われないケースがあります。具体的には、サブモジュールのコミットが分岐してマージする必要があるようなケースです。その場合、以下のような状態になります。

```
$ git pull
remote: Counting objects: 2, done.
remote: Compressing objects: 100% (1/1), done.
remote: Total 2 (delta 1), reused 2 (delta 1)
Unpacking objects: 100% (2/2), done.
From https://github.com/chaconinc/MainProject
   9a377d1..eb974f8  master     -> origin/master
Fetching submodule DbConnector
warning: Failed to merge submodule DbConnector (merge following commits
not found)
Auto-merging DbConnector
CONFLICT (submodule): Merge conflict in DbConnector
Automatic merge failed; fix conflicts and then commit the result.
```

何が起こったのでしょうか。まず、サブモジュールの歴史の分岐点になっているブランチが2つあって、マージする必要があることがわかります。次に、“merge following commits not found” であることもわかります。え、何がわかったの？と思った方、ご安心ください。もう少し先で説明します。

この問題を解決するには、サブモジュールがどういった状態にあるべきかを把握しなければなりません。ですが、いつもとは違い、上記の Git コマンド出力からは有用な情報は得られません。分岐してしまった歴史で問題となっているコミット SHA-1 すら表示されません。ただ、ありがたいことに、それらは簡単に確認できます。`git diff` を実行してみましょう。マージしようとしていた両ブランチのコミット SHA-1 が表示されます。

```
$ git diff
diff --cc DbConnector
index eb41d76,c771610..0000000
--- a/DbConnector
+++ b/DbConnector
```

この例では、コミット `eb41d76` は **手元** のサブモジュールに追加されていたもので、コミット `c771610` は上流にあったものであることがわかります。さきほどのマージでは処理が行えなかったので、サブモジュール用ディレクトリの最新コミットは `eb41d76` のはずです。何らかの理由で仮にそうなっていなければ、そのコミットが最新になっているブランチを作成し、チェックアウトすればよいでしょう。

注目すべきは上流のコミット SHA-1 です。マージしてコンフリクトを解消しなければなりません。SHA-1 を直接指定してマージしてみてもよいですし、該当のコミットを指定して作ったブランチをマージしても構いま

せん。どちらかと言えば後者がオススメです（マージコミットのメッセージがわかりやすくなるくらいのメリットしかありませんが）。

では実際にやってみましょう。サブモジュール用ディレクトリで該当のコミット（さきほどの `git diff` の2番目の SHA-1）を指定してブランチを作り、手動でマージしてみます。

```
$ cd DbConnector

$ git rev-parse HEAD
eb41d764bccf88be77aced643c13a7fa86714135

$ git branch try-merge c771610
(DbConnector) $ git merge try-merge
Auto-merging src/main.c
CONFLICT (content): Merge conflict in src/main.c
Recorded preimage for 'src/main.c'
Automatic merge failed; fix conflicts and then commit the result.
```

そうすると、実際にどこがコンフリクトしているかがわかります。それを解決してコミットすれば、その結果をもとにメインプロジェクトがアップデートできる、というわけです。

```
$ vim src/main.c ①
$ git add src/main.c
$ git commit -am 'merged our changes'
Recorded resolution for 'src/main.c'.
[master 9fd905e] merged our changes

$ cd .. ②
$ git diff ③
diff --cc DbConnector
index eb41d76,c771610..0000000
--- a/DbConnector
+++ b/DbConnector
@@@ -1,1 -1,1 +1,1 @@@
- Subproject commit eb41d764bccf88be77aced643c13a7fa86714135
 -Subproject commit c77161012afbbe1f58b5053316ead08f4b7e6d1d
++Subproject commit 9fd905e5d7f45a0d4cbc43d1ee550f16a30e825a
$ git add DbConnector ④

$ git commit -m "Merge Tom's Changes" ⑤
[master 10d2c60] Merge Tom's Changes
```

① まずはコンフリクトを解決します

② 次にメインプロジェクトのディレクトリに戻ります

③ SHA を改めて確認します

④ コンフリクトしていたサブモジュールの登録を解決します

⑤ マージした内容をコミットします

少しややこしいかもしれませんが、そう難しくはないはずです。

また、こういったときに別の方法で処理されることもあります。 サブモジュール用ディレクトリの歴史にマージコミットがあって、上述した **両方** のコミットがすでにマージされている場合です。それを用いてもコンフリクトを解消できます。サブモジュールの歴史を確認した Git からすれば、「該当のコミットふたつが含まれたブランチを、誰かがすでにマージしてるよ。それでいいんじゃない？」というわけです。

これは、さきほど説明を省略したエラーメッセージ “merge following commits not found” の原因でもあります。1つめの例、このエラーメッセージを初めて紹介したときは **この方法** は使えなかったからです。わかりにくいのも当然で、誰もそんなことが **行われようとしてる** なんて思わないですよね。

この方法で処理するのに使えそうなマージコミットが見つかると、以下のようなメッセージが表示されます。

```
$ git merge origin/master
warning: Failed to merge submodule DbConnector (not fast-forward)
Found a possible merge resolution for the submodule:
 9fd905e5d7f45a0d4cbc43d1ee550f16a30e825a: > merged our changes
If this is correct simply add it to the index for example
by using:

  git update-index --cacheinfo 160000
9fd905e5d7f45a0d4cbc43d1ee550f16a30e825a "DbConnector"

which will accept this suggestion.
Auto-merging DbConnector
CONFLICT (submodule): Merge conflict in DbConnector
Automatic merge failed; fix conflicts and then commit the result.
```

インデックスを更新してコミットしましょう、ということのようです。`git add` コマンドを実行してコミットを解消するのと同じですね。ただ、素直にそうするのはやめておいたほうがよさそうです。その代わり、サブモジュール用ディレクトリの差分を確認し、指示されたコミットまで fast-forward すればいいでしょう。そうすれば、きちんとテストしてからコミットできます。

```
$ cd DbConnector/
$ git merge 9fd905e
Updating eb41d76..9fd905e
Fast-forward

$ cd ..
$ git add DbConnector
$ git commit -am 'Fast forwarded to a common submodule child'
```

この方法でも処理結果は代わりません。そのうえ、きちんと動作するか確認できますし、作業が終わった後に

もサブモジュール用ディレクトリにはコードが残ることになります。

## サブモジュールのヒント

サブモジュールを使った作業の難しさを和らげてくれるヒントをいくつか紹介します。

### Submodule Foreach

submodule `foreach` コマンドを使うと、サブモジュールごとに任意のコードを実行してくれます。たくさんのサブモジュールをプロジェクトで使っていれば、便利だと思います。

例えば、新機能の開発やバグ修正を着手したいとします。ただし、使っているサブモジュールに加えた変更がまだコミットされていません。この場合、そのコミットされていない状態は簡単に隠しておけます。

```
$ git submodule foreach 'git stash'
Entering 'CryptoLibrary'
No local changes to save
Entering 'DbConnector'
Saved working directory and index state WIP on stable: 82d2ad3 Merge from
origin/stable
HEAD is now at 82d2ad3 Merge from origin/stable
```

うまく隠せたら、全サブモジュールで新しいブランチを作ってチェックアウトします。

```
$ git submodule foreach 'git checkout -b featureA'
Entering 'CryptoLibrary'
Switched to a new branch 'featureA'
Entering 'DbConnector'
Switched to a new branch 'featureA'
```

どうでしょう、簡単だと思いませんか。他にも、メインプロジェクトとサブプロジェクトの変更内容の差分をユニファイド形式でとることも可能です。これもとても便利です。

```
$ git diff; git submodule foreach 'git diff'
Submodule DbConnector contains modified content
diff --git a/src/main.c b/src/main.c
index 210f1ae..1f0acdc 100644
--- a/src/main.c
+++ b/src/main.c
@@ -245,6 +245,8 @@ static int handle_alias(int *argcp, const char
***argv)

      commit_pager_choice();

+     url = url_decode(url_orig);
+
      /* build alias_argv */
      alias_argv = xmalloc(sizeof(*alias_argv) * (argc + 1));
      alias_argv[0] = alias_string + 1;
Entering 'DbConnector'
diff --git a/src/db.c b/src/db.c
index 1aaefb6..5297645 100644
--- a/src/db.c
+++ b/src/db.c
@@ -93,6 +93,11 @@ char *url_decode_mem(const char *url, int len)
        return url_decode_internal(&url, len, NULL, &out, 0);
 }

+char *url_decode(const char *url)
+{
+       return url_decode_mem(url, strlen(url));
+}
+
 char *url_decode_parameter_name(const char **query)
 {
        struct strbuf out = STRBUF_INIT;
```

この例では、サブモジュールで関数が定義され、メインプロジェクトでそれを呼び出していることがわかります。簡易な例ではありますが、どんなふうに便利なのかわかったかと思います。

#### 便利なエイリアス

紹介してきたコマンドの一部には、エイリアスを設定しておくとよいかもしれません。長いものが多いですし、紹介した挙動がデフォルトになるようには設定できないものが大半だからです。Git でエイリアスを設定する方法は Git エイリアス で触れましたが、ここでも設定例を紹介しておきます。Git のサブモジュール機能を多用する場合は、参考にしてみてください。

```
$ git config alias.sdiff '!'"git diff && git submodule foreach 'git diff'"
$ git config alias.spush 'push --recurse-submodules=on-demand'
$ git config alias.supdate 'submodule update --remote --merge'
```

このように設定しておくと、`git supdate` コマンドを実行すればサブモジュールが更新されるようになります。同様に、`git spush` コマンドであれば、サブモジュールの依存関係をチェックしたあとでプッシュするようになります。

### サブモジュール使用時に気をつけるべきこと

しかし、サブモジュールを使っているとなにかしらちょっとした問題が出てくるものです。

例えば、サブモジュールを含むブランチを切り替えるのは、これまた用心が必要です。 新しいブランチを作成してそこにサブモジュールを追加し、サブモジュールを含まないブランチに戻ったとしましょう。そこには、サブモジュールのディレクトリが「追跡されていないディレクトリ」として残ったままになります。

```
$ git checkout -b add-crypto
Switched to a new branch 'add-crypto'

$ git submodule add https://github.com/chaconinc/CryptoLibrary
Cloning into 'CryptoLibrary'...
...

$ git commit -am 'adding crypto library'
[add-crypto 4445836] adding crypto library
 2 files changed, 4 insertions(+)
 create mode 160000 CryptoLibrary

$ git checkout master
warning: unable to rmdir CryptoLibrary: Directory not empty
Switched to branch 'master'
Your branch is up-to-date with 'origin/master'.

$ git status
On branch master
Your branch is up-to-date with 'origin/master'.

Untracked files:
  (use "git add <file>..." to include in what will be committed)

    CryptoLibrary/

nothing added to commit but untracked files present (use "git add" to
track)
```

残ったディレクトリを削除するのは大変ではありませんが、そもそもそこにディレクトリが残ってしまうのは

ややこしい感じがします。実際に削除したあとに元のブランチをチェックアウトすると、モジュールを再追加するために `submodule update --init` コマンドを実行しなければなりません。

```
$ git clean -fdx
Removing CryptoLibrary/

$ git checkout add-crypto
Switched to branch 'add-crypto'

$ ls CryptoLibrary/

$ git submodule update --init
Submodule path 'CryptoLibrary': checked out
'b8dda6aa182ea4464f3f3264b11e0268545172af'

$ ls CryptoLibrary/
Makefile    includes    scripts     src
```

繰り返しになりますが、大変ではないけれどややこしい感じがしてしまいます。

次にもうひとつ、多くの人がハマるであろう点を指摘しておきましょう。これは、サブディレクトリからサブモジュールへ切り替えるときに起こることです。 プロジェクト内で追跡しているファイルをサブモジュール内に移動したくなったとしましょう。よっぽど注意しないと、Git に怒られてしまいます。 ファイルをプロジェクト内のサブディレクトリで管理しており、それをサブモジュールに切り替えたくなったとしましょう。サブディレクトリをいったん削除してから `submodule add` と実行すると、Git に怒鳴りつけられてしまいます。

```
$ rm -Rf CryptoLibrary/
$ git submodule add https://github.com/chaconinc/CryptoLibrary
'CryptoLibrary' already exists in the index
```

まず最初に `CryptoLibrary` ディレクトリをアンステージしなければなりません。 それからだと、サブモジュールを追加することができます。

```
$ git rm -r CryptoLibrary
$ git submodule add https://github.com/chaconinc/CryptoLibrary
Cloning into 'CryptoLibrary'...
remote: Counting objects: 11, done.
remote: Compressing objects: 100% (10/10), done.
remote: Total 11 (delta 0), reused 11 (delta 0)
Unpacking objects: 100% (11/11), done.
Checking connectivity... done.
```

これをどこかのブランチで行ったとしましょう。 そこから、(まだサブモジュールへの切り替えがすんでおらず実際のツリーがある状態の) 別のブランチに切り替えようとすると、このようなエラーになります。

```
$ git checkout master
error: The following untracked working tree files would be overwritten by
checkout:
  CryptoLibrary/Makefile
  CryptoLibrary/includes/crypto.h
  ...
Please move or remove them before you can switch branches.
Aborting
```

`checkout -f` を使えば、強引に切り替えられます。ただし、そうしてしまうと未保存の状態はすべて上書きされてしまいます。強引に切り替えるのであれば、すべて保存済みであることをよく確認してから実行してください。

```
$ git checkout -f master
warning: unable to rmdir CryptoLibrary: Directory not empty
Switched to branch 'master'
```

さて、戻ってきたら、なぜか `CryptoLibrary` ディレクトリは空っぽです。しかも、ここで `git submodule update` を実行しても状況は変わらないかもしれません。そんな場合は、サブモジュール用のディレクトリで `git checkout .` を実行してください。ファイルが元通りになっているはずです。サブモジュールが複数ある場合は、`submodule foreach` スクリプトを使ったこの方法を全サブモジュールに対して実行するとよいでしょう。

最後にひとつ、大事なことを説明しておきます。相当古いバージョンの Git でなければ、サブモジュール関連の Git データはメインプロジェクトの `.git` ディレクトリに保存されます。古いバージョンを使っていなければ、サブモジュール用ディレクトリを削除してもコミットやブランチのデータは残ったままです。

この節で説明したツールを使ってみてください。依存関係にある複数プロジェクトを、サブモジュールを使ってわかりやすく効率的に開発できるはずです。

## バンドルファイルの作成

Git データをネットワーク越しに転送する方法（HTTP や SSH など）についてはすでに触れましたが、まだ紹介していない方法があります。あまり使われてはいませんが、とても便利な方法です。

Git では、データを「バンドルファイル」という1つのファイルにまとめられます。これが便利な場面はいくつもあるでしょう。例えば、ネットワークが落ちていて同僚に変更を送れないような場合。あるいは、いつもとは違う場所で仕事をしていて、セキュリティ上の理由によってネットワークへのアクセスが禁止されているのかもしれません。無線/有線LAN用のカードが壊れてしまったとか。もしくは、共有サーバーにはアクセス出来ないので作業内容をメールで送りたいけれど、かといって40ものコミットを `format-patch` を使って送りたくはない、ということかもしれません。

そんなとき、`git bundle` コマンドが役に立つでしょう。このコマンドを使うと、`git push` コマンドで転送されるのと同内容のデータを単一のバイナリファイルにまとめてくれます。あとは、そのファイルをメールで送るか USB メモリに入れるなどしておいて、別のリポジトリ上で展開すればいいのです。

コミットが2つあるリポジトリを使って、簡単な例を紹介します。

```
$ git log
commit 9a466c572fe88b195efd356c3f2bbeccdb504102
Author: Scott Chacon <schacon@gmail.com>
Date:   Wed Mar 10 07:34:10 2010 -0800

    second commit

commit b1ec3248f39900d2a406049d762aa68e9641be25
Author: Scott Chacon <schacon@gmail.com>
Date:   Wed Mar 10 07:34:01 2010 -0800

    first commit
```

このリポジトリを相手に送りたいのだけど、プッシュすべきリポジトリの書き込み権限が付与されていないとしましょう（あるいは、わざわざ権限を設定したくなかったのかもしれません）。そういった場合には、`git bundle create` コマンドを使うとそのリポジトリをまとめられます。

```
$ git bundle create repo.bundle HEAD master
Counting objects: 6, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (6/6), 441 bytes, done.
Total 6 (delta 0), reused 0 (delta 0)
```

これで、`repo.bundle` というファイルが生成されました。対象リポジトリの `master` ブランチを復元できるだけのデータが含まれたファイルです。この `bundle` コマンドを使うには、まとめたい対象を範囲指定されたコミットや参照の形で指定する必要があります。クローン元となる予定であれば、HEAD を参照として追加しておくほうがよいでしょう（上記の例と同様）。

この `repo.bundle` ファイルはメールで送ってもいいですし、USB メモリに入れて持っていってもかまいません。

では、この `repo.bundle` ファイルを受け取った側はどうなるのでしょうか。該当のプロジェクトで作業をしたいとします。その場合、このバイナリファイルをディレクトリ上にクローンできます。URL を指定してクローンするのとなんら変わりありません。

```
$ git clone repo.bundle repo
Cloning into 'repo'...
...
$ cd repo
$ git log --oneline
9a466c5 second commit
b1ec324 first commit
```

まとめる対象として HEAD が含まれていないと、ここで、 `-b master` のように、なんらかのブランチを指定しなければなりません。そうしないと、どのブランチをチェックアウトすべきか、判断する術がないからです。

続いて、さきほど受け取ったリポジトリにコミットを3つ追加しました。バンドルファイルを作成して、USB メモリかメールで送り返してみましょう。

```
$ git log --oneline
71b84da last commit - second repo
c99cf5b fourth commit - second repo
7011d3d third commit - second repo
9a466c5 second commit
b1ec324 first commit
```

それには、バンドルファイルにまとめたいコミット範囲をまず決めます。ネットワークを使った方法であれば転送すべき範囲を最小限に自動で絞り込んでくれますが、ここでは手動で絞りこまねばなりません。最初にバンドルファイルを作ったときのようにリポジトリ全体をまとめてもかまいませんが、差分（この場合は追加したコミット3つ）だけをまとめるほうがよいでしょう。

そうするには、差分を割り出す必要があります。コミットの範囲指定 で解説したとおり、コミット範囲を指定する方法はたくさんあります。手元の master ブランチにはあってクローン元のブランチにはないコミット3つを指定するには、`origin/master..master` や `master ^origin/master` などとするとよいでしょう。記述をテストするには、`log` コマンドを使います。

```
$ git log --oneline master ^origin/master
71b84da last commit - second repo
c99cf5b fourth commit - second repo
7011d3d third commit - second repo
```

対象のコミットがわかったので、ひとつにまとめてみましょう。バンドルファイルのファイル名と対象のコミット範囲を指定して `git bundle create` コマンドを実行します。

```
$ git bundle create commits.bundle master ^9a466c5
Counting objects: 11, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (9/9), 775 bytes, done.
Total 9 (delta 0), reused 0 (delta 0)
```

このようにすると、リポジトリ内に `commits.bundle` ファイルが生成されます。そのファイルを送り返すと、受け取った相手は元のリポジトリにその内容を取り込めます。そのリポジトリに他の作業内容が追加されていたとしても問題にはなりません。

バンドルファイルを受け取った側は、それを検査して中身を確認できます。その後、元のリポジトリに取り込めばよいのです。そのためのコマンドが `bundle verify` で、これを実行すると、そのファイルが Git のバンドルファイルであること、そのバンドルファイルを取り込むのに必要となる祖先が手元のリポジトリにあるかどうかを検査できます。

```
$ git bundle verify ../commits.bundle
The bundle contains 1 ref
71b84daaf49abed142a373b6e5c59a22dc6560dc refs/heads/master
The bundle requires these 1 ref
9a466c572fe88b195efd356c3f2bbeccdb504102 second commit
../commits.bundle is okay
```

バンドルファイルを作る側が、追加されたコミット3つのうち2つしかバンドルファイルに含めなかったとしたらどうなるのでしょうか。その場合、元のリポジトリはそれを取り込めません。歴史を再構成するために必要なデータが揃っていないからです。もし `verify` コマンドを実行すれば、以下のようになるでしょう。

```
$ git bundle verify ../commits-bad.bundle
error: Repository lacks these prerequisite commits:
error: 7011d3d8fc200abe0ad561c011c3852a4b7bbe95 third commit - second repo
```

この例では、1つめに検査したバンドルファイルは有効だったので、コミットを取り出せます。バンドルファイルに含まれている取り込み可能なブランチを知りたければ、ブランチ参照をリストアップするためのコマンドもあります。

```
$ git bundle list-heads ../commits.bundle
71b84daaf49abed142a373b6e5c59a22dc6560dc refs/heads/master
```

`verify` サブコマンドを使っても、同様にブランチ参照をリストアップできます。大事なのは、何が取り込めるのかを確認する、ということです。そうすれば、`fetch` や `pull` コマンドを使ってバンドルファイルからコミットを取り込めるからです。ここでは、バンドルファイルの *master* ブランチを、手元のリポジトリの *other-master* ブランチに取り込んでみましょう。

```
$ git fetch ../commits.bundle master:other-master
From ../commits.bundle
 * [new branch]      master     -> other-master
```

そうすると、*master* ブランチに追加したコミットはそのままで、*other-master* ブランチ上にバンドルファイルからコミットが取り込まれていることがわかります。

```
$ git log --oneline --decorate --graph --all
* 8255d41 (HEAD, master) third commit - first repo
| * 71b84da (other-master) last commit - second repo
| * c99cf5b fourth commit - second repo
| * 7011d3d third commit - second repo
|/
* 9a466c5 second commit
* b1ec324 first commit
```

このように、データの共有やネットワークを使う作業に `git bundle` はとても便利なコマンドです。特にネットワーク環境や共有リポジトリがない状態ではそれを実感できるでしょう。

# Git オブジェクトの置き換え

Git オブジェクトは変更できません。その代わりに用意されているのが、Git データベース上のオブジェクトを他のオブジェクトと置き換えたかのように見せる方法です。

`replace` コマンドを使うと、「このオブジェクトを参照するときは、あたかもあちらを参照してるかのように振る舞え」と Git に指示できます。プロジェクトの歴史のなかで、コミットを別のコミットで置き換えたいときに便利です。

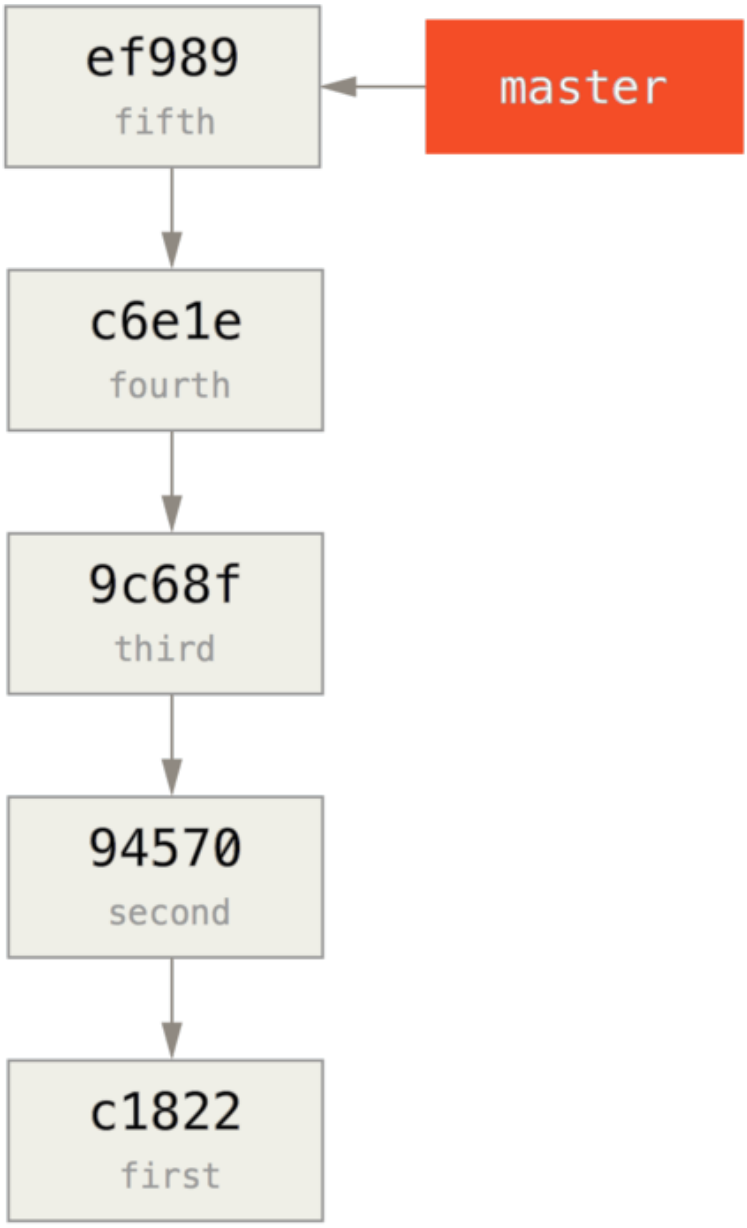
具体的な例として、長い歴史を経たコードベースがあって、それを2つに分割するケースを考えてみましょう。1つは短い歴史で新入りの開発者向け、もう1つは長い歴史でデータマイニングを行いたい人向けです。とある歴史を別の歴史と結びつけるには、新しいほうの歴史の最古のコミットを、古いほうの歴史の最新のコミットと置き換えてやればいいのです。これの利点は、そうしておけば新しいほうの歴史のコミットをすべて書き換える必要がなくなることです。通常であれば、歴史をつなぐにはそうせざるを得ません（コミットの親子関係が算出される SHA-1 に影響するため）。

では、既存のリポジトリを使って実際に試してみましょう。まずは、そのリポジトリを最近のものと過去の経緯を把握するためのものの2つに分割してみます。そのうえで、その2つを結合しつつ前者のリポジトリの SHA-1 を変更せずに済ますために `replace` を使ってみます。

ここでは、コミットが5つだけある以下のようなリポジトリを使って説明します。

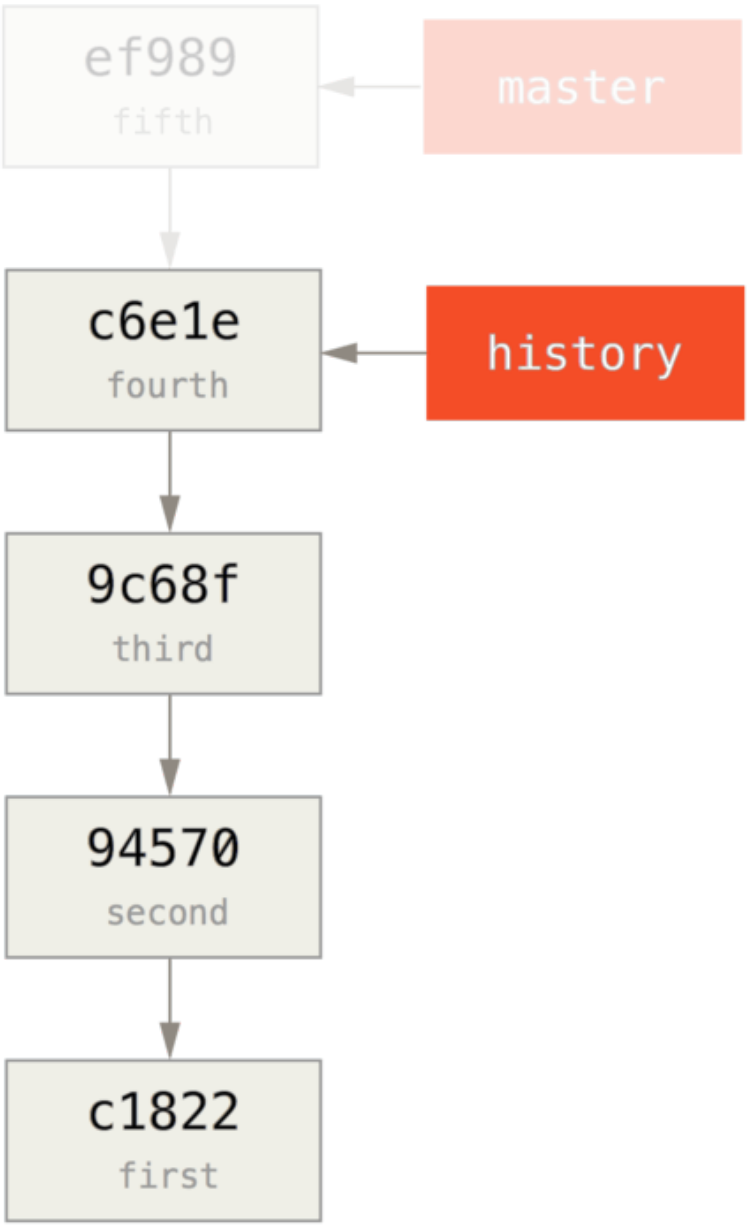
```
$ git log --oneline
ef989d8 fifth commit
c6e1e95 fourth commit
9c68fdc third commit
945704c second commit
c1822cf first commit
```

このリポジトリを2つの歴史に分割してみましょう。1つめの歴史はコミット1からコミット4までで、過去の経緯を把握するためのリポジトリです。2つめの歴史はコミット4とコミット5だけで、これは最近の歴史だけのリポジトリになります。



過去の経緯を把握するための歴史は簡単に取り出せます。過去のコミットを指定してブランチを切り、新たに作成しておいたリモートリポジトリの master としてそのブランチをプッシュすればよいのです。

```
$ git branch history c6e1e95
$ git log --oneline --decorate
ef989d8 (HEAD, master) fifth commit
c6e1e95 (history) fourth commit
9c68fdc third commit
945704c second commit
c1822cf first commit
```



作成した history ブランチを、新規リポジトリの master ブランチにプッシュします。

```
$ git remote add project-history https://github.com/schacon/project-
history
$ git push project-history history:master
Counting objects: 12, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (4/4), done.
Writing objects: 100% (12/12), 907 bytes, done.
Total 12 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (12/12), done.
To git@github.com:schacon/project-history.git
 * [new branch]      history -> master
```

これで新たに作った歴史が公開されました。続いて難しいほう、最近の歴史を小さくするための絞り込みです。双方の歴史に重なる部分がないとコミットの置き換え（一方の歴史のコミットをもう一方の歴史の同等のコミットで置き換え）が出来なくなるので、ここでは最近の歴史をコミット4と5だけに絞り込みます（そうすればコミット4が重なることになります）。

```
$ git log --oneline --decorate
ef989d8 (HEAD, master) fifth commit
c6e1e95 (history) fourth commit
9c68fdc third commit
945704c second commit
c1822cf first commit
```

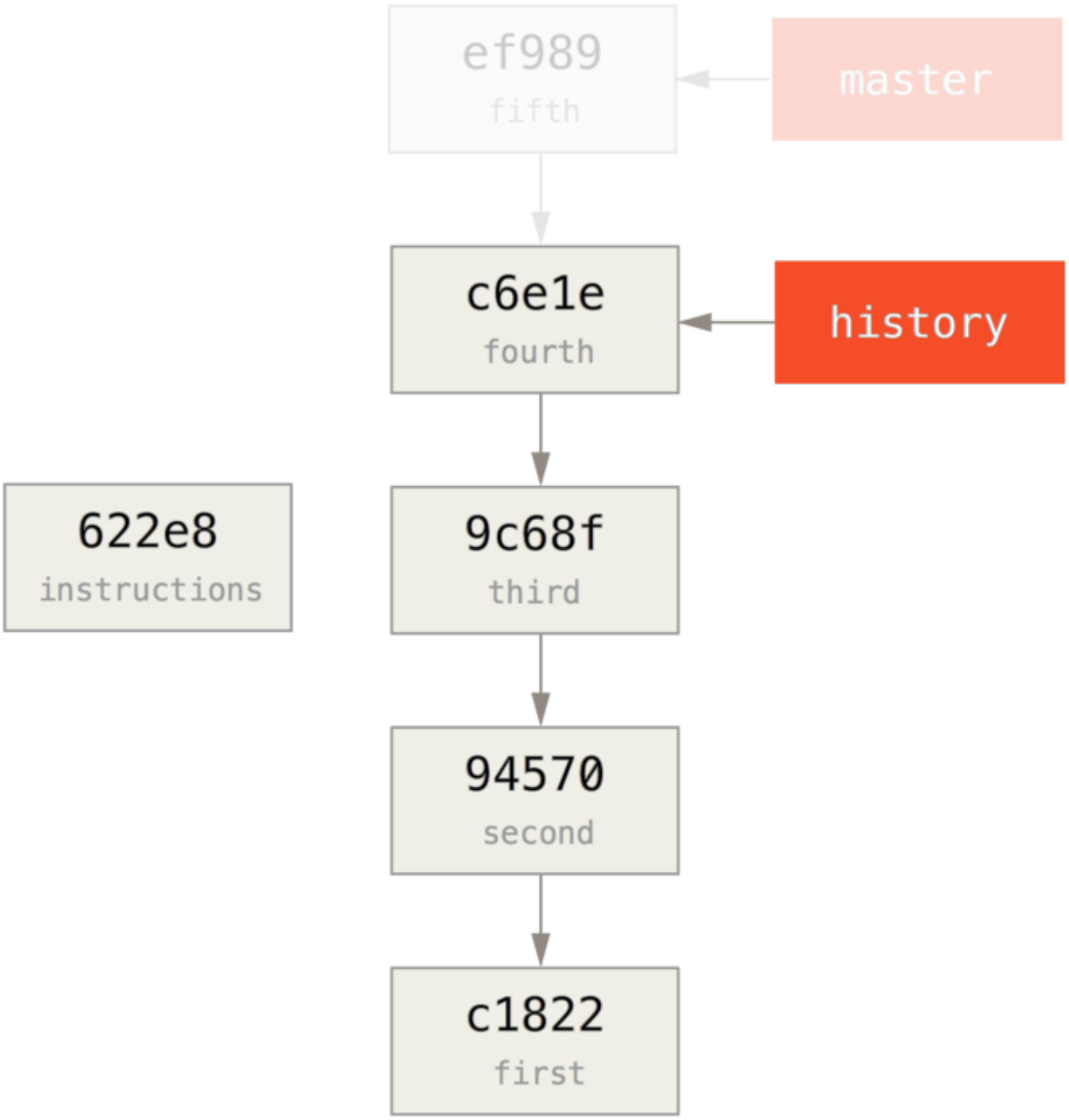
こういったケースでは、ベースとなるコミットを作って、歴史を展開するための手順を説明しておくとよいでしょう。絞りこまれた歴史のベースコミットに行き当たって「この先が知りたいのに」となった開発者達が、次に取るべき手順を把握できるからです。実際にどうするかというと、まずは上述した手順を含めたコミットオブジェクト（これが最近の歴史の方の基点となります）を作り、残りのコミット（コミット4と5）をそれにリベースします。

そのためには、どこで分割するかを決める必要があります。この例ではコミット3、SHA でいうと 9c68fdc です。そのコミットの後ろに、ベースとなるコミットを作成します。このベースコミットは commit-tree コマンドで作成できます。ツリーを指定して実行すると、親子関係のない新規のコミットオブジェクト SHA-1 が生成されます。

```
$ echo 'get history from blah blah blah' | git commit-tree 9c68fdc^{tree}
622e88e9cbfbacfb75b5279245b9fb38dfea10cf
```

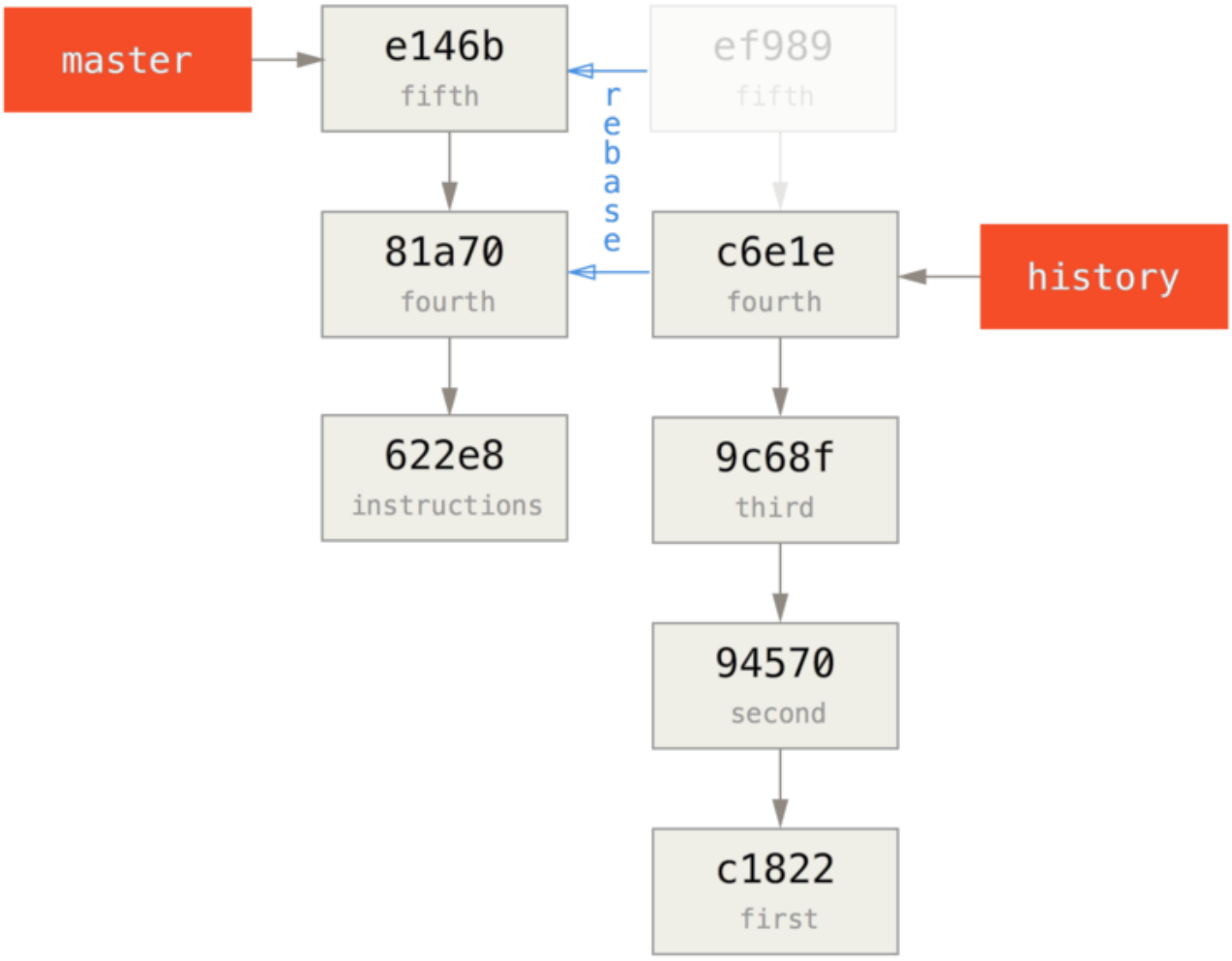
**NOTE**

`commit-tree` コマンドは、「配管」コマンドと呼ばれているコマンド群のうちの1つです。元々は直接呼び出すために作られたコマンドではなく、**他の** Git コマンドから呼び出して細かい処理をするためのものです。とはいえ、ここで説明しているような一風変わった作業をする際に使うと、低レベルの処理が出来るようになります。ただし、普段使うためのものではありません。配管コマンドの詳細は、配管（Plumbing）と磁器（Porcelain） に目を通してみてください。




これでベースとなるコミットができたので、`git rebase --onto` を使って残りの歴史をリベースしましょう。`--onto` オプションの引数は先ほど実行した `commit-tree` コマンドの返り値、リベースの始点はコミット3（保持しておきたい1つめのコミットの親にあたるコミット。`9c68fdc`）です。。

```
$ git rebase --onto 622e88 9c68fdc
First, rewinding head to replay your work on top of it...
Applying: fourth commit
Applying: fifth commit
```




以上で、仮で作ったベースコミットのうえに最近の歴史をリベースできました。ベースコミットには、必要であれば全歴史を組み直すための手順が含まれた状態です。この歴史を新しいプロジェクトとしてプッシュしておきましょう。もしそのリポジトリがクローンされると、直近のコミット2つとベースコミット（手順含む）だけが取得されます。

では次に、プロジェクトをクローンする側の動きを見ていきましょう。初回のクローンで、全歴史を必要としているとします。 絞りこまれたリポジトリをクローンした状態で全歴史を取得するには、過去の経緯を把握するためのリポジトリをリモートとして追加してフェッチします。

```
$ git clone https://github.com/schacon/project
$ cd project

$ git log --oneline master
e146b5f fifth commit
81a708d fourth commit
622e88e get history from blah blah blah

$ git remote add project-history https://github.com/schacon/project-
history
$ git fetch project-history
From https://github.com/schacon/project-history
 * [new branch]      master     -> project-history/master
```

こうすると、`master` ブランチを見れば最近のコミットがわかり、`project-history/master` ブランチを見れば過去のコミットがわかるようになります。

```
$ git log --oneline master
e146b5f fifth commit
81a708d fourth commit
622e88e get history from blah blah blah

$ git log --oneline project-history/master
c6e1e95 fourth commit
9c68fdc third commit
945704c second commit
c1822cf first commit
```

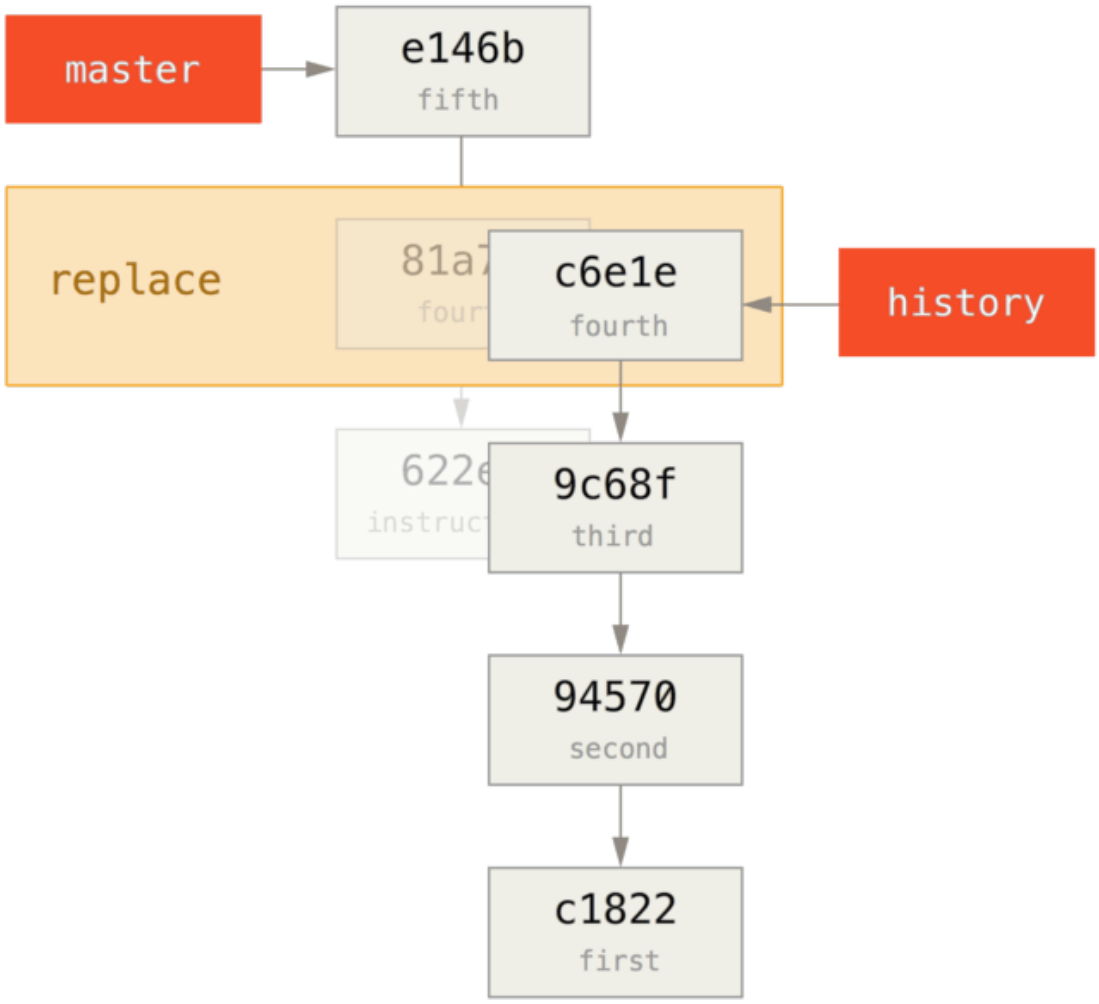
ここで `git replace` を実行すると、これら2つをつなぐことができます。置き換えられるコミット、置き換えるコミットの順に指定して実行しましょう。この例では、`master` ブランチのコミット4を、`project-history/master` ブランチのコミット4で置き換えることになります。

```
$ git replace 81a708d c6e1e95
```

では、 `master` ブランチの歴史を確認してみましょう。以下のようになっているはずです。

```
$ git log --oneline master
e146b5f fifth commit
81a708d fourth commit
9c68fdc third commit
945704c second commit
c1822cf first commit
```

ね、これいいでしょ？上流の SHA-1
をすべて書き換えることなく、歴史上のコミット1つをまったく別のコミットと置き換えることができました。他の Git ツール（`bisect` や `blame` など）も、期待通りに動作してくれます。




1つ気になるのが、表示されている SHA-1 が `81a708d` のまま、という点です。実際に使われているデータは、置き換えるのに使ったコミット `c6e1e95` のものなのですが……仮に `cat-file` のようなコマンドを実行しても、置き換え後のデータが返ってきます。

```
$ git cat-file -p 81a708d
tree 7bc544cf438903b65ca9104a1e30345eee6c083d
parent 9c68fdceee073230f19ebb8b5e7fc71b479c0252
author Scott Chacon <schacon@gmail.com> 1268712581 -0700
committer Scott Chacon <schacon@gmail.com> 1268712581 -0700

fourth commit
```

振り返ってみればわかるように、`81a708d` の本当の親は仮のコミット（`622e88e`）であって、このコマンド

出力にある `9c68fdce` ではありません。

もう1つ注目したいのが、参照のなかに保持されているデータです。

```
$ git for-each-ref
e146b5f14e79d4935160c0e83fb9ebe526b8da0d commit refs/heads/master
c6e1e95051d41771a649f3145423f8809d1a74d4 commit
refs/remotes/history/master
e146b5f14e79d4935160c0e83fb9ebe526b8da0d commit refs/remotes/origin/HEAD
e146b5f14e79d4935160c0e83fb9ebe526b8da0d commit refs/remotes/origin/master
c6e1e95051d41771a649f3145423f8809d1a74d4 commit
refs/replace/81a708dd0e167a3f691541c7a6463343bc457040
```

これはつまり、置き換えの内容を簡単に共有できるということです。サーバーにプッシュできるデータですし、ダウンロードするのも簡単です。この節で説明したように歴史を結びつける場合には、この方法は役に立ちません（というのも、全員が両方の歴史をダウンロードしてしまうからです。そうであれば、わざわざ分割する必要はないですよね）。とはいえ、これが役に立つケースもあるでしょう。

## 認証情報の保存

SSH を使ってリモートサーバーと接続しているのなら、パスフレーズなしの鍵を使えます。そうすれば、データ転送を安全に行おうとする際に、ユーザー名やパスワードを入力せずにすみます。 一方、HTTP プロトコルの場合はこうはいきません。接続のたびにユーザー名とパスワードが必要です。 さらに大変になるのが二要素認証が必要なシステムの場合です。パスワードと組み合わせて使うトークンはランダムに生成されており、unpronounceable だからです。

さいわい、Git には認証情報の仕組みがあり、上述のような大変さを軽減してくれます。 標準の仕組みで選択可能なオプションは以下のとおりです。

- デフォルトでは、なにもキャッシュされません。 接続するたび、ユーザー名とパスワードを尋ねられます。
- “cache” モードにすると、認証情報が一定の間だけメモリーに記憶されます。 パスワードはディスクには保存されません。15分経つとメモリーから除去されます。
- “store” モードにすると、認証情報がテキストファイルでディスクに保存されます。有効期限はありません。 ということは、パスワードを変更するまで、認証情報を入力しなくて済むのです。 ただし、パスワードが暗号化なしのテキストファイルでホームディレクトリに保存される、というデメリットがあります。
- Mac を使っているなら、Git の “osxkeychain” モードが使えます。これを使うと、OS のキーチェーン（システムアカウントと紐づく）に認証情報がキャッシュされます。 このモードでも認証情報がディスクに保存され、有効期限切れもありません。ただし先ほどとは違い、保存内容は暗号化（HTTPS 証明書や Safari の自動入力の暗号化と同じ仕組み）されます。
- Windows を使っているなら、“wincred” という補助ツールがあります。 “osxkeychain” と同じような仕組み（Windows Credential Store）で、重要な情報を管理します。

このオプションを設定するには、以下のように Git を設定します。

```
$ git config --global credential.helper cache
```

補助ツールには、オプションを設定できる場合があります。“store” であれば `--file <path>` という引数を指定できます。テキストファイルの保存場所を指定するために用いるオプションです（デフォルトは `~/.git-credentials`）。“cache” であれば `--timeout <seconds>` という引数を使って、補助ツールのデーモンが動作する時間を設定できます（デフォルトは “900”、15分です）。“store” 補助ツールのデフォルト設定を変更するには、以下のような設定コマンドを実行します。

```
$ git config --global credential.helper store --file ~/.my-credentials
```

また、複数のヘルパーを有効にし設定することもできます。 サーバーの認証情報が必要になると Git はこれらを順番に検索をかけていき、ヒットした時点で検索を中断します。 認証情報を保存する際は、有効なヘルパー **すべて** にユーザー名とパスワードが渡されます。それらをどう処理するかはヘルパー次第です。 以下は、複数のヘルパーを有効にする `.gitconfig` の例になります。USB メモリ上に保存されている認証情報を優先して使うけれど、もし USB メモリが使用不可の場合はパスワードを一定期間キャッシュしておく、という設定です。

```
[credential]
    helper = store --file /mnt/thumbdrive/.git-credentials
    helper = cache --timeout 30000
```

### 認証情報保存の裏側

認証情報を保存する仕組みは、いったいどのようにして動作しているのでしょうか。 認証情報ヘルパーの仕組みを操作する基本となるコマンドは `git credential` です。コマンドと標準入力経由での入力が引数になります。

例を見たほうがわかりやすいかもしれません。 仮に、認証情報ヘルパーが有効になっていて、`mygithost` というサーバーの認証情報を保存しているとします。“fill” コマンド（Git がサーバーの認証情報を探すときに呼び出されるコマンド）を使って設定をおこなうと以下のようになります。

```
$ git credential fill ①
protocol=https ②
host=mygithost
③
protocol=https ④
host=mygithost
username=bob
password=s3cre7
$ git credential fill ⑤
protocol=https
host=unknownhost

Username for 'https://unknownhost': bob
Password for 'https://bob@unknownhost':
protocol=https
host=unknownhost
username=bob
password=s3cre7
```

① このコマンドで対話モードが始まります。

② すると、標準入力からの入力を Git-credential が待機している状態になります。 ここでは、わかっている内容（プロトコルとホスト名）を入力してみます。

③ 空白行を入力すると入力が締め切られます。そうすると、認証システムに保存された内容が返ってくるはずです。

④ そうなると Git-credential の出番です。見つかった情報を標準出力に出力します。

⑤ 認証情報が見つからない場合は、ユーザーがユーザー名とパスワードを入力することになります。入力された結果は標準出力に返されます（この例では同じコンソール内で処理されています。）。

認証情報システムが呼び出しているプログラムは Git とは別のプログラムです。どのプログラムがどのように呼び出されるかは、`credential.helper` という設定によって異なっており、以下の様な値を設定できます。

| 設定値 | 挙動 |
|---|---|
| `foo` | `git-credential-foo` を実行する |
| `foo -a --opt=bcd` | `git-credential-foo -a --opt=bcd` を実行する |
| `/absolute/path/foo -xyz` | `/absolute/path/foo -xyz` を実行する |
| `!f() { echo "password=s3cre7"; }; f` | `!` 以降のコードがシェルで評価される |

これはつまり、先ほど説明した一連のヘルパーには、`git-credential-cache` や `git-credential-store` といった名前がつくということです。コマンドライン引数を受け付けるよう設定することもできます。 設定方法は “git-credential-foo [args] <action>.” になります。 なお、標準入出力のプロトコルは git-credential と同じですが、指定できるアクションが少し違ってきます。

- `get` はユーザー名/パスワードの組み合わせを要求するときに使います。
- `store` はヘルパーのメモリーに認証情報を保持するよう要求するときに使います。
- `erase` はヘルパーのメモリーから指定したプロパティの認証情報を削除するよう要求するときに使います。

`store` と `erase` のアクションの場合、レスポンスは必要ありません（Git はレスポンスを無視してしまいますし）。 ですが、`get` アクションの場合は、ヘルパーからのレスポンスは Git にとって重要な意味を持ちます。 まず、使える情報を何も保持していないときは、ヘルパーは何も出力せずに終了できます。ですが、何か情報を保持しているときは、渡された情報に対し自身が保持している情報を付加して返さなければなりません。 ヘルパーからの出力は代入文として処理されます。そしてそれを受け取った Git は、既に保持している情報を受け取った情報で置き換えます。

以下の例は先程のものと同じですが、git-credential の部分を省略して git-credential-store のみになっています。

```
$ git credential-store --file ~/git.store store ①
protocol=https
host=mygithost
username=bob
password=s3cre7
$ git credential-store --file ~/git.store get ②
protocol=https
host=mygithost

username=bob ③
password=s3cre7
```

① まずここでは、`git-credential-store` を呼び出して認証情報を保存しています。この例では、ユーザー名に “bob” 、パスワードに “s3cre7” を使って `https://mygithost` にアクセスすることになります。

② では次に、認証情報を呼び出してみます。 わかっている情報 (`https://mygithost`) を入力し、それに続いて空行も入力します。

③ すると、`git-credential-store` が先ほど保存したユーザー名とパスワード返してくれるのです。

この例での `~/git.store` は以下のようになっています。

```
https://bob:s3cre7@mygithost
```

中身は認証情報つきの URL がずらずらと続く形になっています。 なお、`osxkeychain` や `wincred` ヘルパーは情報を保存するために独自のフォーマットを使用し、`cache` ヘルパーは独自形式でメモリーに情報を保持します（他のプロセスはこの情報にアクセスできません）。

## 独自の認証情報キャッシュ

「`git-credential-store` などのプログラムは Git から独立している。」このことを理解すると、どんなプログラムであれ Git 認証情報ヘルパーとして機能できるということに気づくのもそれほど大変ではないと思います。 Git についてくるヘルパーは多くのユースケースに対応していますが、全てに対応できるわけではありません。 ここでは一例として、あなたのチームには全員が共有している認証情報があるとしましょう。デプロイ用の認証情報であればありえるケースです。 この情報は共有ディレクトリに保存されていますが、自分専用の認証情報としてコピーしておきたくはありません。頻繁に更新されるからです。 既存のヘルパーはどれもこの例には対応していません。この用途に合うヘルパーを作るには何が必要か、順を追って見ていきましょう。 まず、このプログラムには必要不可欠な機能がいくつもあります。

1. 考慮しなければならないアクションは `get` だけなので、書き込みのアクションである `store` や `erase` を受け取った場合は何もせずに終了することにします。
2. 共有されている認証情報のファイルフォーマットは `git-credential-store` のものと同様とします。
3. 同ファイルはみんなが知っているような場所に保存されていますが、もしもの場合に備えてファイルのパスを指定できるようにしておきます。

繰り返しになりますが、今回はこの拡張を Ruby で書いていきますが実際はどんな言語でも書くことができます。できあがった拡張をGit が実行さえできれば問題ありません。

```
#!/usr/bin/env ruby

require 'optparse'

path = File.expand_path '~/.git-credentials' ①
OptionParser.new do |opts|
    opts.banner = 'USAGE: git-credential-read-only [options] <action>'
    opts.on('-f', '--file PATH', 'Specify path for backing store') do
|argpath|
        path = File.expand_path argpath
    end
end.parse!

exit(0) unless ARGV[0].downcase == 'get' ②
exit(0) unless File.exists? path

known = {} ③
while line = STDIN.gets
    break if line.strip == ''
    k,v = line.strip.split '=', 2
    known[k] = v
end

File.readlines(path).each do |fileline| ④
    prot,user,pass,host =
fileline.scan(/^(.*?):\/\/(.*?):(.*?)@(.*)$/).first
    if prot == known['protocol'] and host == known['host'] then
        puts "protocol=#{prot}"
        puts "host=#{host}"
        puts "username=#{user}"
        puts "password=#{pass}"
        exit(0)
    end
end
```

① まずここでコマンドラインオプションをパースし、読み込ませるファイルをユーザーが指定できるようにしておきます。デフォルトで読み込まれるファイルは `~/.git-credentials` です。

② このプログラムが応答するのはアクションが `get` で、かつ認証情報を保持しているファイルが存在している場合に限られます。

③ このループは標準入力を読み取っていて、空行が渡されるまで続きます。 入力された内容は `known` というハッシュに保存しておき、のちのち参照することになります。

④ こちらのループではファイルの情報を検索します。 `known` ハッシュに保持されているプロトコルとハッシュに検索結果が合致した場合、検索結果が標準出力に返されます。

このヘルパーを `git-credential-read-only` としてパスの通っているところに保存したら、ファイルを実行可能にしましょう。 実際に実行したときの対話型セッションは、以下のようになります。

```
$ git credential-read-only --file=/mnt/shared/creds get
protocol=https
host=mygithost

protocol=https
host=mygithost
username=bob
password=s3cre7
```

ファイル名が “git-” で始まっているので、シンプルな書式を使って設定できます。

```
$ git config --global credential.helper read-only --file /mnt/shared/creds
```

このとおり、Git の認証情報の仕組みを拡張するのはとても単純ですし、個人やチームの悩みを解決するのに役立つはずです。

# まとめ

さまざまな高度な道具を使い、コミットやステージングエリアをより細やかに操作できる方法をまとめました。 何か問題が起こったときには、いつ誰がどのコミットでそれを仕込んだのかを容易に見つけられるようになったことでしょう。 また、プロジェクトの中で別のプロジェクトを使いたくなったときのための方法も紹介しました。 Git を使った日々のコマンドラインでの作業の大半を、自信を持ってできるようになったことでしょう。

# Git のカスタマイズ

ここまで本書では、Git の基本動作やその使用法について扱ってきました。また、Git をより簡単に効率よく使うためのさまざまなツールについても紹介しました。 本章では、重要な設定項目やフックシステムを使用して、よりカスタマイズされた方法で Git を操作する方法について扱います。 これらを利用すれば、みなさん自身やその勤務先、所属グループのニーズにあわせた方法で Git を活用できるようになるでしょう。

## Git の設定

使い始める で手短にごらんいただいたように、`git config` コマンドで Gitの設定が行えます。 最初にすることと言えば、名前とメールアドレスの設定でしょう。

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

ここでは、同じようにして設定できるより興味深い項目をいくつか身につけ、Git をカスタマイズしてみましょう。

まず、簡単におさらいしましょう。Git では、いくつかの設定ファイルを使ってデフォルト以外の挙動を定義します。 最初に Git が見るのは `/etc/gitconfig` で、ここにはシステム上の全ユーザーの全リポジトリ向けの設定値を記述します。 `git config` にオプション `--system` を指定すると、このファイルの読み書きを行います。

次に Git が見るのは `~/.gitconfig` （または `~/.config/git/config`）で、これは各ユーザー専用のファイルです。 Git でこのファイルの読み書きをするには、`--global` オプションを指定します。

最後に Git が設定値を探すのは、現在使用中のリポジトリの設定ファイル (`.git/config`) です。 この値は、そのリポジトリだけで有効なものです。

これらの “レベル” （システム、グローバル、ローカル）の間では、いずれも後から読んだ値がその前の値を上書きします。したがって、たとえば `.git/config` に書いた値は `/etc/gitconfig` での設定よりも優先されます。

> **NOTE** Git の設定ファイルはプレーンテキストなので、これらのファイルを手動で編集し、正しい構文で内容を追加することで、上記のような設定を行うことも可能ですが、通常は `git config` コマンドを使ったほうが簡単です。

### 基本的なクライアントのオプション

Git の設定オプションは、おおきく二種類に分類できます。クライアント側のオプションとサーバー側のオプションです。 大半のオプションは、クライアント側のもの、つまり個人的な作業環境を設定するためのものとなります。 大量の、*本当に大量の* オプションが使用できますが、ここでは、もっとも一般的で、もっともよく使われているものだけを取り上げます。 その他のオプションの多くは特定の場合にのみ有用なものなの

で、ここでは扱いません。 Git
で使えるすべてのオプションを知りたい場合は、次のコマンドを実行しましょう。

```
$ man git-config
```

このコマンドは、利用できるすべてのオプションを、簡単な説明とともに一覧表示します。 この内容は、http://git-scm.com/docs/git-config.html にあるリファレンスでも見ることができます。

### core.editor

デフォルトでは、コミットやタグのメッセージを編集するときには、ユーザーがデフォルトエディタとして設定したエディタ（$VISUAL または $EDITOR）が使われます。デフォルトエディタが設定されていない場合は vi エディタが使われます。 このデフォルト設定を別のものに変更するには core.editor を設定します。

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

これで、シェルのデフォルトエディタに関係なく、Git でメッセージを編集する際には Emacs が起動されるようになりました。

### commit.template

システム上のファイルへのパスをここに設定すると、Git はそのファイルをコミット時のデフォルトメッセージとして使います。 たとえば、次のようなテンプレートファイルを作って ~/.gitmessage.txt においたとしましょう。

```
subject line

what happened

[ticket: X]
```

git commit のときにエディタに表示されるデフォルトメッセージをこれにするには、commit.template の設定を変更します。

```
$ git config --global commit.template ~/.gitmessage.txt
$ git commit
```

すると、コミットメッセージの雛形としてこのような内容がエディタに表示されます。

```
subject line

what happened

[ticket: X]
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
# modified:   lib/test.rb
#
~
~
".git/COMMIT_EDITMSG" 14L, 297C
```

コミットメッセージについてチーム内に所定の決まりがあるのなら、その決まりに従ったテンプレートをシステム上に作って Git にそれを使わせるようにするとよいでしょう。そうすれば、その決まりに従ってもらいやすくなります。

`core.pager`

core.pager は、Git が `log` や `diff` などを出力するときに使うページャを設定します。 `more` などのお好みのページャを設定したり (デフォルトは `less` です)、空文字列を設定してページャを使わないようにしたりできます。

```
$ git config --global core.pager ''
```

これを実行すると、すべてのコマンドの出力を、どんなに長くなったとしても全部 Git が出力するようになります。

`user.signingkey`

署名入りの注釈付きタグ (作業内容への署名 で取り上げました) を作る場合は、GPG 署名用の鍵を登録しておくと便利です。 鍵の ID を設定するには、このようにします。

```
$ git config --global user.signingkey <gpg-key-id>
```

これで、`git tag` コマンドでいちいち鍵を指定しなくてもタグに署名できるようになりました。

```
$ git tag -s <tag-name>
```

`core.excludesfile`

プロジェクトごとの `.gitignore` ファイルでパターンを指定すると、`git add` したときに Git がそのファイルを無視してステージしないようになります。これについては ファイルの無視 で説明しました。

ですが、作業中のすべてのリポジトリで、ある特定のファイルを無視したい場合もあります。 Mac OS X を使っているのなら、 `.DS_Store` というファイルに見おぼえがあるでしょう。 使っているエディタが Emacs か Vim なら、 `~` で終わるファイルのことを知っていることと思います。

このような設定を行うには、グローバルな `.gitignore` のようなファイルが必要です。
`~/.gitignore_global` ファイルへ次の内容を書き込んで、

```
*~
.DS_Store
```

その上で `git config --global core.excludesfile ~/.gitignore_global` を実行すれば、これらのファイルで手を煩わすことは二度となくなります。

`help.autocorrect`

Git でコマンドを打ち間違えると、こんなふうに表示されます。

```
$ git chekcout master
git: 'chekcout' is not a git command. See 'git --help'.

Did you mean this?
    checkout
```

Git は気を利かせて、何をしたかったのか推測はしてくれますが、実行まではしません。
`help.autocorrect` を 1 にしておくと、 Git は実際にそのコマンドを実行しようとします。

```
$ git chekcout master
WARNING: You called a Git command named 'chekcout', which does not exist.
Continuing under the assumption that you meant 'checkout'
in 0.1 seconds automatically...
```

“0.1 seconds” という箇所に注目してください。 `help.autocorrect` は整数値で、0.1秒単位での時間を表しています。 そのため、仮に 50 を設定したなら、自動修正したコマンドが実行される前に 5 秒の猶予が与えら

れます。

## Git における色

Git では、ターミナルへの出力に色をつけることができます。ぱっと見て、すばやくお手軽に出力内容を把握できるようになるでしょう。 さまざまなオプションで、お好みに合わせて色を設定しましょう。

`color.ui`

Git は自動的に大半の出力に色づけをします。ですが、この挙動が気に入らないなら、そのためのマスタースイッチがあります。 ターミナルへの出力への色付けをすべてオフにするなら、以下のようにします。

```
$ git config --global color.ui false
```

デフォルトの設定は `auto` で、直接ターミナルへ出力する場合には色付けを行いますが、パイプやファイルへリダイレクトした場合にはカラーコントロールコードを出力しません。

また `always` を指定すると、ターミナルであってもパイプであっても色をつけます。 `always` を使うことは、まずないでしょう。たいていの場合は、カラーコードを含む結果をリダイレクトしたければ、 Git コマンドに `--color` フラグを渡せばカラーコードの使用を強制できます。 ふだんはデフォルトの設定で要望を満たせるでしょう。

`color.*`

どのコマンドをどのように色づけするかをより細やかに指定したい場合、コマンド単位の色づけ設定を使用します。 これらの項目には `true` 、 `false` あるいは `always` が指定できます。

```
color.branch
color.diff
color.interactive
color.status
```

さらに、これらの項目ではサブ設定が使え、出力の一部について特定の色を使うように指定することもできます。 たとえば、diff の出力で、メタ情報を黒地に青の太字で出力させたい場合は次のようにします。

```
$ git config --global color.diff.meta "blue black bold"
```

色として指定できる値は `normal`、 `black`、 `red`、 `green`、 `yellow`、 `blue`、 `magenta`、 `cyan`、 `white` のいずれかです。先ほどの例の bold のように属性も指定できます。 `bold`、 `dim`、 `ul` （下線つき）、 `blink`、 `reverse` （文字と背景の色を逆にする）のいずれかを指定できます。

## 外部のマージツールおよび diff ツール

Git には、内部的な diff の実装が組み込まれています。本書でこれまで見てきた内容は、それを使用しています。ですが、外部のツールを使うよう設定することもできます。 また、コンフリクトを手動で解決するのではなくグラフィカルなコンフリクト解消ツールを使うよう設定することもできます。 ここでは Perforce Visual Merge Tool (P4Merge) を使って diff の表示とマージの処理を行えるようにする例を示します。これはすばらしいグラフィカルツールで、しかも無料で使えるからです。

P4Merge はすべての主要プラットフォーム上で動作するので、実際に試してみたい人は試してみるとよいでしょう。 この例では、Mac や Linux 形式のパス名を例に使います。Windows の場合は、`/usr/local/bin` のところを環境に合わせたパスに置き換えてください。

まず、P4Merge を からダウンロードします。 次に、コマンドを実行するための外部ラッパースクリプトを用意します。 この例では、Mac 用の実行パスを使います。他のシステムで使う場合は、`p4merge` のバイナリがインストールされた場所に置き換えてください。 次のような内容のマージ用ラッパースクリプト `extMerge` を用意してください。これは、 `p4merge` にすべての引数を渡して呼び出します。

```
$ cat /usr/local/bin/extMerge
#!/bin/sh
/Applications/p4merge.app/Contents/MacOS/p4merge $*
```

diff のラッパーは、7 つの引数が渡されていることを確認したうえでそのうちのふたつをマージスクリプトに渡します。 デフォルトでは、Git は次のような引数を diff プログラムに渡します。

```
path old-file old-hex old-mode new-file new-hex new-mode
```

ここで必要な引数は `old-file` と `new-file` だけなので、ラッパースクリプトではこれらを渡すようにします。

```
$ cat /usr/local/bin/extDiff
#!/bin/sh
[ $# -eq 7 ] && /usr/local/bin/extMerge "$2" "$5"
```

また、これらのツールは実行可能にしておかなければなりません。

```
$ sudo chmod +x /usr/local/bin/extMerge
$ sudo chmod +x /usr/local/bin/extDiff
```

これで、自前のマージツールや diff ツールを使えるように設定する準備が整いました。 設定項目はひとつだけではありません。まず `merge.tool` でどんなツールを使うのかを Git に伝え、 `mergetool.<tool>.cmd` でそのコマンドを実行する方法を指定し、`mergetool.<tool>.trustExitCode` では「そのコマンドの終

了コードでマージが成功したかどうかを判断できるのか」を指定し、`diff.external` では diff の際に実行するコマンドを指定します。つまり、このような 4 つのコマンドを実行することになります。

```
$ git config --global merge.tool extMerge
$ git config --global mergetool.extMerge.cmd \
  'extMerge \"$BASE\" \"$LOCAL\" \"$REMOTE\" \"$MERGED\"'
$ git config --global mergetool.extMerge.trustExitCode false
$ git config --global diff.external extDiff
```

あるいは、`~/.gitconfig` ファイルを編集してこのような行を追加します。

```
[merge]
  tool = extMerge
[mergetool "extMerge"]
  cmd = extMerge "$BASE" "$LOCAL" "$REMOTE" "$MERGED"
  trustExitCode = false
[diff]
  external = extDiff
```

すべて設定し終えたら、このような diff コマンドを実行すると、

```
$ git diff 32d1776b1^ 32d1776b1
```

結果をコマンドラインに出力するかわりに、Git から P4Merge が呼び出され、次のようになります。

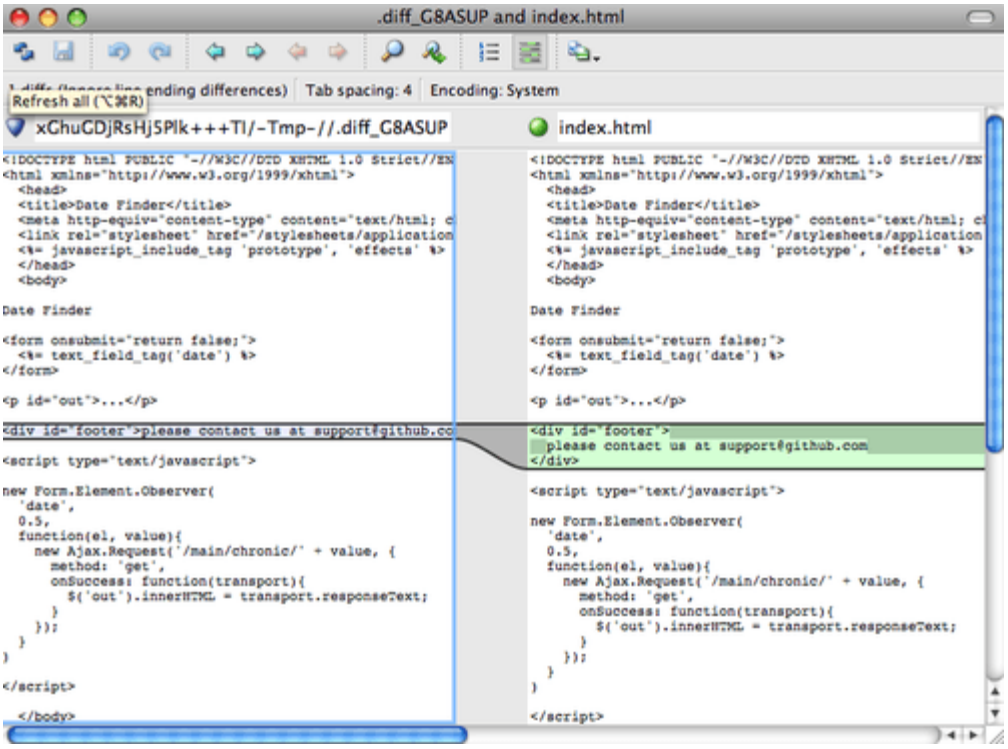



*Figure 142. P4Merge.*

ふたつのブランチをマージしてコンフリクトが発生した場合は `git mergetool` を実行します。するとP4Merge が立ち上がり、コンフリクトの解決を GUI ツールで行えるようになります。

このようなラッパーを設定しておくと、あとで diff ツールやマージツールを簡単に変更できます。 たとえば `extDiff` や `extMerge` で KDiff3 を実行させるように変更するには `extMerge` ファイルをこのように変更するだけでよいのです。

```
$ cat /usr/local/bin/extMerge
#!/bin/sh
/Applications/kdiff3.app/Contents/MacOS/kdiff3 $*
```

これで、Git での diff の閲覧やコンフリクトの解決の際に KDiff3 が立ち上がるようになりました。

Git にはさまざまなマージツール用の設定が事前に準備されており、特に設定しなくても利用できます。 サポートされているツールを確認するには、次のコマンドを実行します。

```
$ git mergetool --tool-help
'git mergetool --tool=<tool>' may be set to one of the following:
        emerge
        gvimdiff
        gvimdiff2
        opendiff
        p4merge
        vimdiff
        vimdiff2

The following tools are valid, but not currently available:
        araxis
        bc3
        codecompare
        deltawalker
        diffmerge
        diffuse
        ecmerge
        kdiff3
        meld
        tkdiff
        tortoisemerge
        xxdiff

Some of the tools listed above only work in a windowed
environment. If run in a terminal-only session, they will fail.
```

KDiff3 を diff ツールとしてではなくマージのときにだけ使いたい場合は、kdiff3 コマンドにパスが通っている状態で次のコマンドを実行します。

```
$ git config --global merge.tool kdiff3
```

`extMerge` や `extDiff` を準備せずにこのコマンドを実行すると、マージの解決の際には KDiff3 を立ち上げて diff の際には通常の Git の diff ツールを使うようになります。

## 書式設定と空白文字

書式設定や空白文字の問題は微妙にうっとうしいもので、とくにさまざまなプラットフォームで開発している人たちと共同作業をするときに問題になりがちです。 使っているエディタが知らぬ間に空白文字を埋め込んでしまっていたり Windows で開発している人が行末にキャリッジリターンを付け加えてしまったりなどしてパッチが面倒な状態になってしまうことも多々あります。 Git では、こういった問題に対処するための設定項目も用意しています。

### core.autocrlf

自分が Windows で開発している一方、チームの中に Windows 以外の環境で開発している人がいる場合（逆も同様）には、改行コードの問題に巻き込まれることがありがちです。 Windows ではキャリッジリターンとラインフィードでファイルの改行を表すのですが、Mac や Linux ではラインフィードだけで改行を表すという違いが原因です。 これはささいな違いではありますが、さまざまなプラットフォームにまたがる作業では非常に面倒なものです。Windows のエディタには、LFだけの改行をだまってCRLFに置き換えたり、ユーザがEnterキーを押下した際にCRとLFの両方を挿入したりするものが数多くあります。

Git はこの問題に対処するために、コミットする際には行末の CRLF を LF に自動変換し、ファイルシステム上にチェックアウトするときには逆の変換を行うようにできます。 この機能を使うには core.autocrlf を設定します。

Windows で作業をするときにこれを true に設定すると、コードをチェックアウトするときに行末の LF を CRLF に自動変換してくれます。

```
$ git config --global core.autocrlf true
```

Linux や Mac などの行末に LF を使うシステムで作業をしている場合は、Git にチェックアウト時の自動変換をされてしまうと困ります。しかし、行末が CRLF なファイルが紛れ込んでしまった場合には Git に自動修正してもらいたいものです。 コミット時の CRLF から LF への変換はさせたいけれどもそれ以外の自動変換が不要な場合は、core.autocrlf を input に設定します。

```
$ git config --global core.autocrlf input
```

この設定は、Windows にチェックアウトしたときの CRLF への変換は行いますが、Mac や Linux へのチェックアウト時は LF のままにします。

Windows のみのプロジェクトで作業をしているのなら、この機能を無効にしてキャリッジリターンをそのままリポジトリに記録してもよいでしょう。その場合は、値 false を設定します。

```
$ git config --global core.autocrlf false
```

### core.whitespace

Git には、空白文字に関する問題を見つけて修正するための設定もあります。 空白文字に関する主要な六つの問題に対応するもので、そのうち三つはデフォルトで有効になっています。残りの三つはデフォルトでは有効になっていませんが、有効化することもできます。

デフォルトで有効になっている設定は、行末の空白文字を見つける blank-at-eol 、ファイル末尾の空白文字を見つける blank-at-eof 、行頭のタブ文字より前にある空白文字を見つける space-before-tab です。

デフォルトでは無効だけれども有効にすることもできる三つの設定は、行頭がタブ文字でなく空白文字になっている行を見つける indent-with-non-tab （空白文字の数は tabwidth オプションで制御可能）、行内のインデント部分にあるタブ文字を見つける tab-in-indent 、行末のキャリッジリターンを許容する cr-at-eol です。

これらのオン・オフを切り替えるには、core.whitespace にカンマ区切りで項目を指定します。 無効にしたい場合は、設定文字列でその項目を省略するか、あるいは項目名の前に - をつけます。 たとえば cr-at-eol 以外のすべてを設定したい場合は、このようにします。

```
$ git config --global core.whitespace \
    trailing-space,space-before-tab,indent-with-non-tab
```

git diff コマンドを実行したときに Git がこれらの問題を検出すると、その部分を色付けして表示します。 修正してからコミットするようにしましょう。 この設定は、git apply でパッチを適用する際にも助けとなります。 空白に関する問題を含むパッチを適用するときに警告を発してほしい場合には、次のようにします。

```
$ git apply --whitespace=warn <patch>
```

あるいは、問題を自動的に修正してからパッチを適用したい場合は、次のようにします。

```
$ git apply --whitespace=fix <patch>
```

これらの設定は、git rebase コマンドにも適用されます。 空白に関する問題を含むコミットをしたけれどまだそれを公開リポジトリにプッシュしていない場合は、 git rebase --whitespace=fix を実行すれば、パッチを書き換えて空白問題を自動修正してくれます。

## サーバーの設定

Git のサーバー側の設定オプションはそれほど多くありませんが、いくつか興味深いものがあるので紹介します。

### receive.fsckObjects

デフォルトでは、Git はプッシュで受け取ったオブジェクトの SHA-1 チェックサムが一致していて有効なオブジェクトを指しているということをチェックさせることができます。 ですが、デフォルトではこのチェックは行わないようになっています。このチェックは比較的重たい処理であり、リポジトリのサイズが大きかったりプッシュする量が多かったりすると、毎回チェックさせるのには時間がかかるでしょう。 毎回のプッシュの際に Git にオブジェクトの一貫性をチェックさせたい場合は、receive.fsckObjects を true にして強制的にチェックさせるようにします。

```
$ git config --system receive.fsckObjects true
```

これで、Git
がリポジトリの整合性を確認してからでないとプッシュが認められないようになります。壊れたデータをまちがって受け入れてしまうことがなくなりました。

`receive.denyNonFastForwards`

すでにプッシュしたコミットをリベースしてもう一度プッシュした場合、あるいはリモートブランチが現在指しているコミットを含まないコミットをプッシュしようとした場合は、プッシュが拒否されます。 これは悪くない方針でしょう。しかしリベースの場合は、自分が何をしているのかをきちんと把握していれば、プッシュの際に `-f` フラグを指定して強制的にリモートブランチを更新することもできます。

このような強制更新機能を無効にするには、`receive.denyNonFastForwards` を設定します。

```
$ git config --system receive.denyNonFastForwards true
```

もうひとつの方法として、サーバー側の receive フックを使うこともできます。こちらの方法については後ほど簡単に説明します。 receive フックを使えば、特定のユーザーだけ強制更新を無効にするなどより細やかな制御ができるようになります。

`receive.denyDeletes`

`denyNonFastForwards` の制限を回避する方法として、いったんブランチを削除してから新しいコミットを参照するブランチをプッシュしなおすことができます。 これを無効にするには、 `receive.denyDeletes` を true に設定します。

```
$ git config --system receive.denyDeletes true
```

これは、プッシュによるブランチやタグの削除を一切拒否し、誰も削除できないようにします。 リモートブランチを削除するには、サーバー上の ref ファイルを手で削除しなければなりません。ACL を使って、ユーザー単位でこれを制限することもできますが、その方法は Git ポリシーの実施例 で扱います。

# Git の属性

設定項目の中には、パスに対して指定できるものもあります。Git はこれらの設定を、指定したパスのサブディレクトリやファイルにのみ適用します。 これらパス固有の設定は、 Git の属性と呼ばれ、あるディレクトリ（通常はプロジェクトのルートディレクトリ）の直下の `.gitattributes` か、あるいはそのファイルをプロジェクトとともにコミットしたくない場合は `.git/info/attributes` に設定します。

属性を使うと、ファイルやディレクトリ単位で個別のマージ戦略を指定したり、テキストファイル以外の diff を取る方法を指示したり、あるいはチェックインやチェックアウトの前にその内容を Git にフィルタリングさせたりできます。 このセクションでは、Git プロジェクトでパスに対して設定できる属性のいくつかについて学び、実際にその機能を使う例を見ていきます。

## バイナリファイル

Git の属性を使ってできるちょっとした技として、どのファイルがバイナリファイルなのかを (その他の方法で判別できない場合のために) 指定した上で、 Git に対してバイナリファイルの扱い方を指示するというものがあります。 たとえば、機械で生成したテキストファイルの中には diff が取得できないものがありますし、バイナリファイルであっても diff が取得できるものもあります。 それを Git に指示する方法を紹介します。

### バイナリファイルの特定

テキストファイルのように見えるファイルであっても、何らかの目的のために意図的にバイナリデータとして扱いたいことがあります。 たとえば、Mac の Xcode プロジェクトの中には `.pbxproj` で終わる名前のファイルがあります。これは JSON (プレーンテキスト形式の JavaScript のデータフォーマット) のデータセットで、IDE がビルドの設定などをディスクに書き出したものです。 このファイルの内容はすべて UTF-8 の文字なので、理論上はテキストファイルであると言えます。しかし、このファイルをテキストファイルとして扱いたくはありません。実際のところ、このファイルは軽量なデータベースとして使われているからです。他の人が変更した内容はマージできませんし、diff をとってもあまり意味がありません。 このファイルは、基本的に機械が処理するものなのです。 要するに、バイナリファイルと同じように扱いたいということです。

すべての `pbxproj` ファイルをバイナリデータとして扱うよう Git に指定するには、次の行を `.gitattributes` ファイルに追加します。

```
*.pbxproj binary
```

これで、Git が CRLF 問題の対応をすることもなくなりますし、`git show` や `git diff` を実行したときにもこのファイルの diff を調べることはなくなります。

### バイナリファイルの差分

バイナリファイルに対して意味のある差分を取る際にも、Git の属性を使うことができます。 普通の diff でも比較できるよう、バイナリデータをテキストデータに変換する方法をGitに教えればいいのです。

このテクニックを使ってまず解決したいことといえば、人類にとって最も厄介な問題のひとつ、Wordで作成した文書のバージョン管理ではないでしょうか。 奇妙なことに、Wordは最悪のエディタだと全ての人が知っているにも係わらず、皆がWordを使っています。 Word文書をバージョン管理したいと思ったなら、Gitのリポジトリにそれらを追加して、まとめてコミットすればいいのです。しかし、それでいいのでしょうか？ あなたが `git diff` をいつも通りに実行すると、次のように表示されるだけです。

```
$ git diff
diff --git a/chapter1.docx b/chapter1.docx
index 88839c4..4afcb7c 100644
Binary files a/chapter1.docx and b/chapter1.docx differ
```

これでは、2つのバージョンをチェックアウトして、目視で見比べなくては、比較はできませんよね？ Gitの属性を使えば、これをうまく解決できます。 `.gitattributes`に次の行を追加して下さい。

```
*.docx diff=word
```

これは、指定したパターン (`.docx`) にマッチした全てのファイルに対して、差分を表示する時には “word” というフィルタを使うよう Git に指示しているのです。 では、 “word” フィルタとは何でしょうか？ これは自分で用意しなければなりません。 ここでは、 `docx2txt` を使ってWord文書をテキストファイルに変換した上で、正しく diff が取れるように設定してみましょう。

まず、 `docx2txt` をインストールする必要があります。 http://docx2txt.sourceforge.net からダウンロードしたら、 `INSTALL` ファイルの指示に従って、シェルから見える場所にファイルを置いてください。 次に、出力を Git に合わせて変換するラッパースクリプトを作成します。 パスの通った場所に、 `docx2txt` という名前のファイルを次の内容で作成してください。

```
#!/bin/bash
docx2txt.pl $1 -
```

作ったファイルに `chmod a+x` するのを忘れないでください。 最後に、Git がこのファイルを使うように設定します。

```
$ git config diff.word.textconv docx2txt
```

これで、二つのスナップショットの diff を取る際に、ファイル名の末尾が `.docx` だったら、 “word” フィルタを通す（この “word” フィルタは `docx2txt` というプログラムとして定義されている）ということが Git に伝わりました。 こうすることで、Wordファイルの差分を取る際に、より効果的なテキストベースでの差分を取ることができるようになります。

例を示しましょう。この本の第1章をWord形式に変換し、Gitリポジトリに登録しました。 さらに、新しい段落を追加しました。 `git diff` の出力は次のようになります。

```
$ git diff
diff --git a/chapter1.docx b/chapter1.docx
index 0b013ca..ba25db5 100644
--- a/chapter1.docx
+++ b/chapter1.docx
@@ -2,6 +2,7 @@
 This chapter will be about getting started with Git. We will begin at the
beginning by explaining some background on version control tools, then
move on to how to get Git running on your system and finally how to get it
setup to start working with. At the end of this chapter you should
understand why Git is around, why you should use it and you should be all
setup to do so.
 1.1. About Version Control
 What is "version control", and why should you care? Version control is a
system that records changes to a file or set of files over time so that
you can recall specific versions later. For the examples in this book you
will use software source code as the files being version controlled,
though in reality you can do this with nearly any type of file on a
computer.
+Testing: 1, 2, 3.
 If you are a graphic or web designer and want to keep every version of an
image or layout (which you would most certainly want to), a Version
Control System (VCS) is a very wise thing to use. It allows you to revert
files back to a previous state, revert the entire project back to a
previous state, compare changes over time, see who last modified something
that might be causing a problem, who introduced an issue and when, and
more. Using a VCS also generally means that if you screw things up or lose
files, you can easily recover. In addition, you get all this for very
little overhead.
 1.1.1. Local Version Control Systems
 Many people's version-control method of choice is to copy files into
another directory (perhaps a time-stamped directory, if they're clever).
This approach is very common because it is so simple, but it is also
incredibly error prone. It is easy to forget which directory you're in and
accidentally write to the wrong file or copy over files you don't mean to.
```

Gitは、追加した “Testing: 1, 2, 3.” という正しい文字列を首尾よく、かつ、簡潔に知らせてくれました。 これだけでは完璧ではありません（書式の変更はここでは表示されていません）が、確実に動作しています。

その他の興味深い問題としては、画像ファイルの差分があります。 ひとつの方法として、EXIF情報（多くのファイル形式で使用されているメタデータ）を抽出するフィルタを使う方法があります。 exiftool`をダウンロードしてインストールすれば、画像データを、メタデータを表すテキストデータへ変換できます。これによって、 diff では少なくとも、変更内容をテキスト形式で表示できるようになります。 ではここで、以下の行を.gitattributes`に追加してみましょう。

```
*.png diff=exif
```

続いて、さきほどインストールしたツールを使うようGitの設定を変更します。

```
$ git config diff.exif.textconv exiftool
```

プロジェクト中の画像データを置き換えて `git diff` を実行すると、次のように表示されるでしょう。

```
diff --git a/image.png b/image.png
index 88839c4..4afcb7c 100644
--- a/image.png
+++ b/image.png
@@ -1,12 +1,12 @@
 ExifTool Version Number         : 7.74
-File Size                       : 70 kB
-File Modification Date/Time     : 2009:04:21 07:02:45-07:00
+File Size                       : 94 kB
+File Modification Date/Time     : 2009:04:21 07:02:43-07:00
 File Type                       : PNG
 MIME Type                       : image/png
-Image Width                     : 1058
-Image Height                    : 889
+Image Width                     : 1056
+Image Height                    : 827
 Bit Depth                       : 8
 Color Type                      : RGB with Alpha
```

ファイルのサイズと画像のサイズが変更されたことが簡単に見て取れます。

## キーワード展開

SubversionやCVSを使っていた開発者から、キーワード展開機能をリクエストされることがよくあります。ここでの主な問題は、Git では、コミットの後に、コミットに関する情報を使ってファイルを変更することはできないということです。これは、Git がコミットの最初にファイルのチェックサムを生成するためです。 しかし、ファイルをチェックアウトする際にテキストを挿入し、コミットへ追加する際にそれを削除することは可能です。 Gitの属性はこれを行うための方法を2つ提供します。

ひとつめの方法として、ファイルの `$Id$` フィールドへ、 blob の SHA-1 チェックサムを自動的に挿入できます。 あるファイル、もしくはいくつかのファイルに対してこの属性を設定すれば、次にそのブランチをチェックアウトする時、Gitはこの置き換えを行うようになります。 ただし、挿入されるチェックサムはコミットに対するものではなく、対象となるblobのものであるという点に注意して下さい。 ではここで、以下の行を`.gitattributes`に追加してみましょう。

```
*.txt ident
```

続いて、`$Id$`への参照をテスト用ファイルに追加します。

```
$ echo '$Id$' > test.txt
```

そうすると、次にこのファイルをチェックアウトする時、GitはblobのSHA-1チェックサムを挿入します。

```
$ rm test.txt
$ git checkout -- test.txt
$ cat test.txt
$Id: 42812b7653c7b88933f8a9d6cad0ca16714b9bb3 $
```

しかし、この結果はあまり役に立ちません。 CVSやSubversionのキーワード展開ではタイムスタンプを含めることができます。対して、SHA-1チェックサムは完全にランダムな値ですから、2つの値の新旧を知るための助けにはなりません。

これには、コミットおよびチェックアウトの時にキーワード展開を行うフィルタを書いてやれば対応できます。 このフィルタは “clean” および “smudge” フィルタと呼ばれます。 `.gitattributes` ファイルで、特定のパスにフィルタを設定し、チェックアウトの直前（ “smudge” 、 チェックアウトする時に “smudge” フィルタを実行する を参照）およびステージングの直前（ “clean” 、 ステージングする時に “clean” フィルタを実行する を参照）に処理を行うスクリプトを設定できます。 これらのフィルタは、色々と面白いことに使えます。

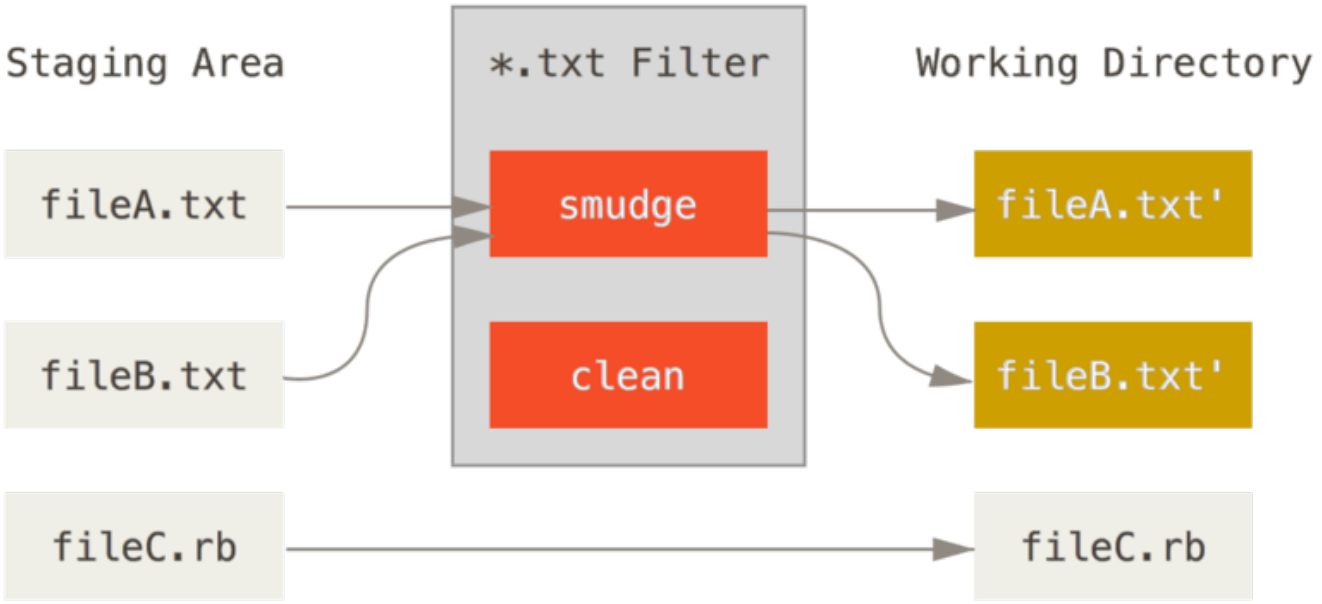



*Figure 143. チェックアウトする時に “smudge” フィルタを実行する*

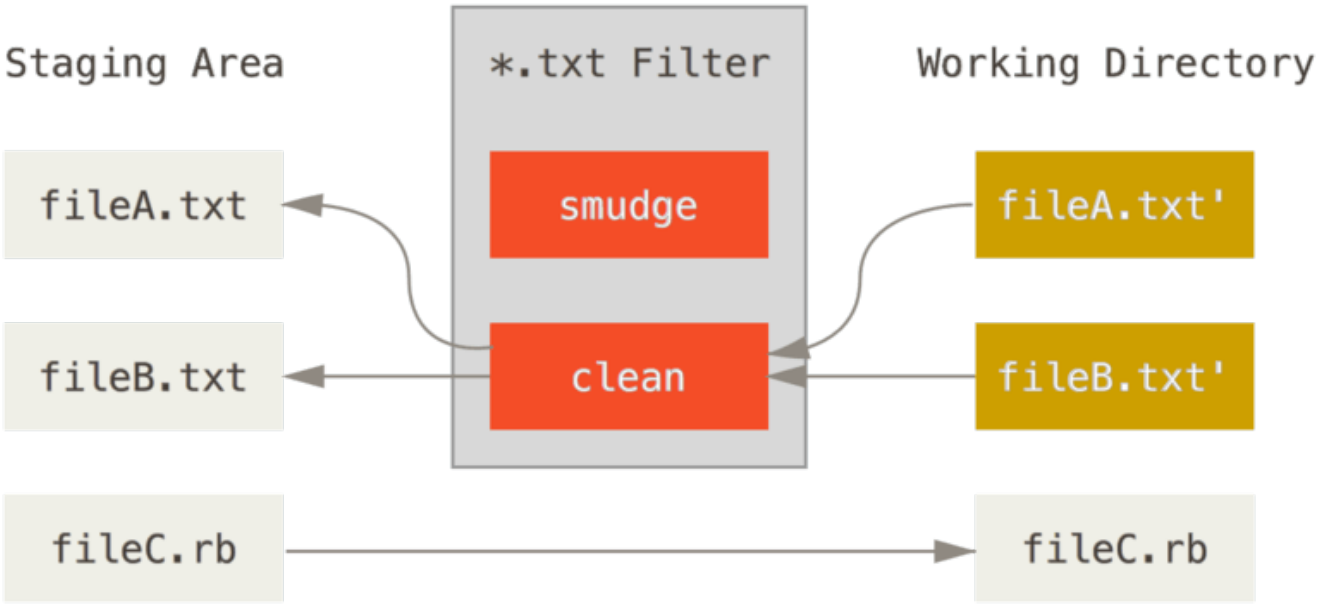



*Figure 144. ステージングする時に “clean” フィルタを実行する*

この機能に対してオリジナルのコミットメッセージは簡単な例を与えてくれています。それはコミット前にCのソースコードを `indent` プログラムに通すというものです。 `*.c` ファイルに対してこのフィルタを実行するように、`.gitattributes`ファイルにfilter属性を設定できます。

```
*.c filter=indent
```

それから、smudgeとcleanで “indent” フィルタが何を行えばいいのかをGitに教えます。

```
$ git config --global filter.indent.clean indent
$ git config --global filter.indent.smudge cat
```

このケースでは、 `*.c` にマッチするファイルをコミットした時、Gitはステージング前にindentプログラムにファイルを通し、チェックアウトする前には `cat` を通すようにします。 `cat`は基本的に何もしません。入力されたデータと同じデータを吐き出すだけです。 この組み合わせを使えば、Cのソースコードのコミット前に、効果的に `indent` を通せます。

もうひとつの興味深い例として、RCSスタイルの `$Date$` キーワード展開があります。 これを正しく行うには、ファイル名を受け取り、プロジェクトの最新のコミットの日付を見て、その日付をファイルに挿入するちょっとしたスクリプトが必要になります。 これを行うRubyスクリプトを以下に示します。

```
#! /usr/bin/env ruby
data = STDIN.read
last_date = `git log --pretty=format:"%ad" -1`
puts data.gsub('$Date$', '$Date: ' + last_date.to_s + '$')
```

このスクリプトは、`git log` コマンドの出力から最新のコミットの日付を取得し、標準入力中のすべての `$Date$` 文字列にその日付を追加し、結果を出力します。お気に入りのどんな言語で書くにしても、簡単なスクリプトになるでしょう。 このスクリプトファイルに`expand_date`と名前をつけ、実行パスのどこかに置きます。 次に、Git にフィルタ（ここでは `dater`とします）を設定し、チェックアウト時に `smudge` で `expand_date` フィルタを使うように指定します。 コミット時に日付を削除するのには、 Perl の正規表現が使えます。

```
$ git config filter.dater.smudge expand_date
$ git config filter.dater.clean 'perl -pe
"s/\\\$Date[^\\\$]*\\\$/\\\$Date\\\$/"'
```

このPerlのスニペットは、 `$Date$` 文字列の内側にある内容を削除し、日付を挿入する前の状態に戻します。さて、フィルタの準備ができました。このファイルが新しいフィルタに引っかかるように Git の属性を設定し、ファイルに `$Date$` キーワードを追加した上で、テストしてみましょう。

```
date*.txt filter=dater
```

```
$ echo '# $Date$' > date_test.txt
```

これらの変更をコミットして、再度ファイルをチェックアウトすれば、キーワードが正しく置き換えられているのがわかります。

```
$ git add date_test.txt .gitattributes
$ git commit -m "Testing date expansion in Git"
$ rm date_test.txt
$ git checkout date_test.txt
$ cat date_test.txt
# $Date: Tue Apr 21 07:26:52 2009 -0700$
```

アプリケーションのカスタマイズにあたり、このテクニックがどれほど強力か、おわかりいただけたと思います。 しかし、注意してほしいのですが、 `.gitattributes` ファイルはコミットされてプロジェクト内で共有されますが、ドライバ（このケースで言えば、`dater`）そうはそうはいきません。そのため、この機能はどこででも働くわけではありません。 フィルタを設計する時には、たとえフィルタが正常に動作しなかったとしても、プロジェクトは適切に動き続けられるようにすべきです。

## リポジトリをエクスポートする

あなたのプロジェクトのアーカイブをエクスポートする時には、Gitの属性データを使って興味深いことができます。

### export-ignore

アーカイブを生成するとき、特定のファイルやディレクトリをエクスポートしないように設定できます。 プロジェクトにはチェックインしたいが、アーカイブファイルには含めたくないディレクトリやファイルがあるなら、それらに export-ignore 属性を設定することで、分別が行えます。

例えば、プロジェクトをエクスポートする際に tarball に含めたくないテストファイルが、 `test/` ディレクトリ以下に入っているとしましょう。 その場合、次の1行をGitの属性ファイルに追加します。

```
test/ export-ignore
```

これで、プロジェクトのtarballを作成するために git archive を実行した時、アーカイブには test/ ディレクトリが含まれないようになります。

### export-subst

デプロイ用にファイルをエクスポートする際に、export-subst 属性のついたファイルを指定して git log のログ書式指定機能とキーワード展開機能で生成した内容をファイルに付与できます。 例えば、LAST_COMMIT`という名前のファイルをプロジェクトに追加し、`git archive`を実行した時にそのファイルのメタデータを最新コミットと同じ内容に変換したい場合、.gitattributes`ファイルと`LAST_COMMIT`ファイルを 次のように設定します。

```
LAST_COMMIT export-subst
```

```
$ echo 'Last commit date: $Format:%cd by %aN$' > LAST_COMMIT
$ git add LAST_COMMIT .gitattributes
$ git commit -am 'adding LAST_COMMIT file for archives'
```

git archive を実行すると、 LAST_COMMIT は以下のような内容になっているはずです。

```
$ git archive HEAD | tar xCf ../deployment-testing -
$ cat ../deployment-testing/LAST_COMMIT
Last commit date: Tue Apr 21 08:38:48 2009 -0700 by Scott Chacon
```

このような置換に、コミットメッセージや git note を用いることもできます。その際、git log コマンドのワードラップ処理が適用されます。

```
$ echo '$Format:Last commit: %h by %aN at %cd%n%+w(76,6,9)%B$' >
LAST_COMMIT
$ git commit -am 'export-subst uses git log's custom formatter

git archive uses git log's `pretty=format:` processor
directly, and strips the surrounding `$Format:` and `$`
markup from the output.
'
$ git archive @ | tar xfO - LAST_COMMIT
Last commit: 312ccc8 by Jim Hill at Fri May 8 09:14:04 2015 -0700
       export-subst uses git log's custom formatter

         git archive uses git log's `pretty=format:` processor directly,
and
         strips the surrounding `$Format:` and `$` markup from the output.
```

この結果作成されたアーカイブはデプロイするのにぴったりです。一方、いったんエクスポートされてしまったアーカイブで開発を続けるのはおすすめできません。

## マージの戦略

Gitの属性を使えば、プロジェクト中の特定のファイルに対して、異なるマージ戦略を使うこともできます。非常に有用なオプションのひとつに、指定したファイルで競合が発生した場合に、マージを行わずに、あなたの変更内容で他の誰かの変更を上書きするように設定するというものがあります。

これはプロジェクトにおいて、分岐したブランチや、特別版のブランチで作業をしている時、そのブランチでの変更をマージさせたいが、特定のファイルの変更はなかったことにしたいというような時に助けになります。 例えば、 `database.xml` というデータベースの設定ファイルがあり、ふたつのブランチでその内容が異なっているとしましょう。そして、そのデータベースファイルを台無しにすることなしに、一方のブランチへとマージしたいとします。 これは、次のように属性を設定すれば実現できます。

```
database.xml merge=ours
```

その上で、ダミーのマージ戦略 `ours` を次のように定義します。

```
$ git config --global merge.ours.driver true
```

もう一方のブランチでマージを実行すると、 `database.xml` に関する競合は発生せず、次のような結果になります。

```
$ git merge topic
Auto-merging database.xml
Merge made by recursive.
```

この場合、 `database.xml` は元々のバージョンのまま、書き変わりません。

# Git フック

他のバージョンコントロールシステムと同じように、Gitにも特定のアクションが発生した時にカスタムスクリプトを叩く方法があります。 このようなフックは、クライアントサイドとサーバーサイドの二つのグループに分けられます。 クライアントサイドフックはコミットやマージといったクライアントでの操作の際に、サーバーサイドフックはプッシュされたコミットの受け取りといったネットワーク操作の際に、それぞれ実行されます。 これらのフックは、さまざまなな目的に用いることができます。

## フックをインストールする

フックは、Gitディレクトリの `hooks` サブディレクトリ（一般的なプロジェクトでは、 `.git/hooks` ）に格納されています。 `git init` で新しいリポジトリを初期化する時には、Gitに同梱されているスクリプトのサンプルがこの hooks ディレクトリに格納されます。サンプルの多くはそのままでも十分有用ですし、また、各スクリプトの入力値に関するドキュメントもついています。 サンプルは全てシェルスクリプトで書かれており、その中の一部では Perl も使われています。ですが、どんなスクリプトでも、実行可能かつ適切に命名されてさえいれば、問題なく動きます。Ruby や Python などで書くこともできます。 これら同梱のフックスクリプトを使用する場合は、ファイル名の末尾が `.sample` となっていますので適宜リネームしてください。

フックスクリプトを有効にするには、Gitディレクトリの `hooks` サブディレクトリに、実行可能なファイルを適切な名前（拡張子は使えません）で配置すれば、以降そのファイルが呼び出されます。 ここでは重要なフックファイル名をいくつか取り上げます。

## クライアントサイドフック

クライアントサイドフックにはたくさんの種類があります。 ここではコミットワークフローフック、Eメールワークフロースクリプト、その他クライアントサイドフックに分類します。

| | |
|---|---|
| **NOTE** | 特筆すべき点として、クライアントサイドフックはリポジトリをクローンする際には **コピーされません** 。 スクリプトを使って何らかのポリシーを強制したいのなら、サーバサイドで行う方がよいでしょう。サンプルが Git ポリシーの実施例 にあります。 |

### コミットワークフローフック

最初の4つのフックはコミットプロセスに関するものです。

`pre-commit` フックは、コミットメッセージが入力される前に実行されます。 これは、いまからコミットされるスナップショットを検査したり、何かし忘れた事がないか確認したり、テストが実行できるか確認したり、何かしらコードを検査する目的で使用されます。 このフックがゼロでない値を返すと、コミットが中断

されます。また、この検査は `git commit --no-verify` で飛ばすこともできます。
ここではコーディングスタイルの検査（lintを実行するなど）や、行末の空白文字の検査（デフォルトのフックがまさにそうです）、新しく追加されたメソッドのドキュメントが正しいかどうかの検査といったことが可能です。

`prepare-commit-msg`フックは、コミットメッセージエディターが起動する直前、デフォルトメッセージが生成された直後に実行されます。 このフックでは、デフォルトメッセージを、コミットの作者の目に触れる前に編集できます。 このフックにはパラメータがあり、その時点でのコミットメッセージを保存したファイルへのパス、コミットのタイプ、さらにamendされたコミットの場合はコミットの SHA-1 をパラメータとして取ります。 このフックは普段のコミットにおいてはあまり有用ではありませんが、テンプレートが用意されているコミットメッセージ・mergeコミット・squashコミット・amendコミットのような、デフォルトメッセージが自動生成されるコミットにおいて効果を発揮します。 コミットメッセージのテンプレートと組み合わせれば、プログラムで情報を動的に挿入できます。

`commit-msg` フックは、開発者の書いたコミットメッセージを保存した一時ファイルへのパスをパラメータに取ります。 このスクリプトがゼロ以外の値を返した場合、Git はコミットプロセスを中断します。これを使えば、コミットを許可して処理を進める前に、プロジェクトの状態やコミットメッセージを検査できます。
この章の最後のセクションでは、このフックを使用してコミットメッセージが要求された様式に沿っているか検査するデモンストレーションを行います。

コミットプロセスが全て完了した後には、`post-commit`フックが実行されます。 このフックはパラメータを取りませんが、 `git log -1 HEAD` を実行することで直前のコミットを簡単に取り出すことができます。 一般的にこのスクリプトは何かしらの通知といった目的に使用されます。

#### Eメールワークフローフック

Eメールを使ったワークフロー用として、三種類のクライアントサイドフックを設定できます。 これらはすべて `git am` コマンドに対して起動されるものなので、ふだんのワークフローでこのコマンドを使っていない場合は次のセクションまで読み飛ばしてもかまいません。 `git format-patch` で作ったパッチを受け取ることがあるなら、ここで説明する内容の中に有用なものがあるかもしれません。

最初に実行されるフックは `applypatch-msg` です。 これは引数をひとつ（コミットメッセージを含む一時ファイル名）だけ受け取ります。 このスクリプトがゼロ以外の戻り値で終了した場合、Git はパッチの処理を強制終了させます。 このフックを使うと、コミットメッセージの書式が正しいかどうかを確認したり、スクリプトで正しい書式に手直ししたりできます。

`git am` でパッチを適用するときに二番目に実行されるフックは `pre-applypatch` です。 少々ややこしいのですが、このフックはパッチが *適用された後* 、コミットが作成される前に実行されます。そのため、このフックでは、スナップショットの内容を、コミットする前に調べることができます。 このスクリプトを使えば、テストを実行したり、ワーキングツリーの調査をしたりといったことが行えます。 なにか抜けがあったりテストが失敗したりした場合はスクリプトをゼロ以外の戻り値で終了させます。そうすれば、`git am` はパッチをコミットせずに強制終了します。

`git am` において最後に実行されるフックは `post-applypatch` です。このフックは、コミットが作成された後に実行されます。 これを使うと、特定のグループのメンバーや、プルしたパッチの作者に対して、処理の完了を伝えることができます。 このスクリプトでは、パッチの適用を中断させることはできません。

### その他のクライアントフック

pre-rebase フックは何かをリベースする前に実行され、ゼロ以外を返せばその処理を中断できます。 このフックを使うと、既にプッシュ済みのコミットのリベースを却下できます。 Git に同梱されているサンプルの pre-rebase フックがこの処理を行いますが、このフックの前提となっている条件のなかには読者のワークフローに合わないものもあるでしょう。

post-rewrite フックは、既存のコミットを書き換えるコマンド、例えば git commit --amend や git rebase を実行した際に実行されます（ただし git filter-branch では実行されません）。 引数はひとつで、コミットの書き換えを行ったコマンドを引数に取ります。また、書き換えを行ったファイルのリストを stdin から受け取ります。 このフックは post-checkout や post-merge といったフックと同じ用途に使えます。

git checkout が正常に終了すると、post-checkout フックが実行されます。これを使うと、作業ディレクトリを自分のプロジェクトの環境にあわせて設定できます。 たとえば、バージョン管理対象外の巨大なバイナリファイルを作業ディレクトリに取り込んだり、ドキュメントを自動生成したりといった処理が行えます。

post-merge フックは、merge コマンドが正常に終了したときに実行されます。 これを使うと、Git では追跡できないパーミッション情報などを作業ツリーに復元できます。 作業ツリーに変更が加わったときに取り込みたい Git の管理対象外のファイルの存在確認などにも使えます。

pre-push フックは、 git push を実行した際、リモート参照が更新された後、オブジェクトの転送が始まる前に実行されます。 このフックはリモートの名前と場所を引数に取ります。また、これから更新する参照のリストを stdin から受け取ります。 このフックは、プッシュを行う前に、更新される参照を検査するのに使用できます（ゼロ以外の値を返すとプッシュが中断されます）。

Git は通常の操作の一環として、時折 git gc --auto を実行してガベージコレクションを行います。 pre-auto-gc フックは、ガベージコレクションが実行される直前に呼び出されます。このフックは、ガベージコレクションが実行されることを通知したり、タイミングが悪い場合にガベージコレクションを中断したりするのに使用できます。

## サーバーサイドフック

システム管理者としてプロジェクトのポリシーを強制させる際には、クライアントサイドフックに加え、いくつかのサーバーサイドフックを使うこともできます。 これらのスクリプトは、サーバへのプッシュの前後に実行されます。 pre フックをゼロ以外の値で終了させると、プッシュを却下してエラーメッセージをクライアントに返すことができます。つまり、プッシュに関して、好きなだけ複雑なポリシーを設定できるということです。

### pre-receive

クライアントからのプッシュを処理するときに最初に実行されるスクリプトが pre-receive です。 このスクリプトは、プッシュされた参照のリストを標準入力から受け取ります。ゼロ以外の値で終了させると、これらはすべて却下されます。 このフックを使うと、更新内容がすべてfast-forwardであることをチェックしたり、プッシュによって変更されるファイルや参照に対するアクセス制御を行ったりできます。

`update`

`update` スクリプトは `pre-receive` スクリプトと似ていますが、プッシュしてきた人が更新しようとしているブランチごとに実行されるという点が異なります。 複数のブランチへのプッシュがあったときに `pre-receive` が実行されるのは一度だけですが、update はブランチ単位でそれぞれ一度ずつ実行されます。 このスクリプトは、標準入力を読み込むのではなく三つの引数を受け取ります。参照 (ブランチ) の名前、プッシュ前を指す参照の SHA-1、そしてプッシュしようとしている参照の SHA-1 です。 update スクリプトをゼロ以外で終了させると、その参照のみが却下されます。それ以外の参照はそのまま更新を続行します。

`post-receive`

`post-receive` フックは処理が終了した後で実行されるもので、他のサービスの更新やユーザーへの通知などに使えます。 このフックは、 `pre-receive` フックと同じデータを標準入力から受け取ります。 サンプルのスクリプトには、リストをメールしたり、継続的インテグレーションサーバーへ通知したり、チケット追跡システムを更新したりといった処理が含まれています。コミットメッセージを解析して、チケットのオープン・修正・クローズなどの必要性を調べることもできます。 このスクリプトではプッシュの処理を中断させることはできませんが、クライアント側ではこのスクリプトが終了するまで接続を切断できません。このスクリプトで時間のかかる処理をさせるときには十分注意しましょう。

# Git ポリシーの実施例

このセクションでは、これまでに学んだ内容を使って実際に Git のワークフローを確立してみます。 コミットメッセージの書式をチェックし、またプロジェクト内の特定のサブディレクトリを特定のユーザーだけが変更できるようにします。 以降では、開発者に対して「なぜプッシュが却下されたのか」を伝えるためのクライアントスクリプトと、ポリシーを強制するためのサーバースクリプトを作成していきます。

以降で示すスクリプトは Ruby で書かれています。理由としては、我々の知的習慣によるところもありますが、Ruby は（たとえ書けないとしても）読むのが簡単というのも理由のひとつです。 しかし、それ以外の言語であってもきちんと動作します。Git に同梱されているサンプルスクリプトはすべて Perl あるいは Bash で書かれています。サンプルスクリプトを見れば、それらの言語による大量のフックの例を見ることができます。

## サーバーサイドフック

サーバーサイドで行う処理は、すべて `hooks` ディレクトリの `update` ファイルにまとめます。 `update` ファイルはプッシュされるブランチごとに実行され、次の3つの引数を取ります。

- プッシュされる参照の名前
- 操作前のブランチのリビジョン
- プッシュされる新しいリビジョン

また、SSH 経由でのプッシュの場合は、プッシュしたユーザーを知ることもできます。 全員に共通のユーザー（“git” など）を使って公開鍵認証をしている場合は、公開鍵の情報に基づいて実際のユーザーを判断して環境変数を設定するというラッパーが必要です。 ここでは、接続しているユーザー名が環境変数 `$USER` に格納されているものとします。 `update` スクリプトは、まず必要な情報を取得するところから始まります。

```
#!/usr/bin/env ruby

$refname = ARGV[0]
$oldrev  = ARGV[1]
$newrev  = ARGV[2]
$user    = ENV['USER']

puts "Enforcing Policies..."
puts "(#{$refname}) (#{$oldrev[0,6]}) (#{$newrev[0,6]})"
```

そう、グローバル変数を使ってますね。 が、責めないでください – 実例を示すには、こっちの方が簡単なんです。

**特定のコミットメッセージ書式の強制**

まずは、コミットメッセージを特定の書式に従わせることに挑戦してみましょう。 ここでは、コミットメッセージには必ず “ref: 1234” 形式の文字列を含むこと、というルールにします。個々のコミットをチケットシステムの作業項目とリンクさせたいという意図です。 やらなければならないことは、プッシュされてきた各コミットのコミットメッセージに上記の文字列があるか調べ、なければゼロ以外の値を返して終了し、プッシュを却下することです。

プッシュされたすべてのコミットの SHA-1 値を取得するには、`$newrev` と `$oldrev` の内容を `git rev-list` という Git の配管（plumbing）コマンドに渡します。 これは基本的には `git log` コマンドのようなものですが、デフォルトでは SHA-1 値だけを表示してそれ以外の情報は出力しません。 ふたつのコミットの間のすべてのコミットの SHA-1 を得るには、次のようなコマンドを実行します。

```
$ git rev-list 538c33..d14fc7
d14fc7c847ab946ec39590d87783c69b031bdfb7
9f585da4401b0a3999e84113824d15245c13f0be
234071a1be950e2a8d078e6141f5cd20c1e61ad3
dfa04c9ef3d5197182f13fb5b9b1fb7717d2222a
17716ec0f1ff5c77eff40b7fe912f9f6cfd0e475
```

この出力を受け取って、ループさせて各コミットの SHA-1 を取得し、個々のメッセージを取り出せば、正規表現でそのメッセージを調べることができます。

さて、これらのコミットからコミットメッセージを取り出す方法を見つけなければなりません。 生のコミットデータを取得するには、別の配管コマンド `git cat-file` を使います。 配管コマンドについては Gitの内側 で詳しく説明しますが、とりあえずはこのコマンドがどんな結果を返すのだけを示します。

```
$ git cat-file commit ca82a6
tree cfda3bf379e4f8dba8717dee55aab78aef7f4daf
parent 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
author Scott Chacon <schacon@gmail.com> 1205815931 -0700
committer Scott Chacon <schacon@gmail.com> 1240030591 -0700

changed the version number
```

SHA-1 値がわかっているときにコミットからコミットメッセージを得るシンプルな方法は、空行を探してそれ以降をすべて取得するというものです。 これには、Unix システムの `sed` コマンドが使えます。

```
$ git cat-file commit ca82a6 | sed '1,/^$/d'
changed the version number
```

プッシュしようとしているコミットから、この呪文を使ってコミットメッセージを取得し、もし条件にマッチしないものがあれば終了させればよいのです。 スクリプトを抜けてプッシュを却下するには、ゼロ以外の値を返して終了します。 以上を踏まえると、このメソッドは次のようになります。

```
$regex = /\[ref: (\d+)\]/

# enforced custom commit message format
def check_message_format
  missed_revs = `git rev-list #{$oldrev}..#{$newrev}`.split("\n")
  missed_revs.each do |rev|
    message = `git cat-file commit #{rev} | sed '1,/^$/d'`
    if !$regex.match(message)
      puts "[POLICY] Your message is not formatted correctly"
      exit 1
    end
  end
end
check_message_format
```

これを `update` スクリプトに追加すると、ルールを守らないコミットメッセージが含まれるコミットのプッシュを却下するようになります。

#### ユーザーベースのアクセス制御

アクセス制御リスト (ACL) を使って、ユーザーごとにプロジェクトのどの部分に対して変更をプッシュできるのかを指定できる仕組みを追加したいとしましょう。 全体にアクセスできるユーザーもいれば、特定のサブディレクトリやファイルにしか変更をプッシュできないユーザーもいる、といった具合です。 これを行うには、ルールを書いたファイル `acl` をサーバー上のベア Git リポジトリに置きます。 `update` フックにこのファイルを読ませ、プッシュされてきたコミットにどのようなファイルが含まれているのかを調べ、そしてプッシュしたユーザーにそのファイルを変更する権限があるのか判断します。

まずは ACL を作るところから始めましょう。 ここでは、CVS の ACL と似た書式を使います。これは各項目を一行で表し、最初のフィールドは `avail` あるいは `unavail`、そして次の行がそのルールを適用するユーザーの一覧（カンマ区切り）、そして最後のフィールドがそのルールを適用するパス（ブランクは全体へのアクセスを意味します）です。フィールドの区切りには、パイプ文字 (`|`) を使います。

ここでは、全体にアクセスできる管理者、 `doc` ディレクトリにアクセスできるドキュメント担当者、そして `lib` と `tests` ディレクトリだけにアクセスできる開発者を設定します。ACL ファイルは次のようになります。

```
avail|nickh,pjhyett,defunkt,tpw
avail|usinclair,cdickens,ebronte|doc
avail|schacon|lib
avail|schacon|tests
```

まずはこのデータを読み込んで、スクリプト内で使えるデータ構造にしてみましょう。 例をシンプルにするために、ここでは `avail` ディレクティブだけを使います。 次のメソッドは連想配列を返すものです。配列のキーはユーザー名、キーに対応する値はそのユーザーが書き込み権限を持つパスの配列になります。

```
def get_acl_access_data(acl_file)
  # read in ACL data
  acl_file = File.read(acl_file).split("\n").reject { |line| line == '' }
  access = {}
  acl_file.each do |line|
    avail, users, path = line.split('|')
    next unless avail == 'avail'
    users.split(',').each do |user|
      access[user] ||= []
      access[user] << path
    end
  end
  access
end
```

先ほどの ACL ファイルをこの `get_acl_access_data` メソッドに渡すと、このようなデータ構造を返します。

```
{"defunkt"=>[nil],
 "tpw"=>[nil],
 "nickh"=>[nil],
 "pjhyett"=>[nil],
 "schacon"=>["lib", "tests"],
 "cdickens"=>["doc"],
 "usinclair"=>["doc"],
 "ebronte"=>["doc"]}
```

これで権限がわかったので、あとはプッシュされた各コミットがどのパスを変更しようとしているのかを調べれば、そのユーザーがプッシュできるのか判断できます。

あるコミットでどのファイルが変更されるのかを知るのはとても簡単で、`git log` コマンドに `--name-only` オプションを指定するだけです（Git の基本 で簡単に説明しました）。

```
$ git log -1 --name-only --pretty=format:'' 9f585d

README
lib/test.rb
```

`get_acl_access_data` メソッドが返す ACL のデータとこのファイルリストを付き合わせれば、そのユーザーにコミットをプッシュする権限があるかどうかを判断できます。

```
# only allows certain users to modify certain subdirectories in a project
def check_directory_perms
  access = get_acl_access_data('acl')

  # see if anyone is trying to push something they can't
  new_commits = `git rev-list #{$oldrev}..#{$newrev}`.split("\n")
  new_commits.each do |rev|
    files_modified = `git log -1 --name-only --pretty=format:''
#{rev}`.split("\n")
    files_modified.each do |path|
      next if path.size == 0
      has_file_access = false
      access[$user].each do |access_path|
        if !access_path  # user has access to everything
           || (path.start_with? access_path) # access to this path
          has_file_access = true
        end
      end
      if !has_file_access
        puts "[POLICY] You do not have access to push to #{path}"
        exit 1
      end
    end
  end
end

check_directory_perms
```

最初に `git rev-list` でサーバへプッシュされるコミットの一覧を取得します。 次に、それぞれのコミットでどのファイルが変更されるのかを調べ、プッシュしてきたユーザーにそのファイルを変更する権限があるか確かめています。

これで、まずい形式のコミットメッセージや、指定されたパス以外のファイルの変更を含むコミットはプッシュできなくなりました。

#### テストを実施する

これまでのコードを書き込んだファイルに対して `chmod u+x .git/hooks/update` を実行します。その上で、メッセージが規定に沿っていないコミットをプッシュしてみましょう。すると、こんなメッセージが表示されるでしょう。

```
$ git push -f origin master
Counting objects: 5, done.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 323 bytes, done.
Total 3 (delta 1), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
Enforcing Policies...
(refs/heads/master) (8338c5) (c5b616)
[POLICY] Your message is not formatted correctly
error: hooks/update exited with error code 1
error: hook declined to update refs/heads/master
To git@gitserver:project.git
 ! [remote rejected] master -> master (hook declined)
error: failed to push some refs to 'git@gitserver:project.git'
```

この中には、興味深い点がいくつかあります。 まず、フックの実行が始まったときの次の表示に注目しましょう。

```
Enforcing Policies...
(refs/heads/master) (fb8c72) (c56860)
```

これは、スクリプトの先頭で標準出力に表示した内容でした。 ここで重要なのは「スクリプトから `stdout` に送った内容は、すべてクライアントにも送られる」ということです。

次に注目するのは、エラーメッセージです。

```
[POLICY] Your message is not formatted correctly
error: hooks/update exited with error code 1
error: hook declined to update refs/heads/master
```

最初の行はスクリプトから出力したもので、その他の 2 行は Git が出力したものです。この 2 行では、スクリプトがゼロ以外の値で終了したためにプッシュが却下されたということを説明しています。 最後に、次の部分に注目します。

```
To git@gitserver:project.git
 ! [remote rejected] master -> master (hook declined)
error: failed to push some refs to 'git@gitserver:project.git'
```

フックで却下したすべての参照について、remote rejected メッセージが表示されます。これを見れば、フック内での処理のせいで却下されたのだということがわかります。

また、変更権限のないファイルを変更してそれを含むコミットをプッシュしようとしたときも、同様にエラーが表示されます。 たとえば、ドキュメント担当者が `lib` ディレクトリ内の何かを変更しようとした場合のメ

ッセージは次のようになります。

```
[POLICY] You do not have access to push to lib/test.rb
```

以降、この update スクリプトが動いてさえいれば、指定したパターンを含まないコミットメッセージがリポジトリに登録されることは二度とありません。また、ユーザーに変なところをさわられる心配もなくなります。

## クライアントサイドフック

この方式の弱点は、プッシュが却下されたときにユーザーが泣き寝入りせざるを得なくなるということです。手間暇かけて仕上げた作業が最後の最後で却下されるというのは、非常にストレスがたまるし不可解です。さらに、プッシュするためには歴史を修正しなければならないのですが、気弱な人にとってそれはかなりつらいことです。

このジレンマに対する答えとして、サーバーが却下するであろう作業をするときに、それをユーザーに伝えるためのクライアントサイドフックを用意します。 そうすれば、何か問題があるときに、それをコミットする前に知ることができるので、取り返しのつかなくなる前に問題を修正できます。 なおプロジェクトをクローンしてもフックはコピーされないので、別の何らかの方法で各ユーザーにスクリプトを配布した上で、各ユーザーにそれを .git/hooks にコピーさせ、実行可能にさせる必要があります。 フックスクリプト自体をプロジェクトに含めたり別のプロジェクトにしたりすることはできますが、各自の環境でそれをフックとして自動的に設定することはできません。

はじめに、コミットを書き込む直前にコミットメッセージをチェックしなければなりません。コミットメッセージの書式に問題があったがために、変更がサーバーに却下されるということがないように、コミットメッセージの書式を調べるのです。 これを行うには commit-msg フックを使います。 最初の引数で渡されたファイルからコミットメッセージを読み込んでパターンと比較し、もしマッチしなければ Git の処理を中断させます。

```
#!/usr/bin/env ruby
message_file = ARGV[0]
message = File.read(message_file)

$regex = /\[ref: (\d+)\]/

if !$regex.match(message)
  puts "[POLICY] Your message is not formatted correctly"
  exit 1
end
```

このスクリプトを適切な場所 (.git/hooks/commit-msg) に置いて実行可能にしておくと、不適切なメッセージを書いてコミットしようとしたときに次のような結果となります。

```
$ git commit -am 'test'
[POLICY] Your message is not formatted correctly
```

このとき、実際にはコミットされません。 もしメッセージが適切な書式になっていれば、Git はコミットを許可します。

```
$ git commit -am 'test [ref: 132]'
[master e05c914] test [ref: 132]
 1 file changed, 1 insertions(+), 0 deletions(-)
```

次に、ACL で決められた範囲以外のファイルを変更していないことを確認しましょう。 先ほど使った ACL ファイルのコピーがプロジェクトの `.git` ディレクトリにあれば、次のような `pre-commit` スクリプトでチェックできます。

```
#!/usr/bin/env ruby

$user    = ENV['USER']

# [ insert acl_access_data method from above ]

# only allows certain users to modify certain subdirectories in a project
def check_directory_perms
  access = get_acl_access_data('.git/acl')

  files_modified = `git diff-index --cached --name-only HEAD`.split("\n")
  files_modified.each do |path|
    next if path.size == 0
    has_file_access = false
    access[$user].each do |access_path|
    if !access_path || (path.index(access_path) == 0)
      has_file_access = true
    end
    if !has_file_access
      puts "[POLICY] You do not have access to push to #{path}"
      exit 1
    end
  end
end

check_directory_perms
```

大まかにはサーバーサイドのスクリプトと同じですが、重要な違いがふたつあります。 まず、ACL ファイルの場所が違います。このスクリプトは作業ディレクトリから実行するものであり、`.git` ディレクトリから実行するものではないからです。 ACL ファイルの場所を、先ほどの

```
access = get_acl_access_data('acl')
```

から次のように変更しなければなりません。

```
access = get_acl_access_data('.git/acl')
```

もうひとつの違いは、変更されたファイルの一覧を取得する方法です。 サーバーサイドのメソッドではコミットログを調べていました。しかしこの時点ではまだコミットが記録されていないので、ファイルの一覧はステージング・エリアから取得しなければなりません。 つまり、先ほどの

```
files_modified = `git log -1 --name-only --pretty=format:'' #{ref}`
```

は次のようになります。

```
files_modified = `git diff-index --cached --name-only HEAD`
```

しかし、違うのはこの二点だけで、それ以外はまったく同じように動作します。 ただしこのスクリプトは、ローカルで実行しているユーザーと、リモートマシンにプッシュするときのユーザーが同じであることを前提にしています。 もし異なる場合は、変数 `$user` を手動で設定しなければなりません。

最後に残ったのは fast-forward でないプッシュを止めることです。 fast-forward でない参照を取得するには、すでにプッシュした過去のコミットにリベースするか、別のローカルブランチにリモートブランチと同じところまでプッシュしなければなりません。

サーバーサイドではすでに `receive.denyDeletes` と `receive.denyNonFastForwards` でこのポリシーを強制しているでしょうから、あり得るのは、すでにプッシュ済みのコミットをリベースしようとするときくらいです。

それをチェックする pre-rebase スクリプトの例を示します。 これは書き換えようとしているコミットの一覧を取得し、それがリモート参照の中に存在するかどうかを調べます。 リモート参照から到達可能なコミットがひとつでもあれば、リベースを中断します。

```
#!/usr/bin/env ruby

base_branch = ARGV[0]
if ARGV[1]
  topic_branch = ARGV[1]
else
  topic_branch = "HEAD"
end

target_shas = `git rev-list #{base_branch}..#{topic_branch}`.split("\n")
remote_refs = `git branch -r`.split("\n").map { |r| r.strip }

target_shas.each do |sha|
  remote_refs.each do |remote_ref|
    shas_pushed = `git rev-list ^#{sha}^@ refs/remotes/#{remote_ref}`
    if shas_pushed.split("\n").include?(sha)
      puts "[POLICY] Commit #{sha} has already been pushed to
#{remote_ref}"
      exit 1
    end
  end
end
```

このスクリプトでは、リビジョンの選択 ではカバーしていない構文を使っています。 既にプッシュ済みのコミットの一覧を得るために、次のコマンドを実行します。

```
`git rev-list ^#{sha}^@ refs/remotes/#{remote_ref}`
```

SHA^@ 構文は、そのコミットのすべての親を解決します。 リモートの最後のコミットから到達可能で、これからプッシュしようとする SHA-1 の親のいずれかからもアクセスできないコミット（これによって fast-forward であることが分かります）を探します。

この方式の弱点は非常に時間がかかることで、多くの場合このチェックは不要です。-f つきで強制的にプッシュしようとしない限り、サーバーが警告を出してプッシュできないからです。 しかし練習用の課題としてはおもしろいもので、あとでリベースを取り消してやりなおすはめになることを理屈上は防げるようになります。

# まとめ

Git クライアントとサーバーをカスタマイズして自分たちのプロジェクトやワークフローにあてはめるための主要な方法を説明しました。 あらゆる設定項目やファイルベースの属性、そしてイベントフックについて学び、特定のポリシーを実現するサーバーを構築するサンプルを示しました。 これで、あなたが思い描くであろうほぼすべてのワークフローにあわせて Git を調整できるようになったはずです。

# Gitとその他のシステムの連携

世の中はそんなにうまくいくものではありません。 あなたが関わることになったプロジェクトで使うバージョン管理システムを、すぐさまGitに切り替えられることはほとんどないでしょう。 また、関わっているプロジェクトが他のVCSを使っていて、もしこれがGitだったらなぁと思うことも時々あると思います。 この章の前半では、作業中のプロジェクトが他のバージョン管理システムを使っていた場合に、Git をクライアントとして使用する方法を学びます。

どこかの時点で、既存のプロジェクトを Git へ変換したくなることもあるでしょう。 この章の後半では、いくつかの特定のシステムから Git へ、プロジェクトを移行する方法と、既存のインポート用ツールがない場合に使える手法について説明します。

## Git をクライアントとして使用する

Git は開発者に対し、非常に優れたユーザ体験を提供してくれます。このユーザ体験は、多くの人々がこれまでに編み出した、自分の端末上で Git を使用する方法に基づいています。それは、同じチームの他のメンバーがまったく別の VCS を使用している場合でも同様です。 そのような場合には “ブリッジ” と呼ばれるアダプタが利用できます。 ここでは、その中でも遭遇する機会が多いであろうものを取り上げます。

### Git と Subversion

オープンソース開発プロジェクトの大多数や、かなりの数の企業内プロジェクトが、ソースコードの管理に Subversion を利用しています。 Subversion は10年以上前から使われてきましたが、その間ほとんどの期間、オープンソースプロジェクトのVCSとしては *デファクトスタンダード* の地位にありました。 Subversion 以前は CVS がソースコード管理に広く用いられていたのですが、多くの点で両者はよく似ています。

Git の素晴しい機能のひとつに、Git と Subversion を双方向にブリッジする `git svn` があります。このツールを使うと、Subversion のクライアントとして Git を使うことができます。つまり、ローカルの作業では Git の機能を十分に活用することができて、あたかも Subversion を使っているかのように Subversion サーバーに変更をコミットすることができます。共同作業をしている人達が古き良き方法を使っているのと 同時に、ローカルでのブランチ作成やマージ、ステージング・エリア、リベース、チェリーピックなどの Git の機能を使うことができるということです。共同の作業環境に Git を忍び込ませておいて、仲間の開発者たちが Git より効率良く作業できるように手助けをしつつ、Git の全面的な採用のための根回しをしてゆく、というのが賢いやり方です。Subversion ブリッジは、分散VCS の素晴しい世界へのゲートウェイ・ドラッグといえるでしょう。

#### `git svn`

Git と Subversion の橋渡しをするコマンド群のベースとなるコマンドが `git svn` です。 この後に続くコマンドはかなりたくさんあるので、シンプルなワークフローを通してもっともよく使われるものから見ていきます。

注意すべきことは、`git svn` を使っているときは Subversion を相手にしているのだということです。これは、Git とはまったく異なる動きをします。 ローカルでのブランチ作成やマージは **できることはできます** が、作業内容をリベースするなどして歴史をできるだけ一直線に保つようにし、Git リモートリポジトリを相手にするときのように考えるのは避けましょう。

歴史を書き換えてもう一度プッシュしようなどとしてはいけません。また、他の開発者との共同作業のために複数の Git リポジトリに並行してプッシュするのもいけません。Subversion が扱えるのは一本の直線上の歴史だけで、ちょっとしたことですぐに混乱してしまいます。チームのメンバーの中に SVN を使う人と Git を使う人がいる場合は、全員が SVN サーバーを使って共同作業するようにしましょう。そうすれば、少しは生きやすくなります。

**セットアップ**

この機能を説明するには、書き込みアクセス権を持つ標準的な SVN リポジトリが必要です。 もしこのサンプルをコピーして試したいのなら、私のテスト用リポジトリの書き込み可能なコピーを作らなければなりません。 これを簡単に行うには、Subversion に付属の `svnsync` というツールを使います。 テスト用として、新しい Subversion リポジトリを Google Code 上に作りました。これは `protobuf` プロジェクトの一部で、`protobuf` は構造化されたデータを符号化してネットワーク上で転送するためのツールです。

まずはじめに、新しいローカル Subversion リポジトリを作ります。

```
$ mkdir /tmp/test-svn
$ svnadmin create /tmp/test-svn
```

そして、すべてのユーザーが revprop を変更できるようにします。簡単な方法は、常に 0 で終了する `pre-revprop-change` スクリプトを追加することです。

```
$ cat /tmp/test-svn/hooks/pre-revprop-change
#!/bin/sh
exit 0;
$ chmod +x /tmp/test-svn/hooks/pre-revprop-change
```

これで、ローカルマシンにこのプロジェクトを同期できるようになりました。同期元と同期先のリポジトリを指定して `svnsync init` を実行します。

```
$ svnsync init file:///tmp/test-svn \
  http://progit-example.googlecode.com/svn/
```

このコマンドは、同期を実行するためのプロパティを設定します。次に、このコマンドでコードをコピーします。

```
$ svnsync sync file:///tmp/test-svn
Committed revision 1.
Copied properties for revision 1.
Transmitting file data .............................[...]
Committed revision 2.
Copied properties for revision 2.
[…]
```

この操作は数分で終わりますが、もし元のリポジトリのコピー先がローカルではなく別のリモートリポジトリだった場合、総コミット数がたかだか 100 にも満たなかったとしても、この処理には約一時間かかります。Subversion では、リビジョンごとにクローンを作ってコピー先のリポジトリに投入していかなければなりません。これはばかばかしいほど非効率的ですが、簡単に済ませるにはこの方法しかないのです。

#### 使いはじめる

書き込み可能な Subversion リポジトリが手に入ったので、一般的なワークフローに沿って進めましょう。まずは `git svn clone` コマンドを実行します。このコマンドは、Subversion リポジトリ全体をローカルの Git リポジトリにインポートします。どこかにホストされている実際の Subversion リポジトリから取り込む場合は `file:///tmp/test-svn` の部分を Subversion リポジトリの URL に変更しましょう。

```
$ git svn clone file:///tmp/test-svn -T trunk -b branches -t tags
Initialized empty Git repository in /private/tmp/progit/test-svn/.git/
r1 = dcbfb5891860124cc2e8cc616cded42624897125 (refs/remotes/origin/trunk)
    A   m4/acx_pthread.m4
    A   m4/stl_hash.m4
    A   java/src/test/java/com/google/protobuf/UnknownFieldSetTest.java
    A   java/src/test/java/com/google/protobuf/WireFormatTest.java
…
r75 = 556a3e1e7ad1fde0a32823fc7e4d046bcfd86dae (refs/remotes/origin/trunk)
Found possible branch point: file:///tmp/test-svn/trunk =>
file:///tmp/test-svn/branches/my-calc-branch, 75
Found branch parent: (refs/remotes/origin/my-calc-branch)
556a3e1e7ad1fde0a32823fc7e4d046bcfd86dae
Following parent with do_switch
Successfully followed parent
r76 = 0fb585761df569eaecd8146c71e58d70147460a2 (refs/remotes/origin/my-
calc-branch)
Checked out HEAD:
  file:///tmp/test-svn/trunk r75
```

これは、指定した URL に対して `git svn init` に続けて `git svn fetch` を実行するのと同じ意味です。しばらく時間がかかります。 test プロジェクトには 75 のコミットしかなくてコードベースもそれほど大きくありませんが、Git は各バージョンをそれぞれチェックアウトしては個別にコミットしています。 もし数百数千のコミットがあるプロジェクトで試すと、終わるまでには数時間から下手をすると数日かかってしまうかもしれません。

`-T trunk -b branches -t tags` の部分は、この Subversion リポジトリが標準的なブランチとタグの規約に従っていることを表しています。trunk、branches、tags にもし別の名前をつけているのなら、この部分を変更します。この規約は一般に使われているものなので、単に `-s` とだけ指定することもできます。これは、先の 3 つのオプションを指定したのと同じ標準のレイアウトを表します。つまり、次のようにしても同じ意味になるということです。

```
$ git svn clone file:///tmp/test-svn -s
```

これで、ブランチやタグも取り込んだ Git リポジトリができあがりました。

```
$ git branch -a
* master
  remotes/origin/my-calc-branch
  remotes/origin/tags/2.0.2
  remotes/origin/tags/release-2.0.1
  remotes/origin/tags/release-2.0.2
  remotes/origin/tags/release-2.0.2rc1
  remotes/origin/trunk
```

このツールが Subversion のタグをリモート参照としてどのように管理しているかに注目してください。

Git の配管コマンド `show-ref` について、もう少し詳しく見ていきましょう。

```
$ git show-ref
556a3e1e7ad1fde0a32823fc7e4d046bcfd86dae refs/heads/master
0fb585761df569eaecd8146c71e58d70147460a2 refs/remotes/origin/my-calc-
branch
bfd2d79303166789fc73af4046651a4b35c12f0b refs/remotes/origin/tags/2.0.2
285c2b2e36e467dd4d91c8e3c0c0e1750b3fe8ca refs/remotes/origin/tags/release-
2.0.1
cbda99cb45d9abcb9793db1d4f70ae562a969f1e refs/remotes/origin/tags/release-
2.0.2
a9f074aa89e826d6f9d30808ce5ae3ffe711feda refs/remotes/origin/tags/release-
2.0.2rc1
556a3e1e7ad1fde0a32823fc7e4d046bcfd86dae refs/remotes/origin/trunk
```

Git が Git サーバからクローンを行う場合はこうはなりません。タグつきのリポジトリに対してクローンを行った直後は、このようになっています。

```
$ git show-ref
c3dcbe8488c6240392e8a5d7553bbffcb0f94ef0 refs/remotes/origin/master
32ef1d1c7cc8c603ab78416262cc421b80a8c2df refs/remotes/origin/branch-1
75f703a3580a9b81ead89fe1138e6da858c5ba18 refs/remotes/origin/branch-2
23f8588dde934e8f33c263c6d8359b2ae095f863 refs/tags/v0.1.0
7064938bd5e7ef47bfd79a685a62c1e2649e2ce7 refs/tags/v0.2.0
6dcb09b5b57875f334f61aebed695e2e4193db5e refs/tags/v1.0.0
```

Git は tags ディレクトリの内容をリモートブランチとして扱うのではなく、直接 `refs/tags` に格納しています。

#### Subversion へのコミットの書き戻し

作業リポジトリを手に入れたあなたはプロジェクト上で何らかの作業を終え、コミットを上流に書き戻すことになりました。Git を SVN クライアントとして使います。どれかひとつのファイルを変更してコミットした時点では、Git上でローカルに存在するそのコミットはSubversionサーバー上には存在しません。

```
$ git commit -am 'Adding git-svn instructions to the README'
[master 4af61fd] Adding git-svn instructions to the README
 1 file changed, 5 insertions(+)
```

次に、これをプッシュして上流を変更しなければなりません。この変更が Subversion に対してどのように作用するのかに注意しましょう。オフラインで行った複数のコミットを、すべて一度に Subversion サーバーにプッシュすることができます。Subversion サーバーにプッシュするには `git svn dcommit` コマンドを使います。

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...
    M   README.txt
Committed r77
    M   README.txt
r77 = 95e0222ba6399739834380eb10afcd73e0670bc5 (refs/remotes/origin/trunk)
No changes between 4af61fd05045e07598c553167e0f31c84fd6ffe1 and
refs/remotes/origin/trunk
Resetting to the latest refs/remotes/origin/trunk
```

このコマンドは、Subversionサーバーからのコード上で行われたすべてのコミットに対して個別に Subversion 上にコミットし、ローカルの Git のコミットを書き換えて一意な識別子を含むようにします。ここで重要なのは、書き換えによってすべてのローカルコミットの SHA-1 チェックサムが変化するということです。この理由もあって、Git ベースのリモートリポジトリにあるプロジェクトと Subversion サーバーを同時に使うことはおすすめできません。直近のコミットを調べれば、新たに `git-svn-id` が追記されたことがわかります。

```
$ git log -1
commit 95e0222ba6399739834380eb10afcd73e0670bc5
Author: ben <ben@0b684db3-b064-4277-89d1-21af03df0a68>
Date:   Thu Jul 24 03:08:36 2014 +0000

    Adding git-svn instructions to the README

    git-svn-id: file:///tmp/test-svn/trunk@77 0b684db3-b064-4277-89d1-
21af03df0a68
```

元のコミットの SHA-1 チェックサムが `4af61fd` で始まっていたのに対して今は `95e0222` に変わっていることに注目しましょう。Git と Subversion の両方のサーバーにプッシュしたい場合は、まず Subversion サーバーにプッシュ (`dcommit`) してから Git のほうにプッシュしなければなりません。dcommit でコミットデータが書き換わるからです。

**新しい変更の取り込み**

複数の開発者と作業をしていると、遅かれ早かれ、誰かがプッシュしたあとに他の人がプッシュしようとして衝突を起こすということが発生します。他の人の作業をマージするまで、その変更は却下されます。`git svn` では、このようになります。

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...

ERROR from SVN:
Transaction is out of date: File '/trunk/README.txt' is out of date
W: d5837c4b461b7c0e018b49d12398769d2bfc240a and refs/remotes/origin/trunk
differ, using rebase:
:100644 100644 f414c433af0fd6734428cf9d2a9fd8ba00ada145
c80b6127dd04f5fcda218730ddf3a2da4eb39138 M  README.txt
Current branch master is up to date.
ERROR: Not all changes have been committed into SVN, however the committed
ones (if any) seem to be successfully integrated into the working tree.
Please see the above messages for details.
```

この状態を解決するには `git svn rebase` を実行します。これは、サーバー上の変更のうちまだ取り込んでいない変更をすべて取り込んでから、自分の作業をリベースします。

```
$ git svn rebase
Committing to file:///tmp/test-svn/trunk ...

ERROR from SVN:
Transaction is out of date: File '/trunk/README.txt' is out of date
W: eaa029d99f87c5c822c5c29039d19111ff32ef46 and refs/remotes/origin/trunk
differ, using rebase:
:100644 100644 65536c6e30d263495c17d781962cfff12422693a
b34372b25ccf4945fe5658fa381b075045e7702a M  README.txt
First, rewinding head to replay your work on top of it...
Applying: update foo
Using index info to reconstruct a base tree...
M   README.txt
Falling back to patching base and 3-way merge...
Auto-merging README.txt
ERROR: Not all changes have been committed into SVN, however the committed
ones (if any) seem to be successfully integrated into the working tree.
Please see the above messages for details.
```

これで手元の作業が Subversion サーバー上の最新状態の上でなされたことになったので、無事に `dcommit` することができます。

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...
    M   README.txt
Committed r85
    M   README.txt
r85 = 9c29704cc0bbbed7bd58160cfb66cb9191835cd8 (refs/remotes/origin/trunk)
No changes between 5762f56732a958d6cfda681b661d2a239cc53ef5 and
refs/remotes/origin/trunk
Resetting to the latest refs/remotes/origin/trunk
```

ここで注意すべき点は、Git の場合は上流での変更をすべてマージしてからでなければプッシュできないけれど、`git svn` の場合は衝突さえしなければマージしなくてもプッシュできる（Subversion の挙動と同じように）ということです。 だれかがあるファイルを変更した後で自分が別のファイルを変更してプッシュしても、`dcommit` は正しく動作します。

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...
    M   configure.ac
Committed r87
    M   autogen.sh
r86 = d8450bab8a77228a644b7dc0e95977ffc61adff7 (refs/remotes/origin/trunk)
    M   configure.ac
r87 = f3653ea40cb4e26b6281cec102e35dcba1fe17c4 (refs/remotes/origin/trunk)
W: a0253d06732169107aa020390d9fefd2b1d92806 and refs/remotes/origin/trunk
differ, using rebase:
:100755 100755 efa5a59965fbbb5b2b0a12890f1b351bb5493c18
e757b59a9439312d80d5d43bb65d4a7d0389ed6d M  autogen.sh
First, rewinding head to replay your work on top of it...
```

これは忘れずに覚えておきましょう。というのも、プッシュした後の結果はどの開発者の作業環境にも存在しない状態になっているからです。たまたま衝突しなかっただけで互換性のない変更をプッシュしてしまったときに、その問題を見つけるのが難しくなります。これが、Git サーバーを使う場合と異なる点です。Git の場合はクライアントの状態をチェックしてからでないと変更を公開できませんが、SVN の場合はコミットの直前とコミット後の状態が同等であるかどうかすら確かめられないのです。

もし自分のコミット準備がまだできていなくても、Subversion から変更を取り込むときにもこのコマンドを使わなければなりません。`git svn fetch` でも新しいデータを取得することはできますが、`git svn rebase` はデータを取得するだけでなくローカルのコミットの更新も行います。

```
$ git svn rebase
    M   autogen.sh
r88 = c9c5f83c64bd755368784b444bc7a0216cc1e17b (refs/remotes/origin/trunk)
First, rewinding head to replay your work on top of it...
Fast-forwarded master to refs/remotes/origin/trunk.
```

`git svn rebase` をときどき実行しておけば、手元のコードを常に最新の状態に保っておけます。しかし、このコマンドを実行するときには作業ディレクトリがクリーンな状態であることを確認しておく必要があります。手元で変更をしている場合は、stash で作業を退避させるか一時的にコミットしてからでないと `git svn rebase` を実行してはいけません。さもないと、もしリベースの結果としてマージが衝突すればコマンドの実行が止まってしまいます。

**Git でのブランチに関する問題**

Git のワークフローに慣れてくると、トピックブランチを作ってそこで作業を行い、それをマージすることもあるでしょう。`git svn` を使って Subversion サーバーにプッシュする場合は、それらのブランチをまとめてプッシュするのではなく一つのブランチ上にリベースしてからプッシュしたくなるかもしれません。リベースしたほうがよい理由は、Subversion はリニアに歴史を管理していて Git のようなマージができないからです。`git svn` がスナップショットを Subversion のコミットに変換するときには、最初の親だけに続けます。

歴史が次のような状態になっているものとしましょう。`experiment` ブランチを作ってそこで 2 回のコミッ

トを済ませ、それを master にマージしたところです。ここで dcommit すると、出力はこのようになります。

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...
    M   CHANGES.txt
Committed r89
    M   CHANGES.txt
r89 = 89d492c884ea7c834353563d5d913c6adf933981 (refs/remotes/origin/trunk)
    M   COPYING.txt
    M   INSTALL.txt
Committed r90
    M   INSTALL.txt
    M   COPYING.txt
r90 = cb522197870e61467473391799148f6721bcf9a0 (refs/remotes/origin/trunk)
No changes between 71af502c214ba13123992338569f4669877f55fd and
refs/remotes/origin/trunk
Resetting to the latest refs/remotes/origin/trunk
```

歴史をマージしたブランチで dcommit を実行してもうまく動作します。ただし、Git プロジェクト上での歴史を見ると、experiment ブランチ上でのコミットは書き換えられていません。そこでのすべての変更は、SVN 上での単一のマージコミットとなっています。

他の人がその作業をクローンしたときには、 git merge --squash を実行したときのように、すべての作業をひとまとめにしたマージコミットしか見ることができません。そのコミットがどこから来たのか、そしていつコミットされたのかを知ることができないのです。

#### Subversion のブランチ

Subversion のブランチは Git のブランチとは異なります。可能ならば、Subversion のブランチは使わないようにするのがベストでしょう。 しかし、Subversion のブランチの作成やコミットも、git svn を使ってすることができます。

#### 新しい SVN ブランチの作成

Subversion に新たなブランチを作るには git svn branch [ブランチ名] を実行します。

```
$ git svn branch opera
Copying file:///tmp/test-svn/trunk at r90 to file:///tmp/test-
svn/branches/opera...
Found possible branch point: file:///tmp/test-svn/trunk =>
file:///tmp/test-svn/branches/opera, 90
Found branch parent: (refs/remotes/origin/opera)
cb522197870e61467473391799148f6721bcf9a0
Following parent with do_switch
Successfully followed parent
r91 = f1b64a3855d3c8dd84ee0ef10fa89d27f1584302 (refs/remotes/origin/opera)
```

これは Subversion の `svn copy trunk branches/opera` コマンドと同じ意味で、Subversion サーバー上で実行されます。ここで注意すべき点は、このコマンドを実行しても新しいブランチに入ったことにはならないということです。この後コミットをすると、そのコミットはサーバーの `trunk` に対して行われます。`opera` ではありません。

#### アクティブなブランチの切り替え

Git が dcommit の行き先のブランチを決めるときには、あなたの手元の歴史上にある Subversion ブランチのいずれかのヒントを使います。手元にはひとつしかないはずで、それは現在のブランチの歴史上の直近のコミットにある `git-svn-id` です。

複数のブランチを同時に操作するときは、ローカルブランチを `dcommit` でその Subversion ブランチにコミットするのかを設定することができます。そのためには、Subversion のブランチをインポートしてローカルブランチを作ります。`opera` ブランチを個別に操作したい場合は、このようなコマンドを実行します。

```
$ git branch opera remotes/origin/opera
```

これで、`opera` ブランチを `trunk` (手元の `master` ブランチ) にマージするときに通常の `git merge` が使えるようになりました。しかし、そのときには適切なコミットメッセージを (`-m` で) 指定しなければなりません。さもないと、有用な情報ではなく単なる "Merge branch opera" というメッセージになってしまいます。

`git merge` を使ってこの操作を行ったとしても、そしてそれが Subversion でのマージよりもずっと簡単だったとしても (Git は自動的に適切なマージベースを検出してくれるからね)、これは通常の Git のマージコミットとは違うということを覚えておきましょう。このデータを Subversion に書き戻すことになりますが Subversion では複数の親を持つコミットは処理できません。そのため、プッシュした後は、別のブランチ上で行ったすべての操作をひとまとめにした単一のコミットに見えてしまいます。あるブランチを別のブランチにマージしたら、元のブランチに戻って作業を続けるのは困難です。Git なら簡単なのですが。`dcommit` コマンドを実行すると、どのブランチからマージしたのかという情報はすべて消えてしまいます。そのため、それ以降のマージ元の算出は間違ったものとなります。dcommit は、`git merge` の結果をまるで `git merge --squash` を実行したのと同じ状態にしてしまうのです。残念ながら、これを回避するよい方法はありません。Subversion 側にこの情報を保持する方法がないからです。Subversion をサーバーに使う以上は、常にこの制約に縛られることになります。問題を回避するには、trunk にマージしたらローカルブランチ (この場合は `opera`) を削除しなければなりません。

### Subversion コマンド

`git svn` ツールセットには、Git への移行をしやすくするための多くのコマンドが用意されています。Subversion で使い慣れていたのと同等の機能を提供するコマンド群です。その中からいくつかを紹介します。

#### SVN 形式のログ

Subversion に慣れているので SVN が出力する形式で歴史を見たい、という場合は `git svn log` を実行しましょう。すると、コミットの歴史が SVN 形式で表示されます。

```
$ git svn log
------------------------------------------------------------------------
r87 | schacon | 2014-05-02 16:07:37 -0700 (Sat, 02 May 2014) | 2 lines

autogen change

------------------------------------------------------------------------
r86 | schacon | 2014-05-02 16:00:21 -0700 (Sat, 02 May 2014) | 2 lines

Merge branch 'experiment'

------------------------------------------------------------------------
r85 | schacon | 2014-05-02 16:00:09 -0700 (Sat, 02 May 2014) | 2 lines

updated the changelog
```

`git svn log` に関して知っておくべき重要なことがふたつあります。まず。このコマンドはオフラインで動作します。実際の `svn log` コマンドのように Subversion サーバーにデータを問い合わせたりしません。次に、すでに Subversion サーバーにコミット済みのコミットしか表示されません。つまり、ローカルの Git へのコミットのうちまだ dcommit していないものは表示されないし、その間に他の人が Subversion サーバーにコミットした内容も表示されません。最後に Subversion サーバーの状態を調べたときのログが表示されると考えればよいでしょう。

#### SVN アノテーション

`git svn log` コマンドが `svn log` コマンドをオフラインでシミュレートしているのと同様に、`svn annotate` と同様のことを `git svn blame [FILE]` で実現できます。出力は、このようになります。

```
$ git svn blame README.txt
 2   temporal Protocol Buffers - Google's data interchange format
 2   temporal Copyright 2008 Google Inc.
 2   temporal http://code.google.com/apis/protocolbuffers/
 2   temporal
22   temporal C++ Installation - Unix
22   temporal =======================
 2   temporal
79    schacon Committing in git-svn.
78    schacon
 2   temporal To build and install the C++ Protocol Buffer runtime and the
Protocol
 2   temporal Buffer compiler (protoc) execute the following:
 2   temporal
```

先ほどと同様、このコマンドも Git にローカルにコミットした内容や他から Subversion にプッシュされていたコミットは表示できません。

#### SVN サーバ情報

`svn info` と同様のサーバー情報を取得するには `git svn info` を実行します。

```
$ git svn info
Path: .
URL: https://schacon-test.googlecode.com/svn/trunk
Repository Root: https://schacon-test.googlecode.com/svn
Repository UUID: 4c93b258-373f-11de-be05-5f7a86268029
Revision: 87
Node Kind: directory
Schedule: normal
Last Changed Author: schacon
Last Changed Rev: 87
Last Changed Date: 2009-05-02 16:07:37 -0700 (Sat, 02 May 2009)
```

`blame` や `log` と同様にこれもオフラインで動作し、最後に Subversion サーバーと通信した時点での情報しか表示されません。

#### Subversion が無視するものを無視する

どこかに `svn:ignore` プロパティが設定されている Subversion リポジトリをクローンした場合は、対応する `.gitignore` ファイルを用意したくなることでしょう。コミットすべきではないファイルを誤ってコミットしてしまうことを防ぐためにです。`git svn` には、この問題に対応するためのコマンドが二つ用意されています。まず最初が `git svn create-ignore` で、これは、対応する `.gitignore` ファイルを自動生成して次のコミットに含めます。

もうひとつは `git svn show-ignore` で、これは `.gitignore` に書き込む内容を標準出力に送ります。こ

の出力を、プロジェクトの exclude ファイルにリダイレクトしましょう。

```
$ git svn show-ignore > .git/info/exclude
```

これで、プロジェクトに `.gitignore` ファイルを散らかさなくてもよくなります。Subversion 使いのチームの中で Git を使うのが自分だけだという場合、他のメンバーにとっては `.gitignore` ファイルは目障りでしょう。そのような場合はこの方法が使えます。

#### Git-Svn のまとめ

`git svn` ツール群は、Subversion サーバーに行き詰まっている場合や使っている開発環境が Subversion サーバー前提になっている場合などに便利です。Git のできそこないだと感じるかもしれません。また、他のメンバーとの間で混乱が起こるかもしれません。トラブルを避けるために、次のガイドラインに従いましょう。

- Git の歴史をリニアに保ち続け、`git merge` によるマージコミットを含めないようにする。本流以外のブランチでの作業を書き戻すときは、マージではなくリベースすること。
- Git サーバーを別途用意したりしないこと、新しい開発者がクローンするときのスピードをあげるためにサーバーを用意することはあるでしょうが、そこに `git-svn-id` エントリを持たないコミットをプッシュしてはいけません。`pre-receive` フックを追加してコミットメッセージをチェックし、`git-svn-id` がなければプッシュを拒否するようにしてもよいでしょう。

これらのガイドラインを守れば、Subversion サーバーでの作業にも耐えられることでしょう。しかし、もし本物の Git サーバーに移行できるのなら、そうしたほうがチームにとってずっと利益になります。

### Git と Mercurial

DVCSの世界にあるのはGitだけではありません。 事実、Git以外にも様々なシステムが存在し、分散バージョン管理を正しく行う方法について、それぞれが独自の見方を持っています。 Gitを除くと、もっとも広く使われているのは Mercurial です。Git と Mercurialは多くの点で似通っています。

良いニュースとして、 Git のクライアントサイドの動作がお好みで、しかし作業中のプロジェクトでは Mercurial でソースを管理しているという場合、 Mercurial でホストされているリポジトリのクライアントに Git を使用するという方法があります。 Git はリモートを通してサーバリポジトリとやりとりしているので、このブリッジがリモートヘルパーとして実装されているのは驚くほどのことでもないと思います。 プロジェクト名は git-remote-hg で、 https://github.com/felipec/git-remote-hg から取得できます。

#### git-remote-hg

まず、 git-remote-hg をインストールする必要があります。 ここでは基本的に、そのファイルをどこかパスの通った場所に置く必要があります。以下のような感じです。

```
$ curl -o ~/bin/git-remote-hg \
  https://raw.githubusercontent.com/felipec/git-remote-hg/master/git-
remote-hg
$ chmod +x ~/bin/git-remote-hg
```

…ここでは `~/bin` が `$PATH` に含まれていることを仮定しています。 git-remote-hg にはもう一つ依存先があります。 Python の `mercurial` ライブラリです。 Python をインストール済みなら、これは次のようにシンプルなコマンドで行えます。

```
$ pip install mercurial
```

（Python をインストールしていないなら、まず https://www.python.org/ からPython を入手してください。）

最後に必要なのは Mercurial のクライアントです。 インストール済みでないなら、http://mercurial.selenic.com/ から入手してインストールしてください。

これで準備が整いました。 必要なのはプッシュが可能な Mercurial リポジトリだけです。 幸いなことに、Mercurial リポジトリならどれでもこの操作が可能です。そのため、 Mercurial の使い方を学ぶときにみんなが使う "hello world" リポジトリを使用することにします。

```
$ hg clone http://selenic.com/repo/hello /tmp/hello
```

#### 使いはじめる

これで、都合のいい“サーバサイド”のリポジトリができたので、以降では典型的なワークフローを見ていきます。 これから見ていくように、 Git と Mercurial はよく似ているため、食い違う箇所はそう多くありません。

Git でいつもやるように、まずクローンをします。

```
$ git clone hg::/tmp/hello /tmp/hello-git
$ cd /tmp/hello-git
$ git log --oneline --graph --decorate
* ac7955c (HEAD, origin/master, origin/branches/default, origin/HEAD,
refs/hg/origin/branches/default, refs/hg/origin/bookmarks/master, master)
Create a makefile
* 65bb417 Create a standard "hello, world" program
```

標準的な `git clone` コマンドを使用して Mercurial リポジトリを操作しているのが分かると思います。 これは git-remote-hg が非常に低いレベルで動作しているためです。 git-remote-hg は Git が HTTP/S プロトコルを実装しているのと同じようなメカニズム（リモートヘルパー）を使用しています。 Git と Mercurial は両方とも、すべてのクライアントがリポジトリの歴史の完全なコピーを持つように設計されています。そのためこのコマンドは、プロジェクトのすべての歴史を含む完全なクローンを作成します。また、この処理は非常に高

速に行われます。

git logコマンドは2つのコミットを表示しています。最新のコミットは大量の参照から指されています。 実は、これらの中のいくつかは、実際には存在しません。 `.git` ディレクトリの中に実際には何が入っているか見てみましょう。

```
$ tree .git/refs
.git/refs
├── heads
│   └── master
├── hg
│   └── origin
│       ├── bookmarks
│       │   └── master
│       └── branches
│           └── default
├── notes
│   └── hg
├── remotes
│   └── origin
│       └── HEAD
└── tags

9 directories, 5 files
```

git-remote-hg は、物事をより Git 風にしようとしているわけですが、内部的には、2つの微妙に異なるシステムの間のマッピングを管理しています。 `refs/hg` ディレクトリには実際のリモート参照が格納されています。 例えば、 `refs/hg/origin/branches/default` は “ac7955c” で始まるSHA-1（`master` が指しているコミットを表している）を含むGitの参照ファイルです。 そのため、 `refs/hg` ディレクトリは `refs/remotes/origin` の模造品のようなものとも言えます。ただし、ブックマークとブランチの区別が追加されています。

`notes/hg` ファイルは、 git-remote-hg が Git のコミットハッシュと Mercurial のチェンジセットIDを対応付ける際の開始点となります。 ちょっと見てみましょう。

```
$ cat notes/hg
d4c10386...

$ git cat-file -p d4c10386...
tree 1781c96...
author remote-hg <> 1408066400 -0800
committer remote-hg <> 1408066400 -0800

Notes for master

$ git ls-tree 1781c96...
100644 blob ac9117f...  65bb417...
100644 blob 485e178...  ac7955c...

$ git cat-file -p ac9117f
0a04b987be5ae354b710cefeba0e2d9de7ad41a9
```

`refs/notes/hg` は Git オブジェクトデータベース中にあるツリーを指しており、その内容は他のオブジェクトの名前つきリストになっています。 `git ls-tree` はツリー中の要素のモード、タイプ、オブジェクトハッシュ、およびファイル名を出力します。 ツリー中の要素の一つについて掘り下げていくと、その実体は“ac9117f”（`master` が指しているコミットの SHA-1 ハッシュ）という名前の blob で、内容は“0a04b98”（`default` ブランチの先端の Mercurial チェンジセットのID）であることが分かります。

よいニュースとして、これらすべてのことについて、我々が気にする必要はほとんどありません。 典型的なワークフローは、 Git でリモートに対して作業をする場合と大差ありません。

以降の話をする前に、もう一つ注意しておかなければならないことがあります。 ignoreファイルです。 Mercurial と Git はこの点について非常に似通ったメカニズムを使用しています。ですが、おそらく実際に `.gitignore` ファイルを Mercurial リポジトリへコミットしたい、ということはないでしょう。 幸いなことに、 Git にはローカルからディスク上のリポジトリへファイルを登録する際に、指定したファイルを無視する方法があります。Mercurial のフォーマットは Git と互換性があるので、単にファイルをコピーするだけで済みます。

```
$ cp .hgignore .git/info/exclude
```

`.git/info/exclude` ファイルは `.gitignore` と同様の働きをしますが、コミットには含まれません。

#### ワークフロー

現在、何らかの作業をやり終え、 `master` ブランチにはコミットがいくつか作成されており、それをリモートリポジトリへプッシュできる状態にあるとしましょう。 現時点では、リポジトリは次のような内容になっています。

```
$ git log --oneline --graph --decorate
* ba04a2a (HEAD, master) Update makefile
* d25d16f Goodbye
* ac7955c (origin/master, origin/branches/default, origin/HEAD,
refs/hg/origin/branches/default, refs/hg/origin/bookmarks/master) Create a
makefile
* 65bb417 Create a standard "hello, world" program
```

`master` ブランチは `origin/master` よりコミット2つぶん進んでいますが、これら2つのコミットはローカルのマシン上にしかありません。 ここで、誰か他の人が、何か重要な作業をこれと同時に行っていたらどうなるか見てみましょう。

```
$ git fetch
From hg::/tmp/hello
   ac7955c..df85e87  master     -> origin/master
   ac7955c..df85e87  branches/default -> origin/branches/default
$ git log --oneline --graph --decorate --all
* 7b07969 (refs/notes/hg) Notes for default
* d4c1038 Notes for master
* df85e87 (origin/master, origin/branches/default, origin/HEAD,
refs/hg/origin/branches/default, refs/hg/origin/bookmarks/master) Add some
documentation
| * ba04a2a (HEAD, master) Update makefile
| * d25d16f Goodbye
|/
* ac7955c Create a makefile
* 65bb417 Create a standard "hello, world" program
```

`--all` フラグを指定したため、“notes” 参照が表示されていますが、これは git-remote-hg が内部的に使用しているものなので、無視して構いません。 残りが期待していた内容です。 `origin/master` はコミット1つぶん進んでおり、現在この歴史は枝分かれした状態にあります。 この章で扱っている他のシステムと異なり、Mercurial にはマージをハンドリングする機能が備わっているので、ややこしいことをする必要は何もありません。

```
$ git merge origin/master
Auto-merging hello.c
Merge made by the 'recursive' strategy.
 hello.c | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
$ git log --oneline --graph --decorate
*   0c64627 (HEAD, master) Merge remote-tracking branch 'origin/master'
|\
| * df85e87 (origin/master, origin/branches/default, origin/HEAD,
refs/hg/origin/branches/default, refs/hg/origin/bookmarks/master) Add some
documentation
* | ba04a2a Update makefile
* | d25d16f Goodbye
|/
* ac7955c Create a makefile
* 65bb417 Create a standard "hello, world" program
```

完璧です。 テストを実行して、結果はすべて正常でした。これで、成果物をチームの他のメンバーと共有できる状態になりました。

```
$ git push
To hg::/tmp/hello
   df85e87..0c64627  master -> master
```

これで完了です！ Mercurial リポジトリを見てみれば、期待していた内容が分かるはずです。

```
$ hg log -G --style compact
o    5[tip]:4,2   dc8fa4f932b8   2014-08-14 19:33 -0700   ben
|\     Merge remote-tracking branch 'origin/master'
| |
| o  4   64f27bcefc35   2014-08-14 19:27 -0700   ben
| |    Update makefile
| |
| o  3:1   4256fc29598f   2014-08-14 19:27 -0700   ben
| |    Goodbye
| |
@ |  2   7db0b4848b3c   2014-08-14 19:30 -0700   ben
|/     Add some documentation
|
o  1   82e55d328c8c   2005-08-26 01:21 -0700   mpm
|    Create a makefile
|
o  0   0a04b987be5a   2005-08-26 01:20 -0700   mpm
     Create a standard "hello, world" program
```

番号 *2* のチェンジセットは Mercurial が作成したもので、番号 *3* および *4* のチェンジセットは Git で作成したコミットを git-remote-hg がプッシュして作成したものです。

### ブランチとブックマーク

Git のブランチは1種類しかありません。コミットに合わせて移動する参照です。 Mercurial では、この種の参照は “ブックマーク” と呼ばれており、 Git のブランチとほぼ同じように振る舞います。

Mercurial の言う “ブランチ” は Git のそれよりもっと重量級の概念です。 ブランチの上でチェンジセットが作成された場合、ブランチは *チェンジセットと一緒に* 記録されます。つまり、常にリポジトリの歴史に残ります。 `develop` ブランチの上で作成されたコミットの例を次に示します。

```
$ hg log -l 1
changeset:   6:8f65e5e02793
branch:      develop
tag:         tip
user:        Ben Straub <ben@straub.cc>
date:        Thu Aug 14 20:06:38 2014 -0700
summary:     More documentation
```

“branch” で始まる行に注目してください。 Git はこれを完全に複製することはできません（また、する必要もありません。いずれのタイプのブランチも Git では参照として表現されるため）が、 Mercurial にとってはこの違いが問題となるため、 git-remote-hg はこの違いを理解している必要があります。

Mercurial のブックマークを作成するのは、 Git のブランチを作成するのと同様に簡単です。 Git 側では、

```
$ git checkout -b featureA
Switched to a new branch 'featureA'
$ git push origin featureA
To hg::/tmp/hello
 * [new branch]      featureA -> featureA
```

これだけです。 Mercurial 側では、これは次のように見えます。

```
$ hg bookmarks
   featureA                  5:bd5ac26f11f9
$ hg log --style compact -G
@  6[tip]   8f65e5e02793   2014-08-14 20:06 -0700   ben
|    More documentation
|
o    5[featureA]:4,2   bd5ac26f11f9   2014-08-14 20:02 -0700   ben
|\     Merge remote-tracking branch 'origin/master'
| |
| o  4   0434aaa6b91f   2014-08-14 20:01 -0700   ben
| |    update makefile
| |
| o  3:1   318914536c86   2014-08-14 20:00 -0700   ben
| |    goodbye
| |
o |  2   f098c7f45c4f   2014-08-14 20:01 -0700   ben
|/     Add some documentation
|
o  1   82e55d328c8c   2005-08-26 01:21 -0700   mpm
|    Create a makefile
|
o  0   0a04b987be5a   2005-08-26 01:20 -0700   mpm
     Create a standard "hello, world" program
```

リビジョン5に付いている新しいタグ `[featureA]` に注目してください。 これらの挙動は Git 側から見ると Git のブランチと非常によく似ていますが、一つ例外があります。 Git の側からブックマークを削除することはできません（これはリモートヘルパーの制限によります）。

“重量級” の Mercurial のブランチ上で作業をすることもできます。 `branches` 名前空間にブランチを追加します。

```
$ git checkout -b branches/permanent
Switched to a new branch 'branches/permanent'
$ vi Makefile
$ git commit -am 'A permanent change'
$ git push origin branches/permanent
To hg::/tmp/hello
 * [new branch]      branches/permanent -> branches/permanent
```

Mercurial 側ではこれは次のように見えます。

```
$ hg branches
permanent                      7:a4529d07aad4
develop                        6:8f65e5e02793
default                        5:bd5ac26f11f9 (inactive)
$ hg log -G
o  changeset:   7:a4529d07aad4
|  branch:      permanent
|  tag:         tip
|  parent:      5:bd5ac26f11f9
|  user:        Ben Straub <ben@straub.cc>
|  date:        Thu Aug 14 20:21:09 2014 -0700
|  summary:     A permanent change
|
| @  changeset:   6:8f65e5e02793
|/   branch:      develop
|    user:        Ben Straub <ben@straub.cc>
|    date:        Thu Aug 14 20:06:38 2014 -0700
|    summary:     More documentation
|
o    changeset:   5:bd5ac26f11f9
|\   bookmark:    featureA
| |  parent:      4:0434aaa6b91f
| |  parent:      2:f098c7f45c4f
| |  user:        Ben Straub <ben@straub.cc>
| |  date:        Thu Aug 14 20:02:21 2014 -0700
| |  summary:     Merge remote-tracking branch 'origin/master'
[...]
```

“permanent” という名前のブランチが、 *7* とマークされたチェンジセットと一緒に記録されています。

Git 側では、いずれのタイプのブランチで作業をするのも変わりません。普段と同じように、チェックアウト、コミット、フェッチ、マージ、プル、プッシュが行えます。 一つ知っておくべきこととして、 Mercurial は歴史の書き換えをサポートしておらず、追記しか行えません。 対話的リベースと強制プッシュを行うと、 Mercurial リポジトリは次のような内容になります。

```
$ hg log --style compact -G
o  10[tip]   99611176cbc9   2014-08-14 20:21 -0700   ben
|    A permanent change
|
o  9   f23e12f939c3   2014-08-14 20:01 -0700   ben
|    Add some documentation
|
o  8:1   c16971d33922   2014-08-14 20:00 -0700   ben
|    goodbye
|
| o  7:5   a4529d07aad4   2014-08-14 20:21 -0700   ben
| |    A permanent change
| |
| | @  6   8f65e5e02793   2014-08-14 20:06 -0700   ben
| |/     More documentation
| |
| o    5[featureA]:4,2   bd5ac26f11f9   2014-08-14 20:02 -0700   ben
| |\     Merge remote-tracking branch 'origin/master'
| | |
| | o  4   0434aaa6b91f   2014-08-14 20:01 -0700   ben
| | |    update makefile
| | |
+---o  3:1   318914536c86   2014-08-14 20:00 -0700   ben
| |      goodbye
| |
| o  2   f098c7f45c4f   2014-08-14 20:01 -0700   ben
|/     Add some documentation
|
o  1   82e55d328c8c   2005-08-26 01:21 -0700   mpm
|    Create a makefile
|
o  0   0a04b987be5a   2005-08-26 01:20 -0700   mpm
     Create a standard "hello, world" program
```

チェンジセット *8*、*9*、*10* が作成され、`permanent` ブランチに紐付けられていますが、古いチェンジセットも残っています。 これは Mercurial を使用している他のチームメンバーを **かなり** 混乱させるので、できる限り避けましょう。

#### Mercurial のまとめ

Git と Mercurial は非常に似通っており、相互に作業してもほとんど苦になりません。 （一般的に推奨されているように）あなたのマシン上にある歴史を変更しないようにさえしていれば、相手側にあるのが Mercurial であることを意識する必要もありません。

## Git と Perforce

Perforce は企業内では非常によく使われているバージョン管理システムです。 Perforce が登場したのは 1995 年で、この章で扱う中ではもっとも古いシステムです。 そしてその言葉通り、Perforce は当時の制約に合わせて設計されています。単一の中央サーバへの常時接続を仮定しており、またローカルディスクに保存しておけるバージョンは一つだけです。 確かに、Perforce の機能と制約は、ある特定の問題にはうまく適合します。しかし、実際には Git の方が上手くいくにも関わらず、 Perforce を使用しているというプロジェクトも数多くあります。

Perforce と Git を混在して使用したい場合、2通りの選択肢があります。 1つ目に取り上げるのは Perforce の開発元から出ている “Git Fusion” ブリッジです。これは、 Perforce ディポのサブツリーを読み書き可能な Git リポジトリとして公開させるものです。 2つ目はクライアントサイドのブリッジである git-p4 です。これは Git を Perforce のクライアントとして使用できるようにするもので、 Perforce サーバの設定を変更することなく使用できます。

### Git Fusion

Perforce は Git Fusion という製品を提供しています（ http://www.perforce.com/git-fusion から入手可能）。これは、サーバサイドで Perforce サーバと Git リポジトリとを同期させます。

#### セットアップ

本書の例では、 Git Fusion のもっとも簡単なインストール法として、仮想マシンをダウンロードし、Perforce デーモンと Git Fusion をその上で実行する方法をとります。 仮想マシンイメージは http://www.perforce.com/downloads/Perforce/20-User から入手できます。ダウンロードが完了したら、お好みの仮想ソフトへインポートします（本書では VirtualBox を使用します）。

仮想マシンの初回起動時には、3つの Linux ユーザ（ `root`、 `perforce` 、および `git` ）のパスワードを設定するよう要求されます。また、同じネットワーク上の他の仮想マシンとの区別のため、インスタンス名を決めるよう要求されます。 これらすべてが完了したら、次の画面が表示されるはずです。

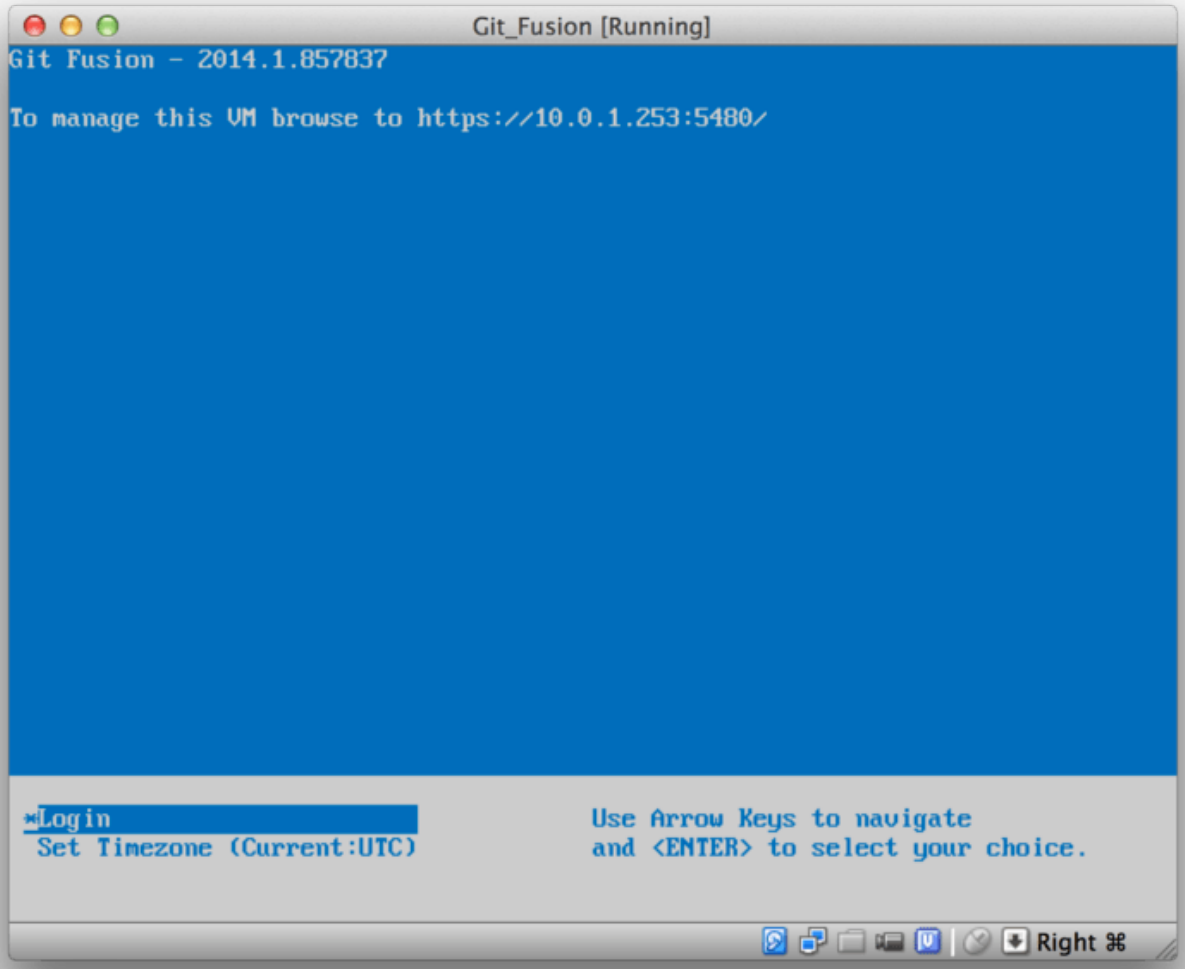



*Figure 145. Git Fusion 仮想マシンのブート画面*

ここで表示されるIPアドレスは、後で使用するので、メモしておいてください。 次に、 Perforce ユーザを作成します。 下部にある “Login” オプションを選択肢、Enterキーを押下し（または仮想マシンへSSHで接続し）、 `root` としてログインします。 続けて、次のコマンドでユーザを作成します。

```
$ p4 -p localhost:1666 -u super user -f john
$ p4 -p localhost:1666 -u john passwd
$ exit
```

1つめのコマンドは、ユーザのカスタマイズのため VI エディタを起動しますが、 `:wq` に続けて Enter を入力すれば、デフォルト設定のまま利用することもできます。 2つめのコマンドは、パスワードを2度入力するようプロンプトを表示します。 シェルプロンプトで行う必要のある作業はこれで全部ですので、セッションを終了します。

次に手順に従って行う必要があるのは、Git が SSL 証明書を検証しないようにすることです。 Git Fusion の仮想マシンイメージには証明書が同梱されていますが、これはあなたの仮想マシンのIPアドレスとは合わないで

あろうドメインのものなので、 Git は HTTPS 接続を拒否してしまいます。
今回インストールした環境を今後も使い続けるつもりなら、 Perforce Git Fusion マニュアルを参考に、個別の証明書をインストールしてください。本書で例を示すだけの用途なら、以下で十分です。

```
$ export GIT_SSL_NO_VERIFY=true
```

これで、すべてが動作しているかテストできるようになりました。

```
$ git clone https://10.0.1.254/Talkhouse
Cloning into 'Talkhouse'...
Username for 'https://10.0.1.254': john
Password for 'https://john@10.0.1.254':
remote: Counting objects: 630, done.
remote: Compressing objects: 100% (581/581), done.
remote: Total 630 (delta 172), reused 0 (delta 0)
Receiving objects: 100% (630/630), 1.22 MiB | 0 bytes/s, done.
Resolving deltas: 100% (172/172), done.
Checking connectivity... done.
```

仮想マシンイメージには、クローンできるサンプルプロジェクトが同梱されています。 ここでは、上で作成したユーザ `john` を使用し、 HTTPS 経由でクローンしています。今回の接続時には認証情報を要求されますが、以降のリクエストでは Git の認証情報キャッシュが働くので、このステップは省略できます。

#### Fusion の設定

Git Fusion をインストールし終わったら、設定を調整したいことと思います。 設定の変更は、お好きな Perforce クライアントを使用して、実際非常に簡単に行えます。Perforce サーバの `//.git-fusion` ディレクトリをワークスペースにマップするだけです。 ファイル構造は次のようになっています。

```
$ tree
.
├── objects
│   ├── repos
│   │   └── [...]
│   └── trees
│       └── [...]
│
├── p4gf_config
├── repos
│   └── Talkhouse
│       └── p4gf_config
└── users
    └── p4gf_usermap

498 directories, 287 files
```

`objects` ディレクトリは Perforce のオブジェクトを Git へ対応付ける（逆も同様）ために Git Fusion が内部的に使用しているもので、この内容に触れる必要はありません。 このディレクトリにはグローバルな `p4gf_config` 設定ファイルがあります。また、このファイルは各リポジトリにも一つずつあります – これらは、 Git Fusion の動作を決定する設定ファイルです。 ルートディレクトリにあるファイルを見てみましょう。

```
[repo-creation]
charset = utf8

[git-to-perforce]
change-owner = author
enable-git-branch-creation = yes
enable-swarm-reviews = yes
enable-git-merge-commits = yes
enable-git-submodules = yes
preflight-commit = none
ignore-author-permissions = no
read-permission-check = none
git-merge-avoidance-after-change-num = 12107

[perforce-to-git]
http-url = none
ssh-url = none

[@features]
imports = False
chunked-push = False
matrix2 = False
parallel-push = False

[authentication]
email-case-sensitivity = no
```

ここでは各フラグの意味については説明しませんが、このファイルが、 Git の設定ファイル同様、単なる INI ファイル形式のテキストファイルであるという点は明記しておきます。 このファイルではグローバルなオプションを設定しています。これらの設定は `repos/Talkhouse/p4gf_config` などのリポジトリ固有の設定で上書きできます。 このファイルを開くと、 `[@repo]` セクションに、グローバルなデフォルト値とは異なる設定がされているのが分かると思います。 また、次のようなセクションがあると思います。

```
[Talkhouse-master]
git-branch-name = master
view = //depot/Talkhouse/main-dev/... ...
```

これは Perforce のブランチと Git のブランチのマッピングです。 このセクションの名前は好きなように決められるので、一意になるように長い名前も付けられます。 `git-branch-name` を使えば、Git にとってはとても長いディポのパスを、より扱いやすい名前に変換できます。 `view` では、 Perforce のファイルが Git のリポ

ジトリへどう対応するかを、通常のビュー・マッピング用のシンタックスで設定します。
複数のマッピングを指定することもできます。次に例を示します。

```
[multi-project-mapping]
git-branch-name = master
view = //depot/project1/main/... project1/...
       //depot/project2/mainline/... project2/...
```

通常のワークスペースのマッピングが、ディレクトリの構造の変更を含む場合、この方法では、それも含めてGitリポジトリを複製することができます。

最後に取り上げるのは `users/p4gf_usermap` で、これは Perforce のユーザを Git のユーザにマッピングするファイルですが、必要ないかもしれません。 Perforce のチェンジセットを Git のコミットへ変換する際、Git Fusion のデフォルトの動作では、 Perforce ユーザを探して、そのメールアドレスとフルネームを Git の作成者/コミッターフィールドに使用します。 逆の方向に変換する場合、デフォルトでは Git の作成者フィールドに格納されているメールアドレスで Perforce ユーザを検索し、そのユーザとしてチェンジセットを送信します（パーミッションも適用されます）。 ほとんどの場合、この動作で上手くいきます。ただし、次のマッピングファイルについても考慮しておいてください。

```
john john@example.com "John Doe"
john johnny@appleseed.net "John Doe"
bob employeeX@example.com "Anon X. Mouse"
joe employeeY@example.com "Anon Y. Mouse"
```

各行のフォーマットは `<ユーザ名> <メールアドレス> "<氏名>"` となっています。一行ごとに一つ、ユーザの対応づけを定義しています。 最初の2行は、2つの異なるメールアドレスを同一の Perforce ユーザアカウントへ対応づけています。 これは、 Git のコミットを複数のメールアドレスで作成していた（または、メールアドレスを変更した）際に、それらを同じ Perforce へ対応づけたい場合に便利です。 Perforce のチェンジセットから Git のコミットを作成する際には、 Perforce のユーザとマッチした最初の行が Git の作成者情報として使用されます。

最後の2行は、Git のコミットから Bob と Joe の氏名とメールアドレスが分からないようにします。 これは、社内のプロジェクトをオープンソース化したいが、社員名簿を全世界へ晒したくはない、というときに役立ちます。 注意すべき点として、すべての Git コミットが実際には存在しない1人のユーザによるものである、としたい場合を除き、メールアドレスと氏名は一意になるよう設定してください。

#### ワークフロー

Perforce Git Fusion は Perforce と Git の間の双方向ブリッジです。 Gitの側から作業するとどんな感じなのかを見てみましょう。 ここでは、前述した設定ファイルを使用して “Jam” プロジェクトをマッピングしたと仮定しましょう。次のようにクローンが行えます。

```
$ git clone https://10.0.1.254/Jam
Cloning into 'Jam'...
Username for 'https://10.0.1.254': john
Password for 'https://ben@10.0.1.254':
remote: Counting objects: 2070, done.
remote: Compressing objects: 100% (1704/1704), done.
Receiving objects: 100% (2070/2070), 1.21 MiB | 0 bytes/s, done.
remote: Total 2070 (delta 1242), reused 0 (delta 0)
Resolving deltas: 100% (1242/1242), done.
Checking connectivity... done.
$ git branch -a
* master
  remotes/origin/HEAD -> origin/master
  remotes/origin/master
  remotes/origin/rel2.1
$ git log --oneline --decorate --graph --all
* 0a38c33 (origin/rel2.1) Create Jam 2.1 release branch.
| * d254865 (HEAD, origin/master, origin/HEAD, master) Upgrade to latest
metrowerks on Beos -- the Intel one.
| * bd2f54a Put in fix for jam's NT handle leak.
| * c0f29e7 Fix URL in a jam doc
| * cc644ac Radstone's lynx port.
[...]
```

最初にこれを実行した際には、ちょっと時間がかかるかもしれません。 ここで何が行われているかというと、 Git Fusion が、 Perforce の歴史中にある、適用可能なチェンジセットをすべて Git のコミットへ変換しています。 この処理はサーバ内部で行われるので、比較的高速ですが、大量の歴史がある場合には、ちょっと時間がかかります。 以降のフェッチでは増分だけを変換するので、体感的に Git 本来のスピードにより近づきます。

見て分かるとおり、このリポジトリは普段作業している Git リポジトリと見た目上まったく変わりません。 3 つのブランチがあり、 Git は親切なことに `origin/master` ブランチを追跡するローカルの `master` ブランチまで作成してくれています。 ちょっと作業をして、新しいコミットを2つほど作成してみましょう。

```
# ...
$ git log --oneline --decorate --graph --all
* cfd46ab (HEAD, master) Add documentation for new feature
* a730d77 Whitespace
* d254865 (origin/master, origin/HEAD) Upgrade to latest metrowerks on
Beos -- the Intel one.
* bd2f54a Put in fix for jam's NT handle leak.
[...]
```

新しいコミットが2つできました。 今度は、他の人が作業していなかったか確認してみましょう。

```
$ git fetch
remote: Counting objects: 5, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 2), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From https://10.0.1.254/Jam
   d254865..6afeb15  master     -> origin/master
$ git log --oneline --decorate --graph --all
* 6afeb15 (origin/master, origin/HEAD) Update copyright
| * cfd46ab (HEAD, master) Add documentation for new feature
| * a730d77 Whitespace
|/
* d254865 Upgrade to latest metrowerks on Beos -- the Intel one.
* bd2f54a Put in fix for jam's NT handle leak.
[...]
```

誰かいたみたいですよ！ このビューからは分からなかったかも知れませんが、コミット `6afeb15` は Perforce クライアントを使用して実際に作成されたものです。 Git の視点から見ると、他のコミットと変わりませんが、 そこがポイントです。 Perforce サーバがマージコミットをどのように処理するのかを見てみましょう。

```
$ git merge origin/master
Auto-merging README
Merge made by the 'recursive' strategy.
 README | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
$ git push
Counting objects: 9, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (9/9), done.
Writing objects: 100% (9/9), 917 bytes | 0 bytes/s, done.
Total 9 (delta 6), reused 0 (delta 0)
remote: Perforce: 100% (3/3) Loading commit tree into memory...
remote: Perforce: 100% (5/5) Finding child commits...
remote: Perforce: Running git fast-export...
remote: Perforce: 100% (3/3) Checking commits...
remote: Processing will continue even if connection is closed.
remote: Perforce: 100% (3/3) Copying changelists...
remote: Perforce: Submitting new Git commit objects to Perforce: 4
To https://10.0.1.254/Jam
   6afeb15..89cba2b  master -> master
```

Git からは、うまくいったように見えているようです。 `p4v` のリビジョングラフ機能を使用して、 `README` ファイルの歴史を Perforce の視点から見てみましょう。

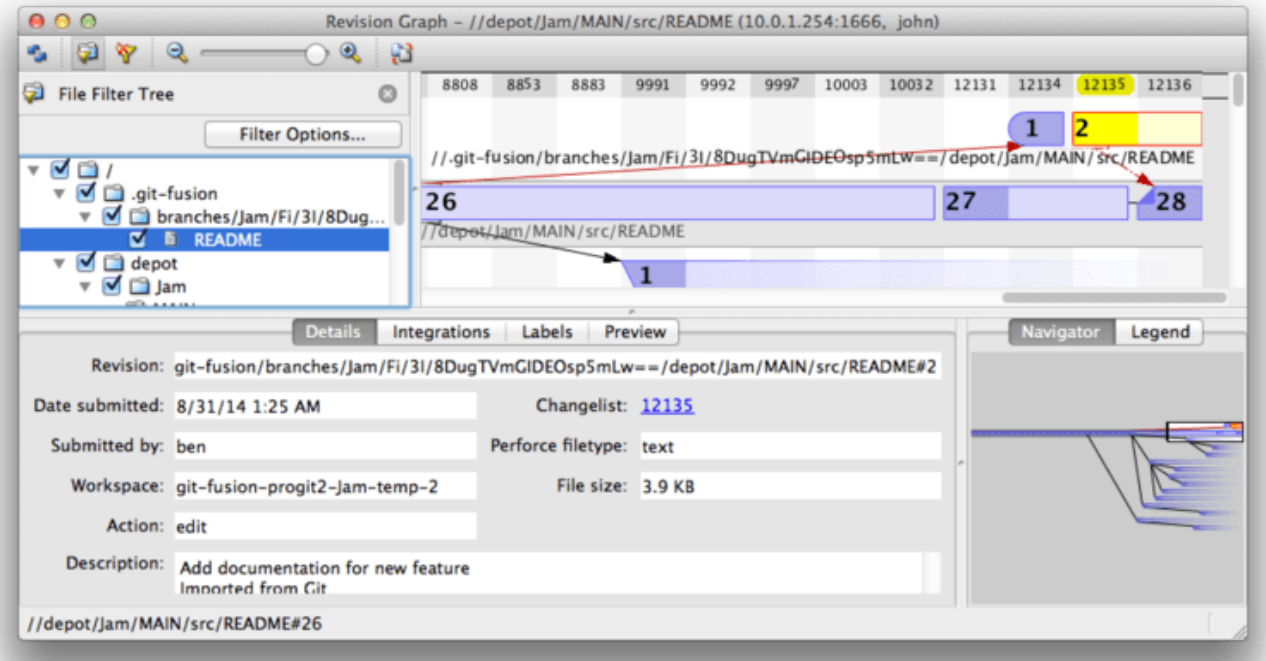



*Figure 146. Git でのプッシュの結果作成される Perforce のリビジョングラフ*

この画面を見たことがないと、混乱するかもしれませんが、 Git の歴史をグラフィカルに表示するビューアと同じ概念を示しています。 ここでは `README` ファイルの歴史を見ています。そのため、左上のディレクトリツリーでは、様々なブランチなかからそのファイルだけを取り上げて表示しています。 右上には、そのファイルの様々なリビジョンがどのように関連しているか、ビジュアライズされたグラフが表示されます。このグラフの全体像は右下にあります。 ビューの残りの部分では、選択したリビジョン（この場合は 2 ）の詳細を表示しています。

ひとつ注目してもらいたいのは、このグラフが Git の歴史のグラフとそっくりだということです。 Perforce にはコミット 1 および 2 を格納する名前つきのブランチがありません。代わりに “anonymous” ブランチを `.git-fusion` ディレクトリに作成し、そこに保管しています。 同様のことは、名前つきの Git のブランチに、対応する名前つきの Perforce のブランチがない場合にも起こります（設定ファイルを使えば、後でそのようなブランチを Perforce のブランチへ対応づけることも可能です）。

これのほとんどは舞台裏で行われますが、最終的には、ひとつのチームの中で、ある人は Git を使用し、またある人は Perforce を使用するということができ、その双方とも他の人が何を使用しているのかを知ることはない、という結果になりました。

#### Git-Fusion のまとめ

Perforce サーバへのアクセス権がある（または取得できる）なら、 Git Fusion は Git と Perforce が互いにやりとりできるようにする素晴らしい方法です。 多少の設定は必要ですが、学習曲線は急ではありません。 本節は、この章において、 Git の全機能を使用することに関して注意事項のない、数少ない節の一つです。 Perforce は指定した処理すべてを喜んでこなす、とは言いません – すでにプッシュ済みの歴史を書き換えようとしたら、 Git Fusion はそれをリジェクトします – ですが、 Git Fusion は Git そのままの感じになるよう非常に苦心しています。 また、Git のサブモジュールを使用したり（Perforce のユーザには変な風にみえますが）、ブランチのマージをしたり（Perforce 側では統合として記録されます）することも可能です。

Git Fusion をセットアップするようサーバの管理者を説得できなかったとしても、Git と Perforce を一緒に使用する方法は他にもあります。

### Git-p4

git-p4 は、 Git と Perforce の間の双方向ブリッジです。 git-p4 は Git リポジトリの中だけで動作するので、 Perforce サーバにおけるいかなるアクセス権も必要としません（もちろん、ユーザ権限は必要ですが）。 git-p4 には Git Fusion ほどの柔軟性や完全性はありません。ですが、やりたいであろうことの大半を、サーバ環境に対して侵襲的になることなく実施できます。

> **NOTE** git-p4 で作業を行う場合、 p4 ツールに PATH が通っている必要があります。 これを書いている時点では、 p4 は http://www.perforce.com/downloads/Perforce/20-User から無料で入手できます。

#### セットアップ

例のため、前に見てきたとおり Perforce サーバを Git Fusion OVA で実行しますが、ここでは Git Fusion サーバをバイパスして、 Perforce のバージョン管理機能を直接使用します。

p4 コマンドラインクライアント（ git-p4 がこれに依存している）を使用するには、環境変数を2つ設定する必要があります。

```
$ export P4PORT=10.0.1.254:1666
$ export P4USER=john
```

#### 使いはじめる

Git でやるのと同様、最初にすることはクローンです。

```
$ git p4 clone //depot/www/live www-shallow
Importing from //depot/www/live into www-shallow
Initialized empty Git repository in /private/tmp/www-shallow/.git/
Doing initial import of //depot/www/live/ from revision #head into
refs/remotes/p4/master
```

これで、 Git の用語で言う “シャロー” クローンが作成されます（Perforce の最新のリビジョンだけが Git へインポートされます）。覚えておいて欲しいのですが、 Perforce はすべてのリビジョンをすべてのユーザに渡すようデザインされてはいません。 Git を Perforce のクライアントとして使用するにはこれで十分ですが、それ以外の用途には不十分といえます。

クローンが完了したら、十分な機能を備えた Git リポジトリの出来上がりです。

```
$ cd myproject
$ git log --oneline --all --graph --decorate
* 70eaf78 (HEAD, p4/master, p4/HEAD, master) Initial import of
//depot/www/live/ from the state at revision #head
```

Perforceを表す “p4” リモートがあることに注意が必要ですが、それ以外はすべて、通常のクローンと同じように見えます。 実際、これは少し誤解をまねきやすいのですが、実際にはリモートがあるわけではありません。

```
$ git remote -v
```

このリポジトリにはリモートはひとつもありません。 git-p4 は、サーバの状態を表すために参照をいくつか作成します。これが `git log` からはリモート参照のように見えます。ですが、これらの参照は Git 自身が管理しているものではなく、またそこへプッシュすることもできません。

**ワークフロー**

オーケー、それでは作業を始めましょう。 ここでは、ある非常に重要な機能に関して進捗があり、その成果をチームの他のメンバーに見せられる状態になっているとしましょう。

```
$ git log --oneline --all --graph --decorate
* 018467c (HEAD, master) Change page title
* c0fb617 Update link
* 70eaf78 (p4/master, p4/HEAD) Initial import of //depot/www/live/ from
the state at revision #head
```

すでに2つのコミットを作成しており、Perforce サーバへ送信する準備もできています。 今日、他の誰かが作業をしていなかったか確認してみましょう。

```
$ git p4 sync
git p4 sync
Performing incremental import into refs/remotes/p4/master git branch
Depot paths: //depot/www/live/
Import destination: refs/remotes/p4/master
Importing revision 12142 (100%)
$ git log --oneline --all --graph --decorate
* 75cd059 (p4/master, p4/HEAD) Update copyright
| * 018467c (HEAD, master) Change page title
| * c0fb617 Update link
|/
* 70eaf78 Initial import of //depot/www/live/ from the state at revision
#head
```

誰かが作業をしていたようです。また、 `master` と `p4/master` が分岐しています。 Perforce のブランチのシステムは Git とは *まったく* 異なり、マージコミットを送信しても意味をなしません。 git-p4 では、コミットをリベースすることを推奨しており、そのためのショートカットも用意されています。

```
$ git p4 rebase
Performing incremental import into refs/remotes/p4/master git branch
Depot paths: //depot/www/live/
No changes to import!
Rebasing the current branch onto remotes/p4/master
First, rewinding head to replay your work on top of it...
Applying: Update link
Applying: Change page title
 index.html | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```

出力から分かったかと思いますが、 `git p4 rebase` は `git p4 sync` の後に `git rebase p4/master` を実行するショートカットです。 実際にはもう少し賢いのですが（特に複数のブランチで作業をしている場合には）、これはよい近似と言えます。

これで歴史がリニアになり、変更を Perforce へ提供できる状態になりました。 `git p4 submit` を実行すると、 Git の `p4/master` から `master` の間の各コミットに対して Perforce のリビジョンを作成しようとします。 実行するとお好みのエディタが開かれます。ファイルの内容は次のような感じです。

```
# A Perforce Change Specification.
#
#  Change:      The change number. 'new' on a new changelist.
#  Date:        The date this specification was last modified.
#  Client:      The client on which the changelist was created.  Read-
only.
#  User:        The user who created the changelist.
#  Status:      Either 'pending' or 'submitted'. Read-only.
#  Type:        Either 'public' or 'restricted'. Default is 'public'.
#  Description: Comments about the changelist.  Required.
#  Jobs:        What opened jobs are to be closed by this changelist.
#               You may delete jobs from this list.  (New changelists
only.)
#  Files:       What opened files from the default changelist are to be
added
#               to this changelist.  You may delete files from this list.
#               (New changelists only.)

Change:  new

Client:  john_bens-mbp_8487

User: john

Status:  new

Description:
   Update link

Files:
   //depot/www/live/index.html   # edit


######## git author ben@straub.cc does not match your p4 account.
######## Use option --preserve-user to modify authorship.
######## Variable git-p4.skipUserNameCheck hides this message.
######## everything below this line is just the diff #######
--- //depot/www/live/index.html  2014-08-31 18:26:05.000000000 0000
+++ /Users/ben/john_bens-mbp_8487/john_bens-
mbp_8487/depot/www/live/index.html   2014-08-31 18:26:05.000000000 0000
@@ -60,7 +60,7 @@
 </td>
 <td valign=top>
 Source and documentation for
-<a href="http://www.perforce.com/jam/jam.html">
+<a href="jam.html">
 Jam/MR</a>,
 a software build tool.
 </td>
```

これは、 git-p4 が気を利かせて末尾に追加している内容を除けば、 `p4 submit` を実行した場合とほぼ同じ内容です。 git-p4 は、コミットやチェンジセットに対して氏名を指定する必要がある場合、 Git と Perforce で設定をそれぞれ個別に行えるようにしていますが、その設定を上書きしたい場合もあります。 例えば、 Git のコミットをインポートしていて、そのコミットの作成者が Perforce のユーザアカウントを持っていない場合を考えます。この場合、最終的にできるチェンジセットは（あなたではなく）そのコミットの作成者が書いたように見えるようにしたいだろうと思います。

git-p4 は気が利いていて、Git のコミットからメッセージをインポートして、この Perforce のチェンジセットの内容にセットしてくれます。そのため、ここではファイルの保存とエディタの終了を2回（コミット1つにつき1回）行うだけで済みます。 結果として、シェルへの出力は次のような感じになります。

```
$ git p4 submit
Perforce checkout for depot path //depot/www/live/ located at
/Users/ben/john_bens-mbp_8487/john_bens-mbp_8487/depot/www/live/
Synchronizing p4 checkout...
... - file(s) up-to-date.
Applying dbac45b Update link
//depot/www/live/index.html#4 - opened for edit
Change 12143 created with 1 open file(s).
Submitting change 12143.
Locking 1 files ...
edit //depot/www/live/index.html#5
Change 12143 submitted.
Applying 905ec6a Change page title
//depot/www/live/index.html#5 - opened for edit
Change 12144 created with 1 open file(s).
Submitting change 12144.
Locking 1 files ...
edit //depot/www/live/index.html#6
Change 12144 submitted.
All commits applied!
Performing incremental import into refs/remotes/p4/master git branch
Depot paths: //depot/www/live/
Import destination: refs/remotes/p4/master
Importing revision 12144 (100%)
Rebasing the current branch onto remotes/p4/master
First, rewinding head to replay your work on top of it...
$ git log --oneline --all --graph --decorate
* 775a46f (HEAD, p4/master, p4/HEAD, master) Change page title
* 05f1ade Update link
* 75cd059 Update copyright
* 70eaf78 Initial import of //depot/www/live/ from the state at revision
#head
```

出力は、実際に行われている処理にもっとも近いアナロジーとして、単に `git push` を実行したかのような内容になっています。

注意事項として、この処理では Git の各コミットが個別の Perforce のチェンジセットに変換されます。複数

のコミットを1つのチェンジセットへスカッシュしたい場合は、 `git p4 submit` の前に対話的リベースを行ってください。 また、チェンジセットとして送信された全コミットの SHA-1 ハッシュが変更される点にも注意してください。これは、 git-p4 が変換した各コミットの末尾に行を追加するためです。

```
$ git log -1
commit 775a46f630d8b46535fc9983cf3ebe6b9aa53145
Author: John Doe <john@example.com>
Date:   Sun Aug 31 10:31:44 2014 -0800

    Change page title

    [git-p4: depot-paths = "//depot/www/live/": change = 12144]
```

マージコミットを送信しようとした場合、何が起こるでしょうか？ やってみましょう。 現在の状況は次のようになっています。

```
$ git log --oneline --all --graph --decorate
* 3be6fd8 (HEAD, master) Correct email address
*   1dcbf21 Merge remote-tracking branch 'p4/master'
|\
| * c4689fc (p4/master, p4/HEAD) Grammar fix
* | cbacd0a Table borders: yes please
* | b4959b6 Trademark
|/
* 775a46f Change page title
* 05f1ade Update link
* 75cd059 Update copyright
* 70eaf78 Initial import of //depot/www/live/ from the state at revision
#head
```

Git と Perforce の歴史が 775a46f の後で分岐しています。 Git 側に2つのコミットがあり、次に Perforce 側の HEAD とのマージコミット、さらにその次にまた別のコミットがあります。 ここでは、これらすべてのコミットを Perforce 側のチェンジセットの一番上に追加してみます。 現時点で送信した場合に何が起こるか見てみましょう。

```
$ git p4 submit -n
Perforce checkout for depot path //depot/www/live/ located at
/Users/ben/john_bens-mbp_8487/john_bens-mbp_8487/depot/www/live/
Would synchronize p4 checkout in /Users/ben/john_bens-mbp_8487/john_bens-
mbp_8487/depot/www/live/
Would apply
  b4959b6 Trademark
  cbacd0a Table borders: yes please
  3be6fd8 Correct email address
```

`-n` フラグは `--dry-run` の省略形で、送信コマンドを本当に実行したら何が起こるかを表示してくれます。この場合、 Perforce サーバに存在しない3つの非マージコミットに対応するように、 Perforce のチェンジセットを3つ作成するように見えます。 これは、望む結果そのもののように見えますので、結果どうなったか見てみましょう。

```
$ git p4 submit
[…]
$ git log --oneline --all --graph --decorate
* dadbd89 (HEAD, p4/master, p4/HEAD, master) Correct email address
* 1b79a80 Table borders: yes please
* 0097235 Trademark
* c4689fc Grammar fix
* 775a46f Change page title
* 05f1ade Update link
* 75cd059 Update copyright
* 70eaf78 Initial import of //depot/www/live/ from the state at revision
#head
```

送信の前にリベースを実行したかのように（実際、それが行われているのですが）、歴史がリニアになりました。 これはつまり、Git 側ではブランチを作成したり、ブランチ上で作業したり、ブランチを放棄したり、ブランチをマージしたりといった作業を自由に行えること、その際に歴史が何らかの形で Perforce と非互換になってしまう心配もないということです。 リベースが行えるなら、その内容を Perforce サーバへ送信できます。

#### ブランチ

Perforce プロジェクトに複数のブランチがある場合でも、運の尽きというわけではありません。 git-p4 はそのようなプロジェクトを、 Git と同じように扱えます。 まず、 Perforce のディポが次のような内容になっているとしましょう。

```
//depot
  └── project
      ├── main
      └── dev
```

さらに、次のようなビュー・スペックを持った `dev` ブランチがあるとしましょう。

```
//depot/project/main/... //depot/project/dev/...
```

git-p4 はこのような状況を自動的に検出し、正しく処理を行います。

```
$ git p4 clone --detect-branches //depot/project@all
Importing from //depot/project@all into project
Initialized empty Git repository in /private/tmp/project/.git/
Importing revision 20 (50%)
    Importing new branch project/dev

    Resuming with change 20
Importing revision 22 (100%)
Updated branches: main dev
$ cd project; git log --oneline --all --graph --decorate
* eae77ae (HEAD, p4/master, p4/HEAD, master) main
| * 10d55fb (p4/project/dev) dev
| * a43cfae Populate //depot/project/main/... //depot/project/dev/....
|/
* 2b83451 Project init
```

ディポのパスに “@all” という指示子がついていることに注意してください。これは git-p4 に対して、パスの示すサブツリーの最新のチェンジセットだけでなく、そのパスにあったことのあるすべてのチェンジセットをクローンするように指示しています。 これは Git のクローンの考え方に近いですが、作業中のプロジェクトに長い歴史がある場合、ちょっと時間がかかるかもしれません。

`--detect-branches` フラグは、 git-p4 に対して、Perforce のブランチを Git の参照へマッピングする際に、 Perforce のブランチ仕様を使用するように指示しています。 そのようなマッピングが Perforce サーバになかった場合（これは Perforce を使う分にはまったく問題ないやり方ですが）でも、git-p4 に対してブランチのマッピングがどのようになっているかを指示でき、同じ結果が得られます。

```
$ git init project
Initialized empty Git repository in /tmp/project/.git/
$ cd project
$ git config git-p4.branchList main:dev
$ git clone --detect-branches //depot/project@all .
```

設定値 `git-p4.branchList` に `main:dev` を設定すると、 “main” と “dev” がいずれもブランチであること、2つめのブランチは1つめのブランチの子であることを git-p4 へ示します。

ここで `git checkout -b dev p4/project/dev` を実行してからコミットを作成した場合でも、 git-p4 は充分に賢いので、 `git p4 submit` を実行した際には正しいブランチを対象にしてくれます。 なお、残念なことに、 git-p4 ではシャロークローンと複数ブランチを混ぜて使うことができません。巨大なプロジェクトにおいて複数のブランチで作業したい場合、ブランチごとに `git p4 clone` を実行する必要があります。

また、ブランチの作成や統合には、 Perforce クライアントを使用する必要があります。 git-p4 にできるのは既存のブランチに対する同期と送信だけで、それも対象にできるのはリニアなチェンジセットを一度にひとつだけです。 Git で2つのブランチをマージして新しいチェンジセットを送信しても、一塊りのファイルの変更として記録されるだけです。マージの対象となったブランチはどれかといったメタデータは失われてしまいます。

### Git と Perforce のまとめ

git-p4 は、 Git のワークフローを Perforce サーバ上で使用できるようにします。また、それを非常にうまいやり方で可能にします。 ですが、大元を管理しているのはあくまで Perforce であり、 Git はローカルでの作業にのみ使用しているということは忘れないでください。 Git のコミットの共有については特に気をつけてください。他のメンバーが使用しているリモートがある場合、 Perforce サーバに送信していないコミットをプッシュしないよう気をつけてください。

ソース管理のクライアントに Perforce と Git を自由に混ぜて使いたい場合、さらにサーバの管理者を説得してインストールの許可を貰える場合、Git Fusion は Git を Perforce サーバ用の第一級のバージョン管理クライアントにしてくれます。

## Git と TFS

Git は Windows を利用する開発者の間でもよく使われるようになってきています。 Windows 上でコードを書いているのなら、 Microsoft の Team Foundation Server (TFS) を使用することもあるでしょう。 TFS はコラボレーションスイートで、不具合および作業成果物に対するトラッキング、 Scrum やその他の開発プロセスのサポート、コードレビュー、そしてバージョン管理といった機能が含まれています。 ここがちょっとややこしいところなのですが、 **TFS** 自体はサーバで、ソースコード管理には、 Git や TFS 専用のバージョン管理システム（ **TFVC** (Team Foundation Version Control) とも呼ばれる）をサポートしています。 TFS の Git サポートは幾分新しい機能（バージョン2013から搭載）なので、それ以前からあったツールはどれも、実際にはほぼ TFVC だけを使用している場合であっても、バージョン管理部分のことを “TFS” と呼んでいます。

所属しているチームは TFVC を使用しているけれど、あなた自身はバージョン管理のクライアントに Git を使用したいという場合には、そのためのプロジェクトがあります。

### どちらのツールを使うか

実際には、ツールは2つあります。 git-tf と git-tfs です。

git-tfs （ https://github.com/git-tfs/git-tfs から入手できます）は .NET プロジェクトで、（これを書いている時点では）Windows でのみ動作します。 git-tfs は、Git リポジトリに対する操作に libgit2 の .NET バインディングを使用しています。libgit2 はライブラリ指向の Git の実装で、処理性能が高く、また Git の内部に対して柔軟な操作が行えるようになっています。 libgit2 は Git の機能を網羅的に実装してはいないため、その差を埋めるために、 git-tfs の一部の操作では実際にはコマンドライン版の Git クライアントが呼び出されています。そのため、Git リポジトリに対する操作に関して、git-tfs の設計に起因した制約は特にありません。 git-tfs は、 TFS サーバの操作に Visual Studio のアセンブリを使用しているため、TFVC 向け機能は非常に成熟しています。 また、これは Visual Studio のアセンブリにアクセスできる必要があるということでもあります。そのため、比較的新しいバージョンの Visual Studio （Visual Studio 2010以降の任意のエディション。バージョン2012以降の Visual Studio Express でもよい）か、Visual Studio SDK のインストールが必要です。

git-tf （ホームページは https://gittf.codeplex.com ）は Java プロジェクトで、Java実行環境のあるあらゆるコンピュータで実行できます。 git-tf は Git リポジトリに対する操作に JGit （JVM 用の Git の実装）を使用しているため、 Git の機能という観点においては事実上制約はありません。 しかし、 TFVC に対するサポートは git-tfs と比較すると限定的です – 例えば、ブランチをサポートしていません。

どちらのツールにも長所と短所があり、また一方よりもう一方が向いている状況というのはいくらでもあります。 本書では、2つのツールの基本的な使用法について取り上げます。

| | |
|---|---|
| **NOTE** | 以降の手順に従って操作を行うには、 TFVC ベースのリポジトリへのアクセス権が必要です。 そのようなリポジトリは Git や Subversion のリポジトリほど世の中にありふれたものではないので、自前で作成する必要があるかもしれません。 この場合、 Codeplex (https://www.codeplex.com) や Visual Studio Online (http://www.visualstudio.com) を利用するのがよいでしょう。 |

**使いはじめる: `git-tf` の場合**

最初に行うことは、あらゆる Git プロジェクトと同様、クローンです。 `git-tf` は次のような感じです。

```
$ git tf clone https://tfs.codeplex.com:443/tfs/TFS13 $/myproject/Main
project_git
```

最初の引数は TFVC コレクションの URL で、2番目の引数は `$/プロジェクト/ブランチ` の形式になっており、3番目の引数はローカルに作成する Git リポジトリのパスです（3番目の引数はオプショナルです）。 git-tf は一度にひとつのブランチしか扱えません。別の TFVC ブランチへチェックインしたい場合は、対象のブランチから新しくクローンを作成する必要があります。

次のコマンドで、フル機能の Git リポジトリが作成できます。

```
$ cd project_git
$ git log --all --oneline --decorate
512e75a (HEAD, tag: TFS_C35190, origin_tfs/tfs, master) Checkin message
```

これは *シャロー* クローンと言われるもので、最新のチェンジセットだけがダウンロードされます。 TFVC は、各クライアントが歴史の完全なコピーを持つようには設計されていません。そのため git-tf は、幾分高速な、最新のバージョンだけを取得する方法をデフォルトとしています。

時間があるなら、プロジェクトの歴史全体をクローンしてみるといいでしょう。 `--deep` オプションを使用します。

```
$ git tf clone https://tfs.codeplex.com:443/tfs/TFS13 $/myproject/Main \
  project_git --deep
Username: domain\user
Password:
Connecting to TFS...
Cloning $/myproject into /tmp/project_git: 100%, done.
Cloned 4 changesets. Cloned last changeset 35190 as d44b17a
$ cd project_git
$ git log --all --oneline --decorate
d44b17a (HEAD, tag: TFS_C35190, origin_tfs/tfs, master) Goodbye
126aa7b (tag: TFS_C35189)
8f77431 (tag: TFS_C35178) FIRST
0745a25 (tag: TFS_C35177) Created team project folder $/tfvctest via the \
        Team Project Creation Wizard
```

TFS_C35189 のような名前のタグに注目してください。これは、Git のどのコミットが、 TFVC のどのチェンジセットに対応しているかを分かりやすくするための機能です。 タグで表現するというのは上手い方法です。簡単な log コマンドだけで、どのコミットが TFVC 中のどのスナップショットに対応しているか確認できます。 なお、このタグ付けは必須ではありません（実際、 `git config git-tf.tag false` で無効にできます） – git-tf では、コミットとチェンジセットとのマッピングは `.git/git-tf` に保存されています。

**使いはじめる: `git-tfs` の場合**

git-tfs のクローン処理は、 git-tf とは少し異なります。 見てみましょう。

```
PS> git tfs clone --with-branches \
    https://username.visualstudio.com/DefaultCollection \
    $/project/Trunk project_git
Initialized empty Git repository in C:/Users/ben/project_git/.git/
C15 = b75da1aba1ffb359d00e85c52acb261e4586b0c9
C16 = c403405f4989d73a2c3c119e79021cb2104ce44a
Tfs branches found:
- $/tfvc-test/featureA
The name of the local branch will be : featureA
C17 = d202b53f67bde32171d5078968c644e562f1c439
C18 = 44cd729d8df868a8be20438fdeeefb961958b674
```

`--with-branches` フラグに注意してください。 git-tfs では TFVC のブランチを Git のブランチへマッピングできます。ここで `--with-branches` フラグは、 git-tfs に対し、すべての TFVC ブランチについて、対応するブランチをローカルの Git に作成するよう指示しています。 TFS 上で一度でもブランチの作成やマージを行っている場合、このオプションを指定することを強くお勧めします。ただし、このオプションは TFS 2010 より古いバージョンのサーバでは動作しません – それ以前のリリースでは “ブランチ” はただのフォルダだったためです。 git-tfs は単なるフォルダからはそのような指示は行えません。

結果の Git リポジトリを見てみましょう。

```
PS> git log --oneline --graph --decorate --all
* 44cd729 (tfs/featureA, featureA) Goodbye
* d202b53 Branched from $/tfvc-test/Trunk
* c403405 (HEAD, tfs/default, master) Hello
* b75da1a New project
PS> git log -1
commit c403405f4989d73a2c3c119e79021cb2104ce44a
Author: Ben Straub <ben@straub.cc>
Date:   Fri Aug 1 03:41:59 2014 +0000

    Hello

    git-tfs-id:
[https://username.visualstudio.com/DefaultCollection]$/myproject/Trunk;C16
```

2つのローカルブランチ `master` と `featureA` があり、それぞれ最初のクローンの開始位置（TFVC の `Trunk`）と、子のブランチ（TFVC の `featureA`）を表しています。 また“リモート” `tfs` にも `default` と `featureA` の2つの参照があり、これは TFVC のブランチを表現しています。 git-tfs はクローン元のブランチを `tfs/default` へマッピングし、それ以外のブランチにはそれぞれのブランチ名を付与します。

もうひとつ注意すべき点として、コミットメッセージにある `git-tfs-id:` という行があります。 タグとは異なり、このマーカーは git-tfs が TFVC のチェンジセットを Git のコミットへ対応づけるのに使用しています。 これは、TFVC にプッシュする前と後とで Git のコミットの SHA-1 ハッシュが異なるということを暗黙的に意味しています。

#### git-tf[s] のワークフロー

**NOTE**

どちらのツールを使用するにせよ、問題を避けるため、次の2つの Git の設定値をセットする必要があります。

```
$ git config set --local core.ignorecase=true
$ git config set --local core.autocrlf=false
```

当たり前ですが、次はプロジェクトに対して作業を行いたいことと思います。 TFVC および TFS には、ワークフローをややこしくする機能がいくつかあります。

1. TFVC 上に現れないフィーチャーブランチがあると、ややこしさが増します。 これには、TFVC と Git とでブランチを表現する方法が **まったく** 異なることが関係しています。
2. TFVC では、ユーザがサーバからファイルを“チェックアウト”して、他の誰も編集できないようにロックを掛けられることを認識しておいてください。 ローカルリポジトリ上でファイルを編集する妨げには当然なりませんが、変更を TFVC サーバへプッシュする段になって邪魔になるかもしれません。
3. TFS には“ゲート”チェックインという概念があります。これは、チェックインが許可されるには、 TFS のビルドとテストのサイクルが正常に終了する必要がある、というものです。 これは TFVC の“シェルブ”機能を使用していますが、これについてはここでは深入りしません。 手作業でなら、git-tf でもこの

方式をまねることはできます。 git-tfs はこれを考慮した `checkintool` コマンドを提供しています。

話を簡潔にするため、ここで取り上げるのは、これらの問題を避けたり起こらないようにした、ハッピーな手順です。

**ワークフロー: `git-tf` の場合**

ここでは、いくつか作業を終えて、 `master` に Git のコミットを2つ作成し、作業の成果を TFVC サーバで共有する準備ができているものとします。 Git リポジトリはこんな内容です。

```
$ git log --oneline --graph --decorate --all
* 4178a82 (HEAD, master) update code
* 9df2ae3 update readme
* d44b17a (tag: TFS_C35190, origin_tfs/tfs) Goodbye
* 126aa7b (tag: TFS_C35189)
* 8f77431 (tag: TFS_C35178) FIRST
* 0745a25 (tag: TFS_C35177) Created team project folder $/tfvctest via the
\
          Team Project Creation Wizard
```

`4178a82` のコミットでスナップショットを取って、 TFVC サーバへプッシュしたいものとします。 大事なことから先にとりかかりましょう。最後にリポジトリへ接続した後に、チームの他のメンバーが何か作業をしていなかったか見てみます。

```
$ git tf fetch
Username: domain\user
Password:
Connecting to TFS...
Fetching $/myproject at latest changeset: 100%, done.
Downloaded changeset 35320 as commit 8ef06a8. Updated FETCH_HEAD.
$ git log --oneline --graph --decorate --all
* 8ef06a8 (tag: TFS_C35320, origin_tfs/tfs) just some text
| * 4178a82 (HEAD, master) update code
| * 9df2ae3 update readme
|/
* d44b17a (tag: TFS_C35190) Goodbye
* 126aa7b (tag: TFS_C35189)
* 8f77431 (tag: TFS_C35178) FIRST
* 0745a25 (tag: TFS_C35177) Created team project folder $/tfvctest via the
\
          Team Project Creation Wizard
```

やはり、他の誰かが作業をしているようです。歴史が分岐しています。 これは Git が得意とするところですが、進め方は2種類あります。

1. Git ユーザなら、マージコミットを行うのが自然に感じられるでしょう（ `git pull` が行うのがマージコ

ミットなので）。 git-tf では単に `git tf pull` とすればマージコミットが行えます。
ですが、ここで注意が必要なのは、 TFVC はこれを自然とは考えないということです。マージコミットをプッシュしたら、歴史は Git 側と TFVC 側とで異なる見た目になりだし、ややこしいことになります。 一方、すべての変更をひとつのチェンジセットとして送信しようとしているのであれば、おそらくそれがもっとも簡単な選択肢です。

2. リベースを行うと、歴史がリニアになります。つまり、 Git のコミットひとつひとつを TFVC のチェンジセットへ変換する選択肢がとれるということです。 これが、以降の選択肢をもっとも広く取れる方法なので、この方法をお勧めします。git-tf でも、`git tf pull --rebase` で簡単に行えるようになっています。

どの方法をとるかはあなた次第です。 この例では、リベースする方法をとったとします。

```
$ git rebase FETCH_HEAD
First, rewinding head to replay your work on top of it...
Applying: update readme
Applying: update code
$ git log --oneline --graph --decorate --all
* 5a0e25e (HEAD, master) update code
* 6eb3eb5 update readme
* 8ef06a8 (tag: TFS_C35320, origin_tfs/tfs) just some text
* d44b17a (tag: TFS_C35190) Goodbye
* 126aa7b (tag: TFS_C35189)
* 8f77431 (tag: TFS_C35178) FIRST
* 0745a25 (tag: TFS_C35177) Created team project folder $/tfvctest via the
\
          Team Project Creation Wizard
```

さて、これで TFVC サーバへチェックインする準備ができました。 git-tf では、直近のチェンジセット以降のすべての変更を単一のチェンジセットにまとめる（`--shallow`、こっちがデフォルト）か、Git のコミットそれぞれに対して新しくチェンジセットを作成する（`--deep`）かを選択できます。 この例では、単一のチェンジセットにまとめる方法をとったとします。

```
$ git tf checkin -m 'Updating readme and code'
Username: domain\user
Password:
Connecting to TFS...
Checking in to $/myproject: 100%, done.
Checked commit 5a0e25e in as changeset 35348
$ git log --oneline --graph --decorate --all
* 5a0e25e (HEAD, tag: TFS_C35348, origin_tfs/tfs, master) update code
* 6eb3eb5 update readme
* 8ef06a8 (tag: TFS_C35320) just some text
* d44b17a (tag: TFS_C35190) Goodbye
* 126aa7b (tag: TFS_C35189)
* 8f77431 (tag: TFS_C35178) FIRST
* 0745a25 (tag: TFS_C35177) Created team project folder $/tfvctest via the
\
          Team Project Creation Wizard
```

新しくタグ `TFS_C35348` ができています。これは、 TFVC がコミット `5a0e25e` とまったく同じスナップショットを格納していることを意味します。 ここが重要なのですが、Git の各コミットが、すべて TFVC 側と対応づいている必要はありません。例えば、コミット `6eb3eb5` は、 TFVC サーバには存在しません。

以上が主なワークフローです。 他にも、考慮すべき点として気をつけるべきものが2つほどあります。

- ブランチはできません。 git-tf にできるのは、 TFVC のブランチから、 Git のリポジトリを作ることだけで、それも一度にひとつずつしか作れません。
- 共同作業の際は、 TFVC と Git のいずれかだけを使用し、両方は使用しないでください。 ひとつの TFVC リポジトリから、 git-tf で複数のクローンを作成した場合、各コミットの SHA-1 ハッシュそれぞれ異なります。コミットが作成されます。これは終わることのない頭痛の種になります。
- チームのワークフローに Git との協調作業が含まれており、定期的に TFVC との同期を取る場合、 TFVC へ接続する Git リポジトリはひとつだけにしてください。

**ワークフロー: `git-tfs` の場合**

git-tfs を使用した場合と同じシナリオを見ていきましょう。 Git リポジトリには、 `master` ブランチに対して行った新しいコミットが入っています。

```
PS> git log --oneline --graph --all --decorate
* c3bd3ae (HEAD, master) update code
* d85e5a2 update readme
| * 44cd729 (tfs/featureA, featureA) Goodbye
| * d202b53 Branched from $/tfvc-test/Trunk
|/
* c403405 (tfs/default) Hello
* b75da1a New project
```

さて、我々が作業している間に、他の誰かが作業をしていなかったか見てみましょう。

```
PS> git tfs fetch
C19 = aea74a0313de0a391940c999e51c5c15c381d91d
PS> git log --all --oneline --graph --decorate
* aea74a0 (tfs/default) update documentation
| * c3bd3ae (HEAD, master) update code
| * d85e5a2 update readme
|/
| * 44cd729 (tfs/featureA, featureA) Goodbye
| * d202b53 Branched from $/tfvc-test/Trunk
|/
* c403405 Hello
* b75da1a New project
```

同僚が新しく TFVC のチェンジセットを追加しており、それが新しいコミット `aea74a0` として表示されている他に、リモートブランチ `tfs/default` が移動されていることがわかりました。

git-tf と同様に、この分岐した歴史を処理する基本的な方法は2つあります。

1. 歴史をリニアに保つためにリベースを行う。
2. 実際に起こったことを残しておくためマージを行う。

この例では、Git の各コミットが TFVC のチェンジセットになる “ディープ” なチェックインを行おうとしているので、リベースをします。

```
PS> git rebase tfs/default
First, rewinding head to replay your work on top of it...
Applying: update readme
Applying: update code
PS> git log --all --oneline --graph --decorate
* 10a75ac (HEAD, master) update code
* 5cec4ab update readme
* aea74a0 (tfs/default) update documentation
| * 44cd729 (tfs/featureA, featureA) Goodbye
| * d202b53 Branched from $/tfvc-test/Trunk
|/
* c403405 Hello
* b75da1a New project
```

これで、コードを TFVC サーバへチェックインして、 作業を完了する準備ができました。 ここでは `rcheckin` コマンドを使用し、HEAD から始めて最初の `tfs` リモートブランチが見つかるまでの Git の各コミットに対して、 TFVC のチェンジセットを作成します（`checkin` コマンドでは、 Git のコミットをスカッシュするのと同様に、チェンジセットをひとつだけ作成します）。

```
PS> git tfs rcheckin
Working with tfs remote: default
Fetching changes from TFS to minimize possibility of late conflict...
Starting checkin of 5cec4ab4 'update readme'
 add README.md
C20 = 71a5ddce274c19f8fdc322b4f165d93d89121017
Done with 5cec4ab4b213c354341f66c80cd650ab98dcf1ed, rebasing tail onto new
TFS-commit...
Rebase done successfully.
Starting checkin of b1bf0f99 'update code'
 edit
.git\tfs\default\workspace\ConsoleApplication1/ConsoleApplication1/Program
.cs
C21 = ff04e7c35dfbe6a8f94e782bf5e0031cee8d103b
Done with b1bf0f9977b2d48bad611ed4a03d3738df05ea5d, rebasing tail onto new
TFS-commit...
Rebase done successfully.
No more to rcheckin.
PS> git log --all --oneline --graph --decorate
* ff04e7c (HEAD, tfs/default, master) update code
* 71a5ddc update readme
* aea74a0 update documentation
| * 44cd729 (tfs/featureA, featureA) Goodbye
| * d202b53 Branched from $/tfvc-test/Trunk
|/
* c403405 Hello
* b75da1a New project
```

TFVC サーバへのチェックインが成功した後に行われている処理に注目してください。 git-tfs は直前の処理結果に対して、残りの作業結果をリベースしています。 これは、コミットメッセージの末尾に `git-tfs-id` フィールドを追記しており、SHA-1 ハッシュが変化するためです。 これは仕様通りの動作であり、何も心配することはありません。ですが、そのような変更がなされていることは（特に、 Git のコミットを他の人と共有している場合は）認識しておいてください。

TFS には、ワークアイテム、レビュー依頼、ゲートチェックインなど、バージョン管理システムと統合されている機能が数多くあります。 これらの機能をコマンドラインツールだけで使うのは大変ですが、幸いなことに、 git-tfs ではグラフィカルなチェックインツールを簡単に起動できるようになっています。

```
PS> git tfs checkintool
PS> git tfs ct
```

だいたいこんな感じで表示されます。

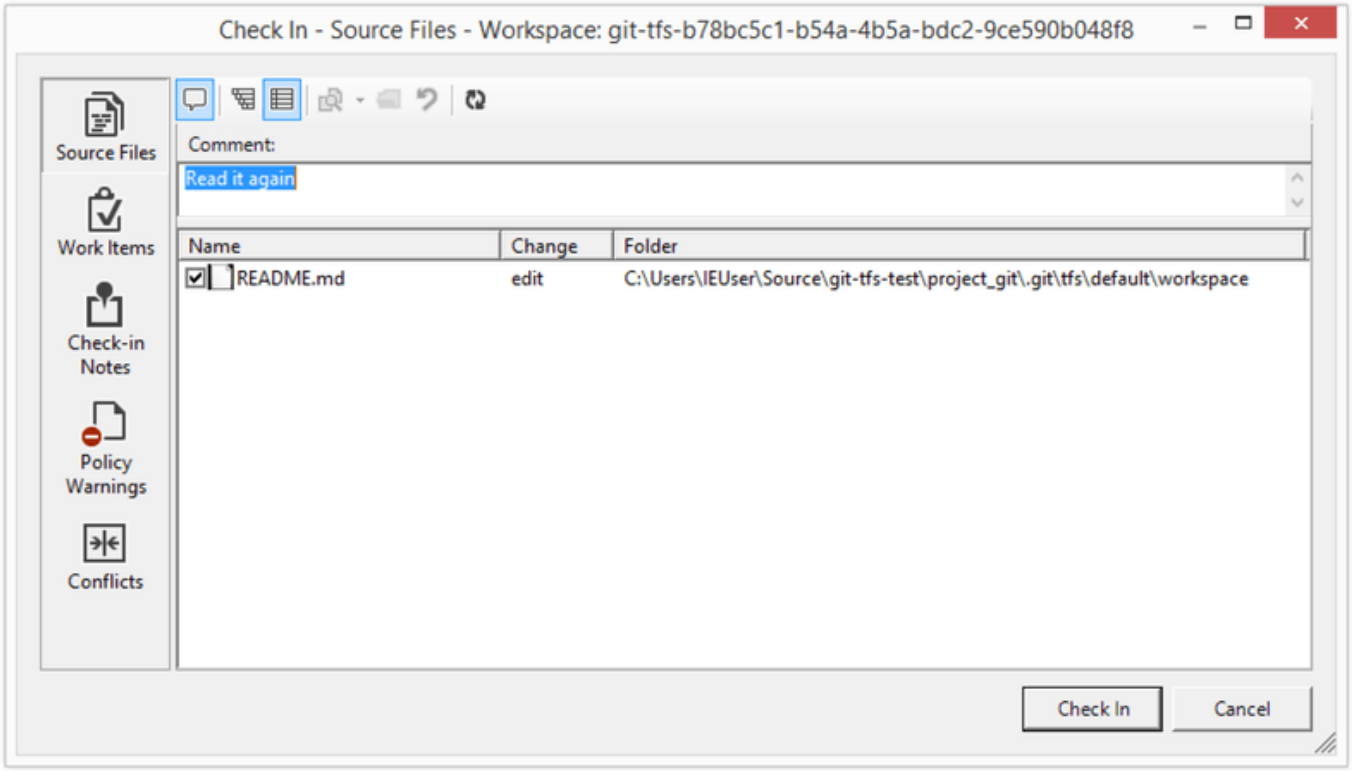



*Figure 147. git-tfs のチェックインツール*

TFS ユーザは見慣れていると思いますが、これは Visual Studio から表示されるものと同じダイアログです。

git-tfs では TFVC のブランチを Git のリポジトリから管理することもできます。 例として、ひとつ作成してみましょう。

```
PS> git tfs branch $/tfvc-test/featureBee
The name of the local branch will be : featureBee
C26 = 1d54865c397608c004a2cadce7296f5edc22a7e5
PS> git log --oneline --graph --decorate --all
* 1d54865 (tfs/featureBee) Creation branch $/myproject/featureBee
* ff04e7c (HEAD, tfs/default, master) update code
* 71a5ddc update readme
* aea74a0 update documentation
| * 44cd729 (tfs/featureA, featureA) Goodbye
| * d202b53 Branched from $/tfvc-test/Trunk
|/
* c403405 Hello
* b75da1a New project
```

TFVC でブランチを作成するということは、そのブランチが存在する場所にチェンジセットを追加するということなので、結果としてそれは Git のコミットへ反映されます。 git-tfs はリモートブランチ `tfs/featureBee` の **作成は** しましたが、 `HEAD` は相変わらず `master` を指していることに注意してください。 新しく作成したブランチ上で作業をしたい場合は、コミット `1d54865` から新しいコミットを作成することになりますが、この場合は恐らくそのコミットから新しくトピックブランチを作成することになるでしょう。

#### Git と TFS のまとめ

git-tf と git-tfs は、いずれも TFVC サーバに接続するための優れたツールです。 これらのツールにより、チーム全体を Git へ移行することなしに、ローカルで Git のパワーを享受でき、中央の TFVC サーバを定期的に巡回しなくて済み、開発者としての生活をより楽にすることができます。 Windows 上で作業をしているのなら（チームが TFS を使用しているなら多分そうだと思いますが）、機能がより網羅的な git-tfs を使用したいことと思います。また別のプラットフォーム上で作業をしているのなら、より機能の限られている git-tf を使用することになると思います。 この章で取り上げているほとんどのツールと同様、バージョン管理システムのうちひとつだけを正式なものとして、他は従属的な使い方にしておくべきです – Git か TFVC の両方ではなく、いずれか片方を共同作業の中心に置くべきです。

# Git へ移行する

Git 以外のバージョン管理システムで管理しているコードベースがあるけれど、Git を使いはじめることにした、という場合、どうにかしてプロジェクトを移行する必要があります。 この節では、主要なバージョン管理システム用のインポーターについて触れた後、独自のインポーターを自前で開発する方法を実際に見ていきます。 ここでは、いくつかの大きくてプロ仕様のソースコード管理システムからデータをインポートする方法を学びます。これは、移行するユーザの多くがそういったシステムのユーザであるのと、そういったシステムでは高品質なツールが簡単に手に入るためです。

## Subversion

先ほどの節で `git svn` の使い方を読んでいれば、話は簡単です。まず `git svn clone` でリポジトリを作り、そして Subversion サーバーを使うのをやめ、新しい Git サーバーにプッシュし、あとはそれを使い始めればいいのです。これまでの歴史が欲しいのなら、それも Subversion サーバーからプルすることができます (多少時間がかかります)。

しかし、インポートは完全ではありません。また時間もかかるので、正しくやるのがいいでしょう。まず最初に問題になるのが作者 (author) の情報です。Subversion ではコミットした人すべてがシステム上にユーザーを持っており、それがコミット情報として記録されます。たとえば先ほどの節のサンプルで言うと `schacon` がそれで、`blame` の出力や `git svn log` の出力に含まれています。これをうまく Git の作者データとしてマップするには、Subversion のユーザーと Git の作者のマッピングが必要です。`users.txt` という名前のファイルを作り、このような書式でマッピングを記述します。

```
schacon = Scott Chacon <schacon@geemail.com>
selse = Someo Nelse <selse@geemail.com>
```

SVN で使っている作者の一覧を取得するには、このようにします。

```
$ svn log --xml | grep author | sort -u | \
  perl -pe 's/.*>(.*?)<.*/$1 = /'
```

これは、まずログを XML フォーマットで生成します。その中から作者の情報を含む行だけを抽出し、重複を削除して、XML タグを除去します。（ちょっと見ればわかりますが、これは `grep` や `sort`、そして `perl` といったコマンドが使える環境でないと動きません） この出力を `users.txt` にリダイレクトし、そこに Git のユーザーデータを書き足していきます。

このファイルを `git svn` に渡せば、作者のデータをより正確にマッピングできるようになります。また、Subversion が通常インポートするメタデータを含めないよう `git svn` に指示することもできます。そのためには `--no-metadata` を `clone` コマンドあるいは `init` コマンドに渡します。そうすると、 `import` コマンドは次のようになります。

```
$ git svn clone http://my-project.googlecode.com/svn/ \
      --authors-file=users.txt --no-metadata -s my_project
```

これで、Subversion をちょっとマシにインポートした `my_project` ディレクトリができあがりました。コミットがこんなふうに記録されるのではなく、

```
commit 37efa680e8473b615de980fa935944215428a35a
Author: schacon <schacon@4c93b258-373f-11de-be05-5f7a86268029>
Date:   Sun May 3 00:12:22 2009 +0000

    fixed install - go to trunk

    git-svn-id: https://my-project.googlecode.com/svn/trunk@94 4c93b258-
373f-11de-
    be05-5f7a86268029
```

次のように記録されています。

```
commit 03a8785f44c8ea5cdb0e8834b7c8e6c469be2ff2
Author: Scott Chacon <schacon@geemail.com>
Date:   Sun May 3 00:12:22 2009 +0000

    fixed install - go to trunk
```

Author フィールドの見た目がずっとよくなっただけではなく、`git-svn-id` もなくなっています。

インポートした後には、ちょっとした後始末も行ったほうがよいでしょう。 たとえば、`git svn` が作成した変な参照は削除しておくべきです。 まずはタグを移動して、奇妙なリモートブランチではなくちゃんとしたタグとして扱えるようにします。そして、残りのブランチを移動してローカルで扱えるようにします。

タグを Git のタグとして扱うには、次のコマンドを実行します。

```
$ cp -Rf .git/refs/remotes/origin/tags/* .git/refs/tags/
$ rm -Rf .git/refs/remotes/origin/tags
```

これは、リモートブランチのうち `remotes/origin/tags/` で始まる名前のものを、実際の (軽量な) タグに変えます。

次に、`refs/remotes` 以下にあるそれ以外の参照をローカルブランチに移動します。

```
$ cp -Rf .git/refs/remotes/origin/* .git/refs/heads/
$ rm -Rf .git/refs/remotes/origin
```

このとき、Subversionではブランチが1つだったのにもかかわらず、名前が`@xxx`（xxxは数字）で終わる余分なブランチがいくつか出来てしまうことがあります。Subversionの「ペグ・リビジョン」という機能が原因なのですが、Gitにはこれと同等の機能は存在しません。よって、`git svn`コマンドはブランチ名にsvnのバージョン番号をそのまま追加します。svnでペグ・リビジョンをブランチに設定するときとまさに同じやり方です。もうペグ・リビジョンがいらないのであれば、`git branch -d`コマンドで削除してしまいましょう。

インポートが終わり、過去のブランチはGitのブランチへ、過去のタグはGitのタグへと変換できました。

最後に後始末についてです。残念なことに、`git svn`は`trunk`という名前の余計なブランチを生成してしまいます。Subversionにおけるデフォルトブランチではあるのですが、`trunk`の参照が指す場所は`master`と同じです。`master`のほうが用語としてもGitらしいですから、余分なブランチは削除してしまいましょう。

```
$ git branch -d trunk
```

これで、今まであった古いブランチはすべて Git のブランチとなり、古いタグもすべて Git のタグになりました。最後に残る作業は、新しい Git サーバーをリモートに追加してプッシュすることです。自分のサーバーをリモートとして追加するには以下のようにします。

```
$ git remote add origin git@my-git-server:myrepository.git
```

すべてのブランチやタグを一緒にプッシュするには、このようにします。

```
$ git push origin --all
$ git push origin --tags
```

これで、ブランチやタグも含めたすべてを、新しい Git サーバーにきれいにインポートできました。

## Mercurial

Mercurial と Git は、バージョンの表現方法がよく似ており、また Git の方が少し柔軟性が高いので、Mercurial から Git へのリポジトリの変換は非常に素直に行えます。変換には "hg-fast-export" というツールを使用します。このツールは次のコマンドで取得できます。

```
$ git clone http://repo.or.cz/r/fast-export.git /tmp/fast-export
```

変換の最初のステップとして、変換の対象となる Mercurial リポジトリのクローンを取得します。

```
$ hg clone <remote repo URL> /tmp/hg-repo
```

次のステップでは、author マッピングファイルを作成します。 チェンジセットの author フィールドへ指定できる内容は、Git より Mercurial の方が制限がゆるいので、これを機に内容を見直すのがよいでしょう。 author マッピングファイルは、bash シェルなら次のワンライナーで生成できます。

```
$ cd /tmp/hg-repo
$ hg log | grep user: | sort | uniq | sed 's/user: *//' > ../authors
```

プロジェクトの歴史の長さによりますが、このコマンドの実行には数秒かかります。実行後には、`/tmp/authors` ファイルが次のような内容で作成されているはずです。

```
bob
bob@localhost
bob <bob@company.com>
bob jones <bob <AT> company <DOT> com>
Bob Jones <bob@company.com>
Joe Smith <joe@company.com>
```

この例では、同じ人（Bob）がチェンジセットを作成したときの名前が4パターンあり、そのうち1つだけが標準に合った書き方で、また別の1つは Git のコミットとしては完全に無効なように見えます。 hg-fast-export では、このような状態を修正する場合、修正したい行の末尾に ={修正後の氏名とメールアドレス} を追加し、変更したくないユーザ名の行はすべて削除します。 すべてのユーザ名が正しいなら、このファイルは必要ありません。 この例では、ファイルの内容を次のようにします。

```
bob=Bob Jones <bob@company.com>
bob@localhost=Bob Jones <bob@company.com>
bob <bob@company.com>=Bob Jones <bob@company.com>
bob jones <bob <AT> company <DOT> com>=Bob Jones <bob@company.com>
```

次のステップでは、新しい Git リポジトリを作成して、エクスポート用スクリプトを実行します。

```
$ git init /tmp/converted
$ cd /tmp/converted
$ /tmp/fast-export/hg-fast-export.sh -r /tmp/hg-repo -A /tmp/authors
```

hg-fast-export に対して、 `-r` フラグで、変換の対象となる Mercurial リポジトリの場所を指定しています。 また、 `-A` フラグで、author マッピングファイルの場所を指定しています。 このスクリプトは、Mercurial のチェンジセットを解析して、Git の "fast-import" 機能（詳細はまた後で説明します）用のスクリプトへ変換します。 これには少し時間がかかります（ネットワーク経由の場合と比べれば *かなり* 速いですが）。出力は非常に長くなります。

```
$ /tmp/fast-export/hg-fast-export.sh -r /tmp/hg-repo -A /tmp/authors
Loaded 4 authors
master: Exporting full revision 1/22208 with 13/0/0 added/changed/removed
files
master: Exporting simple delta revision 2/22208 with 1/1/0
added/changed/removed files
master: Exporting simple delta revision 3/22208 with 0/1/0
added/changed/removed files
[…]
master: Exporting simple delta revision 22206/22208 with 0/4/0
added/changed/removed files
master: Exporting simple delta revision 22207/22208 with 0/2/0
added/changed/removed files
master: Exporting thorough delta revision 22208/22208 with 3/213/0
added/changed/removed files
Exporting tag [0.4c] at [hg r9] [git :10]
Exporting tag [0.4d] at [hg r16] [git :17]
[…]
Exporting tag [3.1-rc] at [hg r21926] [git :21927]
Exporting tag [3.1] at [hg r21973] [git :21974]
Issued 22315 commands
git-fast-import statistics:
---------------------------------------------------------------------
Alloc'd objects:     120000
Total objects:       115032 (    208171 duplicates                  )
      blobs  :        40504 (    205320 duplicates      26117 deltas of
39602 attempts)
      trees  :        52320 (      2851 duplicates      47467 deltas of
47599 attempts)
      commits:        22208 (         0 duplicates          0 deltas of
0 attempts)
      tags   :            0 (         0 duplicates          0 deltas of
0 attempts)
Total branches:         109 (         2 loads     )
      marks:        1048576 (     22208 unique    )
```

```
      atoms:           1952
Memory total:          7860 KiB
       pools:          2235 KiB
     objects:          5625 KiB
---------------------------------------------------------------------
pack_report: getpagesize()            =       4096
pack_report: core.packedGitWindowSize = 1073741824
pack_report: core.packedGitLimit      = 8589934592
pack_report: pack_used_ctr            =      90430
pack_report: pack_mmap_calls          =      46771
pack_report: pack_open_windows        =          1 /          1
pack_report: pack_mapped              =  340852700 /  340852700
---------------------------------------------------------------------

$ git shortlog -sn
   369  Bob Jones
   365  Joe Smith
```

作業はこれだけです。 すべての Mercurial のタグは Git のタグに変換され、 Mercurial のブランチとブックマークは Git のブランチに変換されています。 これで、リポジトリを新しいサーバ側へプッシュする準備が整いました。

```
$ git remote add origin git@my-git-server:myrepository.git
$ git push origin --all
```

## Perforce

次のインポート元としてとりあげるのは Perforce です。 前述の通り、 Git と Perforce を相互接続するには2つの方法があります。 git-p4 と Perforce Git Fusion です。

### Perforce Git Fusion

Git Fusion を使えば、移行のプロセスに労力はほぼかかりません。（ Git Fusion で述べた通り）設定ファイルで、プロジェクトの設定、ユーザのマッピング、およびブランチの設定を行った後、リポジトリをクローンすれば完了です。 Git Fusion がネイティブな Git リポジトリと類似の環境を提供してくれるので、お望みとあればいつでも、本物のネイティブな Git リポジトリへプッシュする準備はできているというわけです。 また、お望みなら、 Perforce を Git のホストとして使用することもできます。

### git-p4

git-p4 はインポート用ツールとしても使えます。 例として、 Perforce Public Depot から Jam プロジェクトをインポートしてみましょう。 クライアントをセットアップするには、環境変数 P4PORT をエクスポートして Perforce ディポの場所を指すようにしなければなりません。

```
$ export P4PORT=public.perforce.com:1666
```

**NOTE** 以降の手順に従うには、アクセスできる Perforce のディポが必要です。 この例では public.perforce.com にある公開ディポを使用していますが、アクセス権があればどんなディポでも使用できます。

`git p4 clone` コマンドを実行して、 Perforce サーバから Jam プロジェクトをインポートします。ディポとプロジェクトのパス、およびプロジェクトのインポート先のパスを指定します。

```
$ git-p4 clone //guest/perforce_software/jam@all p4import
Importing from //guest/perforce_software/jam@all into p4import
Initialized empty Git repository in /private/tmp/p4import/.git/
Import destination: refs/remotes/p4/master
Importing revision 9957 (100%)
```

このプロジェクトにはブランチがひとつしかありませんが、ブランチビューで設定されたブランチ（またはディレクトリ）があるなら、 `git p4 clone` に `--detect-branches` フラグを指定すれば、プロジェクトのブランチすべてをインポートできます。 この詳細については ブランチ を参照してください。

この時点で作業はおおむね完了です。 `p4import` ディレクトリへ移動して `git log` を実行すると、インポートした成果物を確認できます。

```
$ git log -2
commit e5da1c909e5db3036475419f6379f2c73710c4e6
Author: giles <giles@giles@perforce.com>
Date:   Wed Feb 8 03:13:27 2012 -0800

    Correction to line 355; change </UL> to </OL>.

    [git-p4: depot-paths = "//public/jam/src/": change = 8068]

commit aa21359a0a135dda85c50a7f7cf249e4f7b8fd98
Author: kwirth <kwirth@perforce.com>
Date:   Tue Jul 7 01:35:51 2009 -0800

    Fix spelling error on Jam doc page (cummulative -> cumulative).

    [git-p4: depot-paths = "//public/jam/src/": change = 7304]
```

`git-p4` が各コミットメッセージに識別子を追加しているのが分かると思います。 この識別子はそのままにしておいてもかまいません。後で万一 Perforce のチェンジ番号を参照しなければならなくなったときのために使えます。しかし、もし削除したいのなら、新しいリポジトリ上で何か作業を始める前の、この段階で消して

おきましょう。 `git filter-branch` を使えば、この識別子を一括削除することができます。

```
$ git filter-branch --msg-filter 'sed -e "/^\[git-p4:/d"'
Rewrite e5da1c909e5db3036475419f6379f2c73710c4e6 (125/125)
Ref 'refs/heads/master' was rewritten
```

`git log` を実行すると、コミットの SHA-1 チェックサムは変わりましたが、 `git-p4` という文字列がコミットメッセージから消えたことが分かると思います。

```
$ git log -2
commit b17341801ed838d97f7800a54a6f9b95750839b7
Author: giles <giles@giles@perforce.com>
Date:   Wed Feb 8 03:13:27 2012 -0800

    Correction to line 355; change </UL> to </OL>.

commit 3e68c2e26cd89cb983eb52c024ecdfba1d6b3fff
Author: kwirth <kwirth@perforce.com>
Date:   Tue Jul 7 01:35:51 2009 -0800

    Fix spelling error on Jam doc page (cummulative -> cumulative).
```

これで、インポート結果を新しい Git サーバへプッシュする準備ができました。

## TFS

あなたのチームで、ソース管理を TFVC から Git へ移行したいということになった場合、できる限り最高の忠実度で変換を行いたいことと思います。 そのため、相互運用についてのセクションでは git-tfs と git-tf の両方を取り上げましたが、本セクションでは git-tfs のみを取り上げます。これは git-tfs はブランチをサポートしている一方、 git-tf ではブランチの使用が禁止されており、対応が難しいためです。

> **NOTE** 以下で述べるのは、一方通行の変換です。 できあがった Git リポジトリを、元の TFVC プロジェクトと接続することはできません。

最初に行うのはユーザ名のマッピングです。 TFVC ではチェンジセットの author フィールドの内容をかなり自由に設定できますが、 Git では人間に読める形式の名前とメールアドレスが必要です。 この情報は、 `tf` コマンドラインクライアントで次のようにして取得できます。

```
PS> tf history $/myproject -recursive > AUTHORS_TMP
```

このコマンドは、プロジェクトの歴史からすべてのチェンジセットの情報を取ってきて、 AUTHORS_TMP ファイルへ出力します。このファイルは、 *User* カラム（2番目のカラム）のデータを抽出する際に使用します。 AUTHORS_TMP ファイルを開いて、2番目のカラムの開始位置と終了位置を確認したら、次のコマンド

ラインの、 cut コマンドの引数 11-20 を、それぞれ開始位置と終了位置で置き換えてください。

```
PS> cat AUTHORS_TMP | cut -b 11-20 | tail -n+3 | uniq | sort > AUTHORS
```

この cut コマンドは、各行の11文字目から20文字目だけを抽出します。 また、この tail コマンドは、最初の2行（フィールドヘッダと、下線のアスキーアート）を読み飛ばします。 この処理の結果は、重複を排除するためパイプで uniq コマンドへ送られた上で、 AUTHORS ファイルへ保存されます。 次のステップは手作業です。git-tfs でこのファイルを利用するには、各行は次のフォーマットに従っている必要があります。

```
DOMAIN\username = User Name <email@address.com>
```

イコール記号の左側にあるのは TFVC の “User” フィールドの内容、右側にあるのは Git のコミットで使用されるユーザ名です。

このファイルを作りおえたら、次は、対象となる TFVC プロジェクト全体のクローンを作成します。

```
PS> git tfs clone --with-branches --authors=AUTHORS
https://username.visualstudio.com/DefaultCollection $/project/Trunk
project_git
```

次は、コミットメッセージの末尾にある git-tfs-id セクションを消去したいことと思います。 これは、次のコマンドで行えます。

```
PS> git filter-branch -f --msg-filter 'sed "s/^git-tfs-id:.*$//g"' '--'
--all
```

これは、Git-bash から sed コマンドを使用して、 “git-tfs-id:” で始まる行を空文字列で置き換えます。 Git は空文字列を無視します。

これらをすべて実施したら、新しいリモートを追加したり、ブランチをプッシュしたり、チームが Git で作業を始めたりする準備はこれで完了です。

## A Custom Importer

前述した以外のシステムを使っている場合は、それ用のインポートツールをオンラインで探さなければなりません。CVS、Clear Case、Visual Source Safe、あるいはアーカイブのディレクトリなど、多くのバージョン管理システムについて、品質の高いインポーターが公開されています。 これらのツールがうまく動かなかったり、もっとマイナーなバージョン管理ツールを使っていたり、あるいはインポート処理で特殊な操作をしたりしたい場合は git fast-import を使います。 このコマンドはシンプルな指示を標準入力から受け取って、特定の Git データを書き出します。 git fast-import を使えば、生の Git コマンドを使ったり、生のオブジェクトを書きだそうとしたりする（詳細は Gitの内側 を参照してください）よりは、ずっと簡単に Git オブジェクトを作ることができます。 この方法を使えばインポートスクリプトを自作することができます。必

要な情報を元のシステムから読み込み、単純な指示を標準出力に出せばよいのです。
そして、このスクリプトの出力をパイプで `git fast-import` に送ります。

手軽に試してみるために、シンプルなインポーターを書いてみましょう。 `current` で作業をしており、プロジェクトのバックアップは時々ディレクトリまるごとのコピーで行っているものとします。バックアップディレクトリの名前は、タイムスタンプをもとに `back_YYYY_MM_DD` としています。これらを Git にインポートしてみましょう。 ディレクトリの構造は、このようになっています。

```
$ ls /opt/import_from
back_2014_01_02
back_2014_01_04
back_2014_01_14
back_2014_02_03
current
```

Git のディレクトリにインポートするため、まず Git がどのようにデータを格納しているかをおさらいしましょう。 覚えているかもしれませんが、 Git は基本的にはコミットオブジェクトのリンクリストであり、コミットオブジェクトがコンテンツのスナップショットを指しています。 `fast-import` に指示しなければならないのは、コンテンツのスナップショットが何でどのコミットデータがそれを指しているのかということと、コミットデータを取り込む順番だけです。 ここでは、スナップショットをひとつずつたどって各ディレクトリの中身を含むコミットオブジェクトを作り、それらを日付順にリンクさせるものとします。

Git ポリシーの実施例 と同様、ここでも Ruby を使って書きます。Ruby を使うのは、我々が普段使っている言語であり、読みやすくしやすいためです。 このサンプルをあなたの使いなれた言語で書き換えるのも簡単でしょう。単に適切な情報を標準出力に送るだけなのだから。 また、Windows を使っている場合は、行末にキャリッジリターンを含めないように注意が必要です。`git fast-import` が想定している行末は LF だけであり、Windows で使われている CRLF は想定していません。

まず最初に対象ディレクトリに移動し、そのサブディレクトリを認識させます。各サブディレクトリがコミットとしてインポートすべきスナップショットとなります。 続いて各サブディレクトリへ移動し、そのサブディレクトリをエクスポートするためのコマンドを出力します。 基本的なメインループは、このようになります。

```
last_mark = nil

# loop through the directories
Dir.chdir(ARGV[0]) do
  Dir.glob("*").each do |dir|
    next if File.file?(dir)

    # move into the target directory
    Dir.chdir(dir) do
      last_mark = print_export(dir, last_mark)
    end
  end
end
```

各ディレクトリ内で実行している `print_export` は、前のスナップショットの内容一覧とマークを受け取って、このディレクトリの内容一覧とマークを返します。このようにして、それぞれを適切にリンクさせます。“マーク”とは `fast-import` 用語で、コミットに対する識別子を意味します。コミットを作成するときにマークをつけ、それを使って他のコミットとリンクさせます。 つまり、`print_export` メソッドで最初にやることは、ディレクトリ名からマークを生成することです。

```
mark = convert_dir_to_mark(dir)
```

これを行うには、まずディレクトリの配列を作り、そのインデックスの値をマークとして使います。マークは整数値でなければならないからです。 メソッドの中身はこのようになります。

```
$marks = []
def convert_dir_to_mark(dir)
  if !$marks.include?(dir)
    $marks << dir
  end
  ($marks.index(dir) + 1).to_s
end
```

これで各コミットを表す整数値が取得できました。次に必要なのは、コミットのメタデータ用の日付です。日付はディレクトリ名に現れているので、ここから取得します。`print_export` ファイルで次にすることは、これです。

```
date = convert_dir_to_date(dir)
```

`convert_dir_to_date` の定義は次のようになります。

```
def convert_dir_to_date(dir)
  if dir == 'current'
    return Time.now().to_i
  else
    dir = dir.gsub('back_', '')
    (year, month, day) = dir.split('_')
    return Time.local(year, month, day).to_i
  end
end
```

これは、各ディレクトリの日付に対応する整数値を返します。 コミットのメタ情報として必要な最後の情報はコミッターのデータで、これはグローバル変数にハードコードします。

```
$author = 'John Doe <john@example.com>'
```

これで、コミットのデータをインポーターに流せるようになりました。
最初の情報では、今定義しているのがコミットオブジェクトであることと、どのブランチにいるのかを示しています。その後に先ほど生成したマークが続き、さらにコミッターの情報とコミットメッセージが続いた後にひとつ前のコミットが (もし存在すれば) 続きます。 コードはこのようになります。

```
# print the import information
puts 'commit refs/heads/master'
puts 'mark :' + mark
puts "committer #{$author} #{date} -0700"
export_data('imported from ' + dir)
puts 'from :' + last_mark if last_mark
```

タイムゾーン (-0700) をハードコードしているのは、そのほうがお手軽だったからです。 別のシステムからインポートする場合は、タイムゾーンをオフセットとして指定しなければなりません。 コミットメッセージは、次のような特殊な書式にする必要があります。

```
data (size)\n(contents)
```

まず最初に「data」という単語、そして読み込むデータのサイズ、改行、最後にデータがきます。 同じ書式は後でファイルのコンテンツを指定するときにも使うので、ヘルパーメソッド `export_data` を作ります。

```
def export_data(string)
  print "data #{string.size}\n#{string}"
end
```

残っているのは、各スナップショットが持つファイルのコンテンツを指定することです。 今回の場合はどれも一つのディレクトリにまとまっているので簡単です。`deleteall` コマンドを出力し、それに続けてディレクトリ内の各ファイルの中身を出力すればよいのです。 そうすれば、Git が各スナップショットを適切に記録します。

```
puts 'deleteall'
Dir.glob("**/*").each do |file|
  next if !File.file?(file)
  inline_data(file)
end
```

注意:多くのシステムではリビジョンを「あるコミットと別のコミットの差分」と考えているので、fast-importでもその形式でコマンドを受け取ることができます。つまりコミットを指定するときに、追加/削除/変更されたファイルと新しいコンテンツの中身で指定できるということです。 各スナップショットの差分を算出してそのデータだけを渡すこともできますが、処理が複雑になります。すべてのデータを渡して、Git に差分を算出させたほうがよいでしょう。 もし差分を渡すほうが手元のデータに適しているようなら、`fast-import` のマニュアルで詳細な方法を調べましょう。

新しいファイルの内容、あるいは変更されたファイルと変更後の内容を表す書式は次のようになります。

```
M 644 inline path/to/file
data (size)
(file contents)
```

この 644 はモード (実行可能ファイルがある場合は、そのファイルについては 755 を指定する必要があります) を表し、inline とはファイルの内容をこの次の行に続けて指定するという意味です。`inline_data` メソッドは、このようになります。

```
def inline_data(file, code = 'M', mode = '644')
  content = File.read(file)
  puts "#{code} #{mode} inline #{file}"
  export_data(content)
end
```

先ほど定義した `export_data` メソッドを再利用することができます。この書式はコミットメッセージの書式と同じだからです。

最後に必要となるのは、現在のマークを返して次の処理に渡せるようにすることです。

```
return mark
```

**NOTE**

Windows 上で動かす場合はさらにもう一手間必要です。 先述したように、Windows の改行文字は CRLF ですが `git fast-import` は LF にしか対応していません。この問題に対応して `git fast-import` をうまく動作させるには、CRLF ではなく LF を使うよう ruby に指示しなければなりません。

```
$stdout.binmode
```

これで終わりです。 スクリプト全体を以下に示します。

```
#!/usr/bin/env ruby

$stdout.binmode
$author = "John Doe <john@example.com>"

$marks = []
def convert_dir_to_mark(dir)
    if !$marks.include?(dir)
        $marks << dir
    end
```

```
    ($marks.index(dir)+1).to_s
end

def convert_dir_to_date(dir)
    if dir == 'current'
        return Time.now().to_i
    else
        dir = dir.gsub('back_', '')
        (year, month, day) = dir.split('_')
        return Time.local(year, month, day).to_i
    end
end

def export_data(string)
    print "data #{string.size}\n#{string}"
end

def inline_data(file, code='M', mode='644')
    content = File.read(file)
    puts "#{code} #{mode} inline #{file}"
    export_data(content)
end

def print_export(dir, last_mark)
    date = convert_dir_to_date(dir)
    mark = convert_dir_to_mark(dir)

    puts 'commit refs/heads/master'
    puts "mark :#{mark}"
    puts "committer #{$author} #{date} -0700"
    export_data("imported from #{dir}")
    puts "from :#{last_mark}" if last_mark

    puts 'deleteall'
    Dir.glob("**/*").each do |file|
        next if !File.file?(file)
        inline_data(file)
    end
    mark
end


# Loop through the directories
last_mark = nil
Dir.chdir(ARGV[0]) do
    Dir.glob("*").each do |dir|
        next if File.file?(dir)

        # move into the target directory
        Dir.chdir(dir) do
```

```
            last_mark = print_export(dir, last_mark)
        end
    end
end
```

このスクリプトを実行すれば、次のような結果が得られます。

```
$ ruby import.rb /opt/import_from
commit refs/heads/master
mark :1
committer John Doe <john@example.com> 1388649600 -0700
data 29
imported from back_2014_01_02deleteall
M 644 inline README.md
data 28
# Hello

This is my readme.
commit refs/heads/master
mark :2
committer John Doe <john@example.com> 1388822400 -0700
data 29
imported from back_2014_01_04from :1
deleteall
M 644 inline main.rb
data 34
#!/bin/env ruby

puts "Hey there"
M 644 inline README.md
(...)
```

インポーターを動かすには、インポート先の Git レポジトリにおいて、インポーターの出力をパイプで `git fast-import` に渡す必要があります。 インポート先に新しいディレクトリを作成したら、以下のように `git init` を実行し、そしてスクリプトを実行してみましょう。

```
$ git init
Initialized empty Git repository in /opt/import_to/.git/
$ ruby import.rb /opt/import_from | git fast-import
git-fast-import statistics:
---------------------------------------------------------------------
Alloc'd objects:       5000
Total objects:           13 (         6 duplicates                  )
      blobs  :            5 (         4 duplicates          3 deltas of
5 attempts)
      trees  :            4 (         1 duplicates          0 deltas of
4 attempts)
      commits:            4 (         1 duplicates          0 deltas of
0 attempts)
      tags   :            0 (         0 duplicates          0 deltas of
0 attempts)
Total branches:           1 (         1 loads     )
      marks:           1024 (         5 unique    )
      atoms:              2
Memory total:          2344 KiB
       pools:          2110 KiB
     objects:           234 KiB
---------------------------------------------------------------------
pack_report: getpagesize()            =       4096
pack_report: core.packedGitWindowSize = 1073741824
pack_report: core.packedGitLimit      = 8589934592
pack_report: pack_used_ctr            =         10
pack_report: pack_mmap_calls          =          5
pack_report: pack_open_windows        =          2 /          2
pack_report: pack_mapped              =       1457 /       1457
---------------------------------------------------------------------
```

ご覧のとおり、処理が正常に完了すると、処理内容に関する統計情報が表示されます。 この場合は、全部で 13 のオブジェクトからなる 4 つのコミットが 1 つのブランチにインポートされたことがわかります。 では、`git log` で新しい歴史を確認しましょう。

```
$ git log -2
commit 3caa046d4aac682a55867132ccdfbe0d3fdee498
Author: John Doe <john@example.com>
Date:   Tue Jul 29 19:39:04 2014 -0700

    imported from current

commit 4afc2b945d0d3c8cd00556fbe2e8224569dc9def
Author: John Doe <john@example.com>
Date:   Mon Feb 3 01:00:00 2014 -0700

    imported from back_2014_02_03
```

きれいな Git リポジトリができていますね。 ここで重要なのは、この時点ではまだ何もチェックアウトされていないということです。作業ディレクトリには何もファイルがありません。 ファイルを取得するには、ブランチをリセットして `master` の現在の状態にしなければなりません。

```
$ ls
$ git reset --hard master
HEAD is now at 3caa046 imported from current
$ ls
README.md main.rb
```

`fast-import` ツールにはさらに多くの機能があります。さまざまなモードを処理したりバイナリデータを扱ったり、複数のブランチやそのマージ、タグ、進捗状況表示などです。 より複雑なシナリオのサンプルは Git のソースコードの `contrib/fast-import` ディレクトリにあります。

## まとめ

これで、Git を他のバージョン管理システムのクライアントとして使ったり、既存のリポジトリのほぼすべてを、データを失うことなく Git リポジトリにインポートしたりといった作業を、安心して行えるようになったと思います。 次章では、Git の内部に踏み込みます。必要とあらばバイト単位での操作もできることでしょう。

# Gitの内側

どこかの章からここに飛んできたのか、本書の他の部分を読み終えてからここにたどり着いたのか – いずれにせよ、この章ではGitの内部動作と実装を詳細に見ていきます。 我々は、Gitがいかに便利で強力かを理解するには、その内部動作と実装を学ぶことが根本的に重要であると考えました。一方で、そういった内容は、初心者にとっては不必要に複雑で、かえって混乱を招くおそれがあると主張する人もいました。 そこで我々は、この話題を本書の最後の章にして、読者の学習プロセスの最初の方で読んでも最後の方で読んでもいいようにしました。 いつ読むかって？　それは読者の判断にお任せします。

もう既にあなたはこの章を読んでいますので、早速、開始しましょう。 まず最初に明確にしておきたいのですが、Gitの実態は内容アドレスファイルシステム（content-addressable filesystem）であり、その上にVCSのユーザーインターフェイスが実装されています。 これが何を意味するかについては、すぐ後で学びます。

初期（主に1.5より古いバージョン）のGitのユーザーインターフェイスは、今よりずっと複雑でした。これは、当時のGitが、洗練されたVCSであることよりも、このファイルシステムの方を重要視していたためです。 ここ数年で、Gitのユーザーインターフェイスは改良されて、世の中にある他のシステムと同じくらいシンプルで扱いやすいものになりました。しかし、複雑で学習するのが難しいという、初期のGitのUIに対するステレオタイプはいまだに残っています。

内容アドレスファイルシステムの層は驚くほど素晴らしいので、この章の最初で取り上げます。その次に転送メカニズムと、今後あなたが行う必要があるかもしれないリポジトリの保守作業について学習することにします。

## 配管（Plumbing）と磁器（Porcelain）

本書では、`checkout` や `branch`、`remote` などの約30のコマンドを用いて、Gitの使い方を説明しています。 一方、Gitには低レベルの処理を行うためのコマンドも沢山あります。これは、Gitが元々は、完全にユーザフレンドリーなVCSというよりも、VCSのためのツールキットだったことが理由です。これらのコマンドは、UNIX流につなぎ合わせたり、スクリプトから呼んだりすることを主眼において設計されています。 これらのコマンドは、通常 “配管（plumbing）” コマンドと呼ばれています。対して、よりユーザフレンドリーなコマンドは “磁器（porcelain）” コマンドと呼ばれています。

本書のはじめの9つの章では、ほぼ磁器コマンドだけを取り扱ってきました。 一方、本章ではそのほとんどで低レベルの配管コマンドを使用します。これは、Gitの内部動作にアクセスして、Gitが、ある処理を、なぜ、どうやって行うのか確かめるためです。 これら配管コマンドの多くは、コマンドラインから直接実行されるのではなく、新しくツールやカスタムスクリプトを作る際に構成要素となることを意図して作られています。

新規の、または既存のディレクトリで `git init` を実行すると、Git は `.git` というディレクトリを作ります。Git が保管したり操作したりする対象の、ほとんどすべてがここに格納されます。 リポジトリのバックアップやクローンをしたい場合、このディレクトリをどこかへコピーするだけで、ほぼ事足ります。 基本的にこの章では、全体を通して、 `.git` ディレクトリの内容を取り扱います。 `.git` ディレクトリの中は以下のようになっています。

```
$ ls -F1
HEAD
config*
description
hooks/
info/
objects/
refs/
```

これが `git init` を実行した直後のデフォルトのリポジトリであり、デフォルトで表示される内容です。なお、環境によってはここにないファイルが表示されることもあります。 `description` ファイルは、GitWebプログラムでのみ使用するものなので、特に気にしなくて大丈夫です。 `config` ファイルには、プロジェクト固有の設定が書かれています。また、`info` ディレクトリにはグローバルレベルの除外設定ファイル が格納されます。これは、.gitignoreファイルに記録したくない除外パターンを書く際に使用します。 `hooks` ディレクトリには、クライアントサイド、または、サーバーサイドのフックスクリプトが格納されます。フックスクリプトについては Git フック で詳しく説明します。

そして、重要なのは残りの4項目です。具体的には、 `HEAD` ファイル、 `index` ファイル（まだ作成されていない）、 `objects` ディレクトリ、 `refs` ディレクトリです。 これらがGitの中核部分になります。 `objects` ディレクトリにはデータベースのすべてのコンテンツが保管されます。`refs` ディレクトリには、それらコンテンツ内のコミットオブジェクトを指すポインタ（ブランチ）が保管されます。`HEAD` ファイルは、現在チェックアウトしているブランチを指します。`index` ファイルには、Git がステージングエリアの情報を保管します。 以降の各セクションでは、Git がどのような仕組みで動くのかを詳細に見ていきます。

## Gitオブジェクト

Git は内容アドレスファイルシステムです。 素晴らしい。 …で、それはどういう意味なのでしょう? それは、Gitのコアの部分はシンプルなキー・バリュー型データストアである、という意味です。 ここにはどんな種類のコンテンツでも格納でき、それに対応するキーが返されます。キーを使えば格納したコンテンツをいつでも取り出せます。 これは `hash-object` という配管コマンドを使えば実際に確認できます。このコマンドはデータを受け取り、それを `.git` ディレクトリに格納し、そのデータを格納しているキーを返します。 まずは、新しいGitリポジトリを初期化し、 `objects` ディレクトリ配下に何もないことを確認してみましょう。

```
$ git init test
Initialized empty Git repository in /tmp/test/.git/
$ cd test
$ find .git/objects
.git/objects
.git/objects/info
.git/objects/pack
$ find .git/objects -type f
```

Gitは `objects` ディレクトリを初期化して、その中に `pack` と `info` というサブディレクトリを作ります。しかし、ファイルはひとつも作られません。 今からGitデータベースにテキストを幾つか格納してみます。

```
$ echo 'test content' | git hash-object -w --stdin
d670460b4b4aece5915caf5c68d12f560a9fe3e4
```

`-w` オプションは、`hash-object` にオブジェクトを格納するよう指示しています。`-w` オプションを付けない場合、コマンドはただオブジェクトのキーとなる文字列を返します。`--stdin` オプションは、標準入力からコンテンツを読み込むよう指示しています。これを指定しない場合、`hash-object` はコマンドラインオプションの最後にファイルパスが指定されることを期待して動作します。 コマンドを実行すると、40文字から成るチェックサムのハッシュ値が出力されます。 これは、SHA-1ハッシュです。すぐ後で説明しますが、これは格納するコンテンツにヘッダーを加えたデータに対するチェックサムです。 これで、Gitがデータをどのようにして格納するか見ることができるようになりました。

```
$ find .git/objects -type f
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
```

`objects` ディレクトリの中にファイルがひとつあります。 Gitはまずこのようにしてコンテンツを格納します。コンテンツ1つごとに1ファイルで、ファイル名はコンテンツとそのヘッダーに対するSHA-1チェックサムで決まります。 SHA-1ハッシュのはじめの2文字がサブディレクトリの名前になり、残りの38文字がファイル名になります。

`cat-file` コマンドを使うと、コンテンツをGitから取り出すことができます。 このコマンドは、Gitオブジェクトを調べるための万能ナイフのようなものです。 `-p` オプションを付けると、`cat-file` コマンドはコンテンツのタイプを判別し、わかりやすく表示してくれます。

```
$ git cat-file -p d670460b4b4aece5915caf5c68d12f560a9fe3e4
test content
```

これで、Gitにコンテンツを追加したり、取り出したりできるようになりました。 ファイルの内容に対しても、これと同様のことを行えます。 例えば、あるファイルに対して簡単なバージョン管理を行うことができます。 まず、新規にファイルを作成し、データベースにその内容を保存します。

```
$ echo 'version 1' > test.txt
$ git hash-object -w test.txt
83baae61804e65cc73a7201a7252750c76066a30
```

それから、新しい内容をそのファイルに書き込んで、再び保存します。

```
$ echo 'version 2' > test.txt
$ git hash-object -w test.txt
1f7a7a472abf3dd9643fd615f6da379c4acb3e3a
```

データベースには、最初に格納したコンテンツに加えて、上記のファイルのバージョン2つが新規に追加され

ています。

```
$ find .git/objects -type f
.git/objects/1f/7a7a472abf3dd9643fd615f6da379c4acb3e3a
.git/objects/83/baae61804e65cc73a7201a7252750c76066a30
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
```

これで、上記のファイルの変更を取り消して最初のバージョンに戻せるようになりました。

```
$ git cat-file -p 83baae61804e65cc73a7201a7252750c76066a30 > test.txt
$ cat test.txt
version 1
```

また、2つ目のバージョンにもできます。

```
$ git cat-file -p 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a > test.txt
$ cat test.txt
version 2
```

しかし、それぞれのファイルのバージョンのSHA-1キーを覚えておくのは実用的ではありません。加えて、システムにはファイル名は格納されておらず、ファイルの内容のみが格納されています。 このオブジェクトタイプはブロブ（blob）と呼ばれます。 `cat-file -t` コマンドに SHA-1キーを渡すことで、あなたは Git 内にあるあらゆるオブジェクトのタイプを問い合わせることができます。

```
$ git cat-file -t 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a
blob
```

## ツリーオブジェクト

次のタイプはツリーです。これにより、ファイル名の格納の問題を解決して、さらに、複数のファイルをまとめて格納できるようになります。 Git がコンテンツを格納する方法は、UNIXのファイルシステムに似ていますが少し簡略化されています。 すべてのコンテンツはツリーオブジェクトまたはブロブオブジェクトとして格納されます。ツリーは UNIXのディレクトリエントリーと対応しており、ブロブはiノードやファイルコンテンツとほぼ対応しています。 1つのツリーオブジェクトには1つ以上のツリーエントリーが含まれています。このツリーエントリーには、ブロブか、サブツリーとそれに関連するモード、タイプ、ファイル名へのSHA-1ポインターが含まれています。 例えば、あるプロジェクトの最新のツリーはこのように見えるかもしれません。

```
$ git cat-file -p master^{tree}
100644 blob a906cb2a4a904a152e80877d4088654daad0c859      README
100644 blob 8f94139338f9404f26296befa88755fc2598c289      Rakefile
040000 tree 99f1a6d12cb4b6f19c8655fca46c3ecf317074e0      lib
```

`master^{tree}` のシンタックスは、`master` ブランチ上での最後のコミットが指しているツリーオブジェクトを示します。 `lib` サブディレクトリはブロブではなく、別のツリーへのポインタであることに注意してください。

```
$ git cat-file -p 99f1a6d12cb4b6f19c8655fca46c3ecf317074e0
100644 blob 47c6340d6459e05787f644c2447d2595f5d3a54b      simplegit.rb
```

概念的には、Gitが格納するデータは次のようなものです。

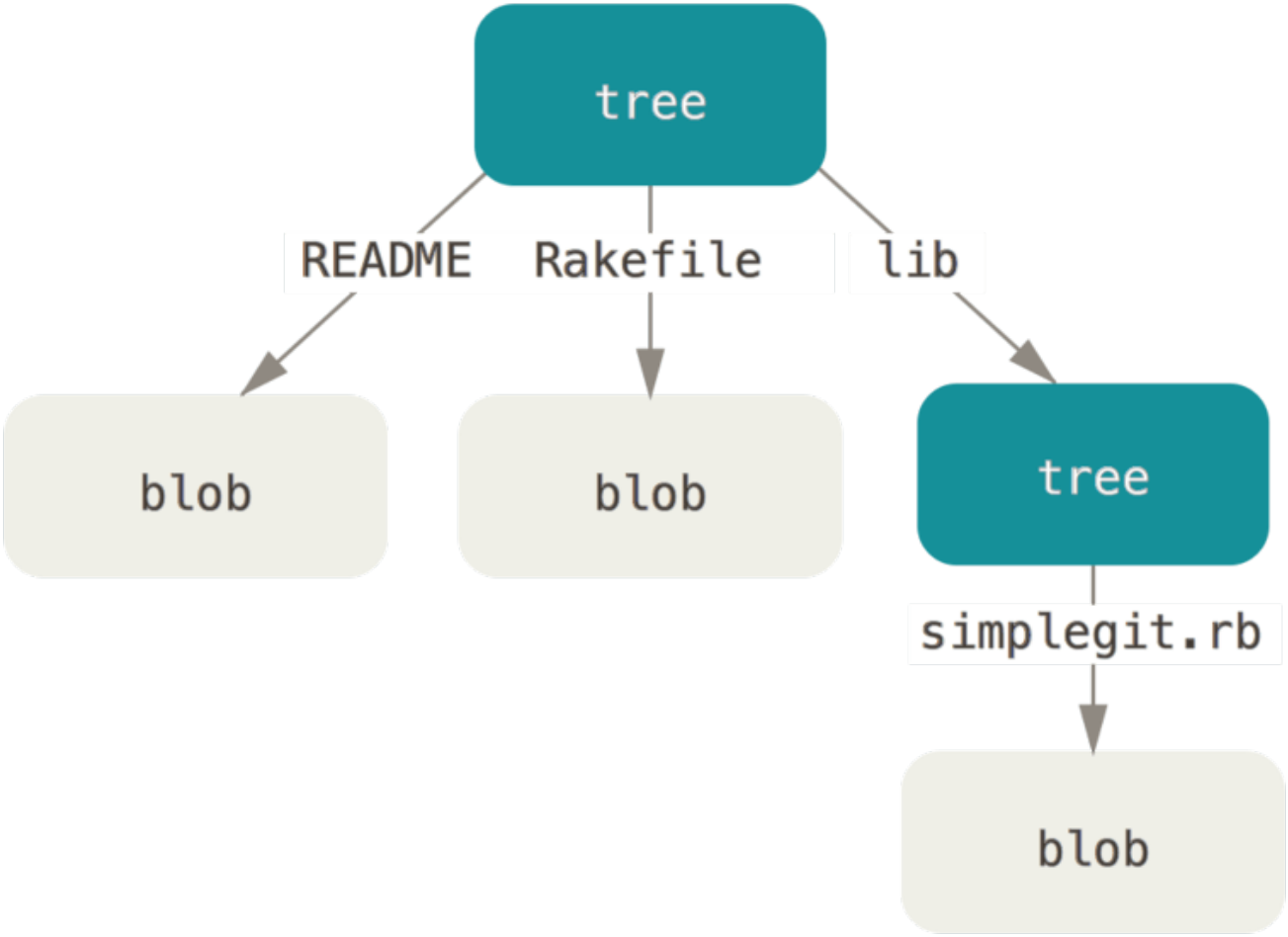



*Figure 148. Gitデータモデルの簡略版*

自前でツリーを作るのも非常に簡単です。 Gitは通常、ステージングエリアやインデックスの状態を取得してツリーを作成し、そのツリーをもとに一連のツリーオブジェクトを書き込みます。 そのため、ツリーオブジェクトを作るには、まずファイルをステージングしてインデックスを作成しなければなりません。 単一のエントリー – ここでは`test.txt`ファイルの最初のバージョン – からインデックスを作るには、`update-index` という配管コマンドを使います。 このコマンドは、前のバージョンの`test.txt`ファイルをあえて新しいステージングエリアに追加する際に使用します。 ファイルはまだステージングエリアには存在しない（まだステージングエリアをセットアップさえしていない）ので、`--add` オプションを付けなければなりません。 また、追加しようとしているファイルはディレクトリには無くデータベースにあるので、`--cacheinfo`オプションを付ける必要があります。 その次に、モード、SHA-1、ファイル名を指定します。

```
$ git update-index --add --cacheinfo 100644 \
  83baae61804e65cc73a7201a7252750c76066a30 test.txt
```

この例では、100644 のモードを指定しています。これは、それが通常のファイルであることを意味します。他に指定できるモードとしては、実行可能ファイルであることを意味する 100755 や、シンボリックリンクであることを示す 120000 があります。 このモードは通常の UNIX モードから取り入れた概念ですが、それほどの柔軟性はありません。Git中のファイル（ブロブ）に対しては、上記3つのモードのみが有効です（ディレクトリとサブモジュールに対しては他のモードも使用できます）。

これで、 write-tree コマンドを使って、ステージングエリアをツリーオブジェクトに書き出せるようになりました。 -w オプションは不要です。write-tree コマンドを呼ぶと、ツリーがまだ存在しない場合には、インデックスの状態をもとに自動的にツリーオブジェクトが作られます。

```
$ git write-tree
d8329fc1cc938780ffdd9f94e0d364e0ea74f579
$ git cat-file -p d8329fc1cc938780ffdd9f94e0d364e0ea74f579
100644 blob 83baae61804e65cc73a7201a7252750c76066a30      test.txt
```

また、これがツリーオブジェクトであることを検証できるようになりました。

```
$ git cat-file -t d8329fc1cc938780ffdd9f94e0d364e0ea74f579
tree
```

今度は、2つめのバージョンの`test.txt`と、新規作成したファイルから、新しくツリーを作ります。

```
$ echo 'new file' > new.txt
$ git update-index test.txt
$ git update-index --add new.txt
```

これでステージングエリアには、`new.txt`という新しいファイルに加えて、新しいバージョンの`test.txt`も登録されました。 このツリーを書き出して（ステージングエリアまたはインデックスの状態をツリーオブジェクトとして記録して）、どのようになったか見てみましょう。

```
$ git write-tree
0155eb4229851634a0f03eb265b69f5a2d56f341
$ git cat-file -p 0155eb4229851634a0f03eb265b69f5a2d56f341
100644 blob fa49b077972391ad58037050f2a75f74e3671e92      new.txt
100644 blob 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a      test.txt
```

このツリーに両方のファイルエントリがあること、また、test.txt`のSHA-1が先ほどの ``version 2''のSHA-1（`1f7a7a）であることに注意してください。 ちょっと試しに、最初のツリーをサブディレクトリとしてこの中に追加してみましょう。 read-tree を呼ぶことで、ステージングエリアの中にツリーを読み込

むことができます。 このケースでは、`--prefix` オプションを付けて `read-tree` コマンドを使用することで、ステージングエリアの中に、既存のツリーをサブツリーとして読み込むことができます。

```
$ git read-tree --prefix=bak d8329fc1cc938780ffdd9f94e0d364e0ea74f579
$ git write-tree
3c4e9cd789d88d8d89c1073707c3585e41b0e614
$ git cat-file -p 3c4e9cd789d88d8d89c1073707c3585e41b0e614
040000 tree d8329fc1cc938780ffdd9f94e0d364e0ea74f579      bak
100644 blob fa49b077972391ad58037050f2a75f74e3671e92      new.txt
100644 blob 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a      test.txt
```

先ほど書き込んだ新しいツリーから作業ディレクトリを作っていれば、作業ディレクトリの直下にファイルが2つと、最初のバージョンの`test.txt`ファイルが含まれている `bak` という名前のサブディレクトリが入ります。 このような構成に対し、Gitが格納するデータのイメージは次のようになります。

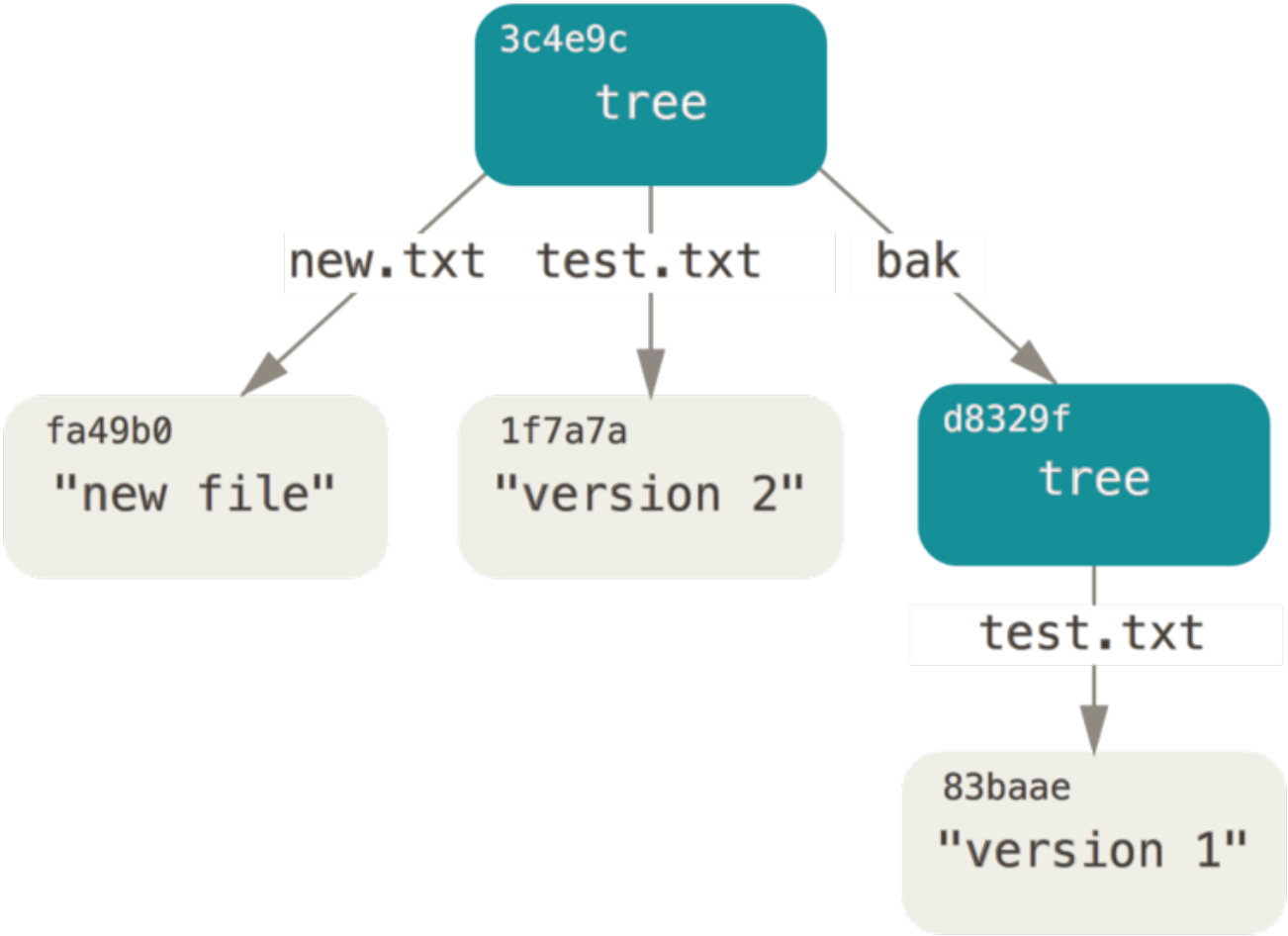



*Figure 149. 現在のGitデータの内容の構成*

## コミットオブジェクト

追跡したいプロジェクトに対し、それぞれ異なる内容のスナップショットを示すツリー3つができました。ですが、各スナップショットを呼び戻すには3つのSHA-1の値すべてを覚えておかなければならない、という以前からの問題は残ったままです。 さらに、そのスナップショットを誰が、いつ、どのような理由で保存したのかについての情報が一切ありません。 これはコミットオブジェクトに保存される基本的な情報です。

コミットオブジェクトを作成するには、ツリーのSHA-1を1つと、もしそれの直前に来るコミットオブジェクトがあれば、それらを指定して `commit-tree` を呼びます。 最初に書き込んだツリーから始めましょう。

```
$ echo 'first commit' | git commit-tree d8329f
fdf4fc3344e67ab068f836878b6c4951e3b15f3d
```

これで、`cat-file` コマンドを使って、新しいコミットオブジェクトを見られるようになりました。

```
$ git cat-file -p fdf4fc3
tree d8329fc1cc938780ffdd9f94e0d364e0ea74f579
author Scott Chacon <schacon@gmail.com> 1243040974 -0700
committer Scott Chacon <schacon@gmail.com> 1243040974 -0700

first commit
```

コミットオブジェクトの形式はシンプルです。その内容は、コミットが作成された時点のスナップショットのトップレベルのツリー、作者とコミッターの情報（`user.name` および `user.email` の設定と現在のタイムスタンプを使用します）、空行、そしてコミットメッセージとなっています。

次に、コミットオブジェクトを新たに2つ書き込みます。各コミットオブジェクトはその直前のコミットを参照しています。

```
$ echo 'second commit' | git commit-tree 0155eb -p fdf4fc3
cac0cab538b970a37ea1e769cbbde608743bc96d
$ echo 'third commit'  | git commit-tree 3c4e9c -p cac0cab
1a410efbd13591db07496601ebc7a059dd55cfe9
```

3つのコミットオブジェクトは、それぞれ、これまでに作成した3つのスナップショットのツリーのひとつを指しています。 奇妙なことに、これで本物のGitヒストリーができており、`git log` コマンドによってログを表示できます。最後のコミットのSHA-1ハッシュを指定して実行すると……

```
$ git log --stat 1a410e
commit 1a410efbd13591db07496601ebc7a059dd55cfe9
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri May 22 18:15:24 2009 -0700

    third commit

 bak/test.txt | 1 +
 1 file changed, 1 insertion(+)

commit cac0cab538b970a37ea1e769cbbde608743bc96d
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri May 22 18:14:29 2009 -0700

    second commit

 new.txt  | 1 +
 test.txt | 2 +-
 2 files changed, 2 insertions(+), 1 deletion(-)

commit fdf4fc3344e67ab068f836878b6c4951e3b15f3d
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri May 22 18:09:34 2009 -0700

    first commit

 test.txt | 1 +
 1 file changed, 1 insertion(+)
```

素晴らしい。 フロントエンドのコマンドを利用せずに、低レベルのオペレーションだけでGitの歴史を作り上げたのです。 これは、本質的に `git add` コマンドと `git commit` コマンドを実行するときにGitが行っていることと同じです。変更されたファイルに対応するブロブを格納し、インデックスを更新し、ツリーを書き出し、トップレベルのツリーと、その直前のコミットを参照するコミットオブジェクトとを書き出しています。 これらの3つの主要な Git オブジェクト – ブロブとツリーとコミット – は、まずは個別のファイルとして `.git/object` ディレクトリに格納されます。 現在、サンプルのディレクトリにあるすべてのオブジェクトを以下に示します。コメントは、それぞれ何を格納しているのかを示します。

```
$ find .git/objects -type f
.git/objects/01/55eb4229851634a0f03eb265b69f5a2d56f341 # tree 2
.git/objects/1a/410efbd13591db07496601ebc7a059dd55cfe9 # commit 3
.git/objects/1f/7a7a472abf3dd9643fd615f6da379c4acb3e3a # test.txt v2
.git/objects/3c/4e9cd789d88d8d89c1073707c3585e41b0e614 # tree 3
.git/objects/83/baae61804e65cc73a7201a7252750c76066a30 # test.txt v1
.git/objects/ca/c0cab538b970a37ea1e769cbbde608743bc96d # commit 2
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4 # 'test content'
.git/objects/d8/329fc1cc938780ffdd9f94e0d364e0ea74f579 # tree 1
.git/objects/fa/49b077972391ad58037050f2a75f74e3671e92 # new.txt
.git/objects/fd/f4fc3344e67ab068f836878b6c4951e3b15f3d # commit 1
```

すべての内部のポインタを辿ってゆけば、次のようなオブジェクトグラフが得られます。

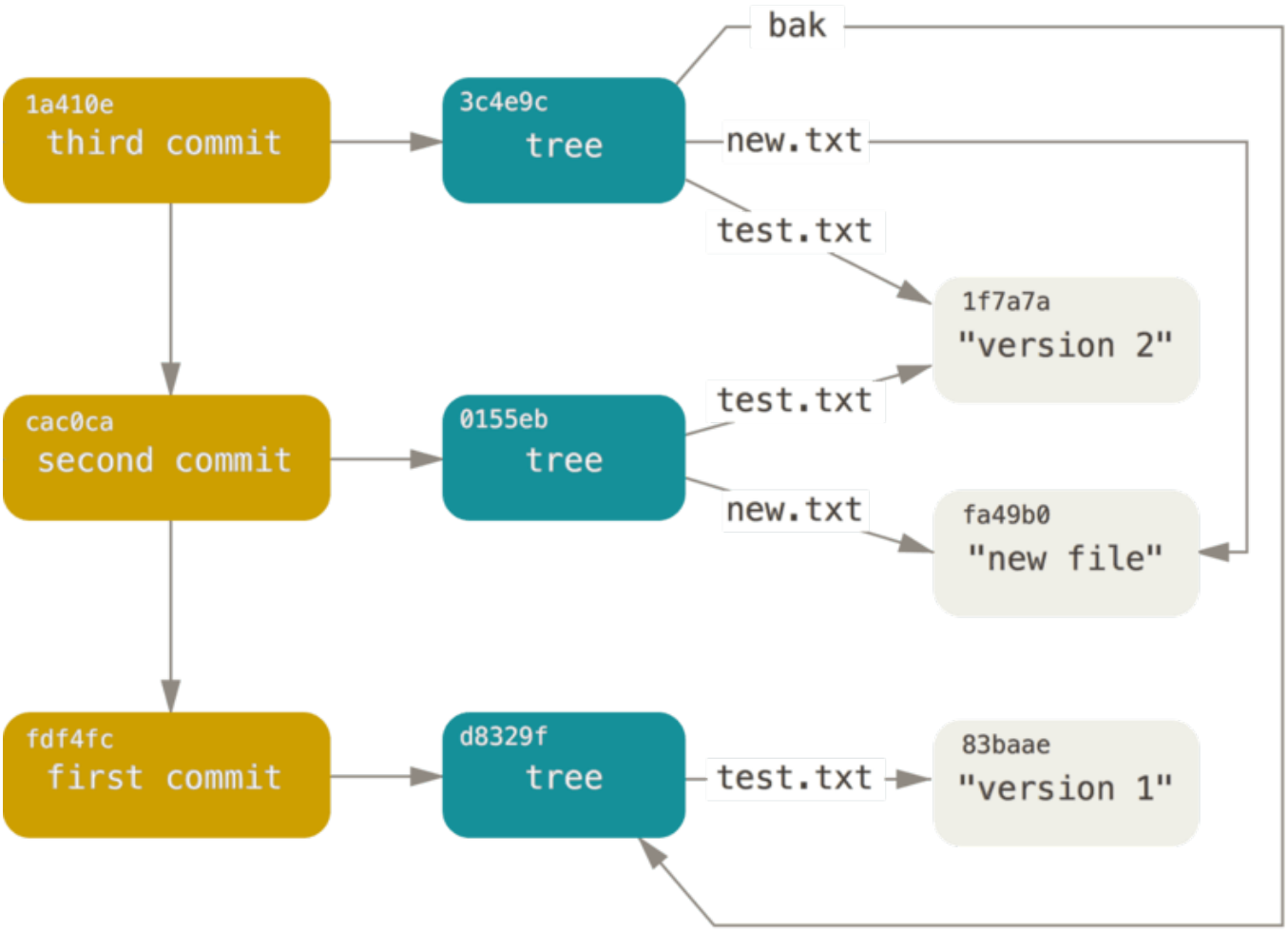



*Figure 150. Gitリポジトリ内のすべてのオブジェクト*

## オブジェクトストレージ

ヘッダはコンテンツと一緒に格納されることを、以前に述べました。 ここでは少し時間を割いて、Gitがどのようにオブジェクトを格納するのかを見ていきましょう。 以降では、ブロブオブジェクト – ここでは “what is up, doc?” という文字列 – をRuby言語を使って対話的に格納する方法を説明します。

`irb` コマンドで、対話モードでRubyを起動します。

```
$ irb
>> content = "what is up, doc?"
=> "what is up, doc?"
```

Gitがヘッダを構築する際には、まず初めにオブジェクトのタイプを表す文字列が来ます。この場合はblobです。 次に、スペースに続いてコンテンツのサイズ、最後にヌルバイトが追加されます。

```
>> header = "blob #{content.length}\0"
=> "blob 16\u0000"
```

Gitはこのヘッダと元々のコンテンツとを結合して、その新しいコンテンツのSHA-1チェックサムを計算します。 Rubyでは、文字列のSHA-1のハッシュ値は、`require` を使用してSHA1ダイジェストライブラリをインクルードし、文字列を引数にして `Digest::SHA1.hexdigest()` 関数を呼ぶことで求められます。

```
>> store = header + content
=> "blob 16\u0000what is up, doc?"
>> require 'digest/sha1'
=> true
>> sha1 = Digest::SHA1.hexdigest(store)
=> "bd9dbf5aae1a3862dd1526723246b20206e5fc37"
```

Gitはzlibを用いてこの新しいコンテンツを圧縮します。Rubyではzlibライブラリをインクルードすれば同じことが行えます。 まず、requireを使用してzlibライブラリをインクルードし、コンテンツに対して `Zlib::Deflate.deflate()` を実行します。

```
>> require 'zlib'
=> true
>> zlib_content = Zlib::Deflate.deflate(store)
=> "x\x9CK\xCA\xC9OR04c(\xCFH,Q\xC8,V(-\xD0QH\xC9O\xB6\a\x00_\x1C\a\x9D"
```

最後に、zlibでdeflate圧縮されたコンテンツをディスク上のオブジェクトに書き込みます。 まず、オブジェクトを書き出す先のパスを決定します（SHA-1ハッシュ値の最初の2文字はサブディレクトリの名前で、残りの38文字はそのディレクトリ内のファイル名になります）。 Rubyでは、サブディレクトリが存在しない場合、 `FileUtils.mkdir_p()` 関数で作成できます。 そして、`File.open()` によってファイルを開いて、前にzlibで圧縮したコンテンツをファイルに書き出します。ファイルへの書き出しは、開いたファイルのハンドルに対して `write()` を呼ぶことで行います。

```
>> path = '.git/objects/' + sha1[0,2] + '/' + sha1[2,38]
=> ".git/objects/bd/9dbf5aae1a3862dd1526723246b20206e5fc37"
>> require 'fileutils'
=> true
>> FileUtils.mkdir_p(File.dirname(path))
=> ".git/objects/bd"
>> File.open(path, 'w') { |f| f.write zlib_content }
=> 32
```

これだけです。これで、正当なGitブロブオブジェクトが出来上がりました。 Gitオブジェクトはすべて同じ方法で格納されますが、オブジェクトのタイプだけは様々で、ヘッダーが blobという文字列ではなく、commitやtreeという文字列で始まることもあります。 また、オブジェクトタイプがブロブの場合、コンテンツはほぼ何でもよいですが、コミットとツリーの場合、コンテンツは非常に厳密に形式が定められています。

## Gitの参照

`git log 1a410e` のように実行すれば、すべての歴史に目を通すことができます。しかし、歴史を辿ってすべてのオブジェクトを探しだすには、 `1a410e` が最後のコミットであることを覚えていなければならないのは変わりません。 SHA-1ハッシュ値をシンプルな名前で保存できれば、生のSHA-1ハッシュ値ではなく、その名前をポインタとして使用できます。

Gitでは、これは “参照” ないしは “refs” と呼ばれます。 `.git/refs` ディレクトリを見ると、SHA-1ハッシュ値を含むファイルがあることが分かります。 現在のプロジェクトでは、このディレクトリにファイルはありませんが、シンプルな構成のディレクトリがあります。

```
$ find .git/refs
.git/refs
.git/refs/heads
.git/refs/tags
$ find .git/refs -type f
```

参照を新しく作成して、最後のコミットがどこかを覚えやすくします。技術的には、以下のように簡単に行えます。

```
$ echo "1a410efbd13591db07496601ebc7a059dd55cfe9" > .git/refs/heads/master
```

これで、Gitコマンドで、SHA-1ハッシュ値の代わりに、たった今作成したhead参照（ブランチ）を使えるようになりました。

```
$ git log --pretty=oneline master
1a410efbd13591db07496601ebc7a059dd55cfe9 third commit
cac0cab538b970a37ea1e769cbbde608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

参照ファイルを直接変更するのは推奨されません。 その代わり、参照をより安全に更新するためのコマンド `update-ref` が、Gitには用意されています。

```
$ git update-ref refs/heads/master
1a410efbd13591db07496601ebc7a059dd55cfe9
```

これは、Gitにおいて、基本的にブランチとは一連の作業の先頭を指す単純なポインタや参照であるということを表しています。 2つ目のコミットが先頭になるブランチを作るには、次のようにします。

```
$ git update-ref refs/heads/test cac0ca
```

作成されたブランチは、さきほど指定したコミット以前の作業のみを含むことになります。

```
$ git log --pretty=oneline test
cac0cab538b970a37ea1e769cbbde608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

この時点で、Gitのデータベースは概念的には以下の図のように見えます。

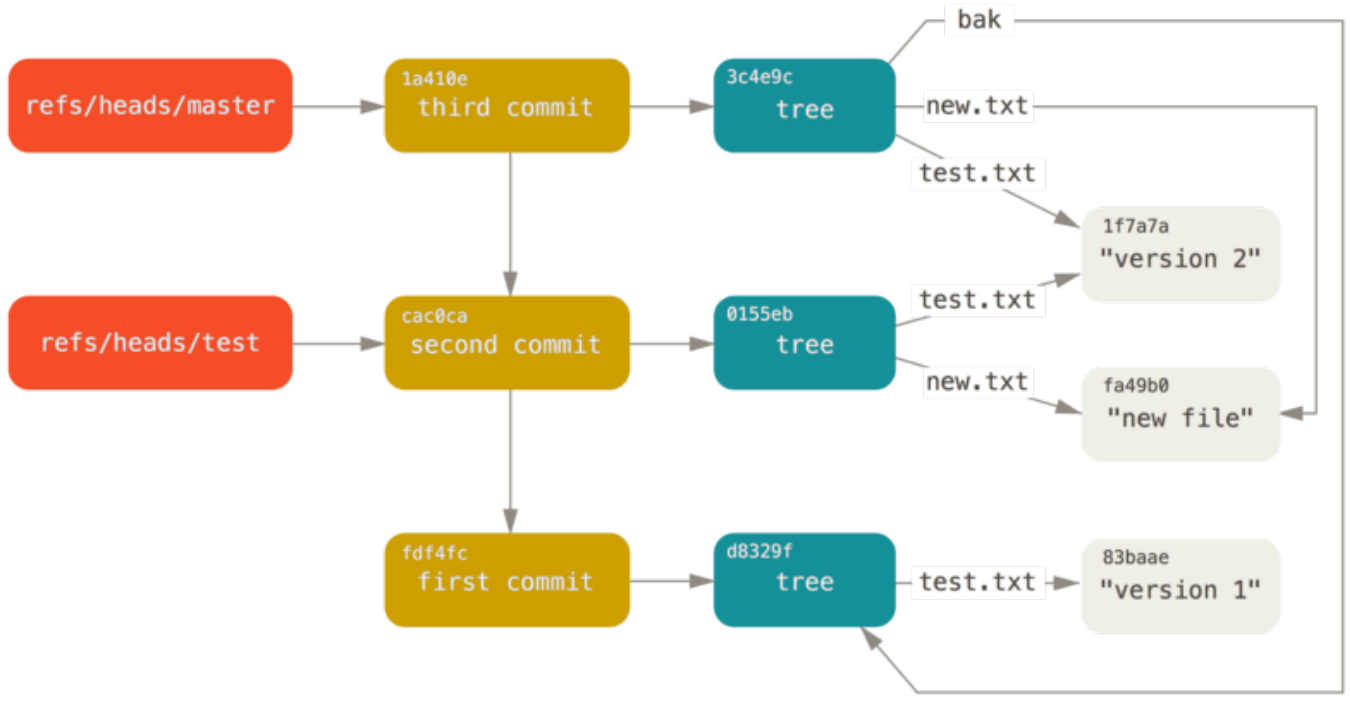



*Figure 151. ブランチの先頭への参照を含むGitディレクトリオブジェクト*

`git branch (ブランチ名)` のようなコマンドを実行すると、あなたが作りたいと思っている新しい参照が何であれ、基本的にGitは `update-ref` コマンドを実行して、いま自分がいるブランチ上の最後のコミットのSHA-1ハッシュをその参照に追加します。

## HEAD

では、`git branch (ブランチ名)` を実行したときに、Gitはどうやって最後のコミットのSHA-1ハッシュを知るのでしょうか？ 答えは、HEADファイルです。

HEADファイルは、現在作業中のブランチに対するシンボリック参照です。 通常の参照と区別する意図でシンボリック参照と呼びますが、これには、一般的にSHA-1ハッシュ値ではなく他の参照へのポインタが格納されています。 ファイルの中身を見ると、通常は以下のようになっています。

```
$ cat .git/HEAD
ref: refs/heads/master
```

`git checkout test` を実行すると、Git はこのようにファイルを更新します。

```
$ cat .git/HEAD
ref: refs/heads/test
```

`git commit` を実行するとコミットオブジェクトが作られますが、そのときコミットオブジェクトの親として、HEADが指し示す参照先のSHA-1ハッシュ値が指定されます。

このファイルを直に編集することもできますが、`symbolic-ref` と呼ばれる、編集を安全に行うためのコマンドが存在します。 このコマンドを使ってHEADの値を読み取ることができます。

```
$ git symbolic-ref HEAD
refs/heads/master
```

HEADの値を設定することもできます。

```
$ git symbolic-ref HEAD refs/heads/test
$ cat .git/HEAD
ref: refs/heads/test
```

`refs` の形式以外では、シンボリック参照を設定することはできません。

```
$ git symbolic-ref HEAD test
fatal: Refusing to point HEAD outside of refs/
```

## タグ

これで Git の主要な三つのオブジェクトを見終わったわけですが、タグという4つ目のオブジェクトがあります。 タグオブジェクトは、コミットオブジェクトに非常によく似ており、タガー、日付、メッセージ、ポインタを格納しています。 主な違いは、タグオブジェクトは通常、ツリーではなくコミットを指しているとい

うことです。 タグオブジェクトはブランチに対する参照に似ていますが、決して変動しません – 常に同じコミットを指しており、より分かりやすい名前が与えられます。

Git の基本 で述べましたが、タグには2つのタイプがあります。軽量 (lightweight) 版と注釈付き (annotated) 版です。 次のように実行すると、軽量版のタグを作成できます。

```
$ git update-ref refs/tags/v1.0 cac0cab538b970a37ea1e769cbbde608743bc96d
```

これが軽量版のタグのすべてです。つまり決して変動しない参照なのです。 一方、注釈付き版のタグはもっと複雑です。 注釈付き版のタグを作ろうとすると、Gitはタグオブジェクトを作った上で、コミットを直接指す参照ではなく、そのタグを指す参照を書き込みます。 注釈付き版のタグを作ると、これが分かります（ `-a` オプションで、注釈付き版のタグを作るよう指定しています）。

```
$ git tag -a v1.1 1a410efbd13591db07496601ebc7a059dd55cfe9 -m 'test tag'
```

作られたオブジェクトのSHA-1ハッシュ値はこうなります。

```
$ cat .git/refs/tags/v1.1
9585191f37f7b0fb9444f35a9bf50de191beadc2
```

ここで、そのSHA-1ハッシュ値に対して `cat-file` コマンドを実行します。

```
$ git cat-file -p 9585191f37f7b0fb9444f35a9bf50de191beadc2
object 1a410efbd13591db07496601ebc7a059dd55cfe9
type commit
tag v1.1
tagger Scott Chacon <schacon@gmail.com> Sat May 23 16:48:58 2009 -0700

test tag
```

`object` の項目が、上でタグ付けしたコミットのSHA-1ハッシュ値を指していることに注意してください。 また、この項目が必ずしもコミットだけをポイントするものではないことも覚えておいてください。あらゆるGitオブジェクトに対してタグを付けることができます。 例えば Git のソースコードリポジトリでは、メンテナが自分のGPG公開鍵をブロブオブジェクトとして追加し、そのオブジェクトにタグを付けています。 Git リポジトリのクローン上で、以下のコマンドを実行すると公開鍵を閲覧できます。

```
$ git cat-file blob junio-gpg-pub
```

Linux カーネルのリポジトリもまた、`object` 項目でコミット以外を指しているタグオブジェクトを持っています。 これは最初のタグオブジェクトであり、最初にソースコードをインポートしたときの初期ツリーオブジェクトを指しています。

### リモート

これから見ていく3つ目の参照のタイプはリモート参照です。 リモートを追加してそこにプッシュすると、Gitはそのリモートへ最後にプッシュした値を、ブランチ毎に `refs/remotes` へ格納します。 例えば、`origin` というリモートを追加して、そこに `master` ブランチをプッシュしたとします。

```
$ git remote add origin git@github.com:schacon/simplegit-progit.git
$ git push origin master
Counting objects: 11, done.
Compressing objects: 100% (5/5), done.
Writing objects: 100% (7/7), 716 bytes, done.
Total 7 (delta 2), reused 4 (delta 1)
To git@github.com:schacon/simplegit-progit.git
  a11bef0..ca82a6d  master -> master
```

ここで `refs/remotes/origin/master` ファイルの中身を確認してみてください。最後にサーバーと通信したときに `origin` リモートの `master` ブランチが何であったかがわかるはずです。

```
$ cat .git/refs/remotes/origin/master
ca82a6dff817ec66f44342007202690a93763949
```

リモート参照は、特に読み取り専用とみなされる点において、ブランチ（`refs/heads` にある参照）とは異なります。 リモート参照に対して `git checkout` を行うことはできますが、GitはHEADの参照先をそのリモートにすることはなく、したがって `commit` コマンドでリモートを更新することもできません。 Gitはリモート参照を一種のブックマークとして管理します。つまり、最後に通信したとき、向こうのサーバー上でリモートブランチが置かれていた状態を指し示すブックマークということです。

## Packfile

Gitリポジトリtestのオブジェクトデータベースに戻りましょう。 この時点で、オブジェクトは11個あります。内訳はブロブが4つ、ツリーが3つ、コミットが3つ、そしてタグが1つです。

```
$ find .git/objects -type f
.git/objects/01/55eb4229851634a0f03eb265b69f5a2d56f341 # tree 2
.git/objects/1a/410efbd13591db07496601ebc7a059dd55cfe9 # commit 3
.git/objects/1f/7a7a472abf3dd9643fd615f6da379c4acb3e3a # test.txt v2
.git/objects/3c/4e9cd789d88d8d89c1073707c3585e41b0e614 # tree 3
.git/objects/83/baae61804e65cc73a7201a7252750c76066a30 # test.txt v1
.git/objects/95/85191f37f7b0fb9444f35a9bf50de191beadc2 # tag
.git/objects/ca/c0cab538b970a37ea1e769cbbde608743bc96d # commit 2
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4 # 'test content'
.git/objects/d8/329fc1cc938780ffdd9f94e0d364e0ea74f579 # tree 1
.git/objects/fa/49b077972391ad58037050f2a75f74e3671e92 # new.txt
.git/objects/fd/f4fc3344e67ab068f836878b6c4951e3b15f3d # commit 1
```

Gitはzlibを使用してこれらのファイルの内容を圧縮します。また、格納しているものもそれほど多くないため、これらすべてのファイルを集めても925バイトにしかなりません。 Gitの興味深い機能を実際に見てみるために、幾つか大きなコンテンツをリポジトリに追加してみましょう。 実例を示すために、Gritライブラリから `repo.rb` ファイルを追加します。これは約22Kバイトのソースコードファイルです。

```
$ curl
https://raw.githubusercontent.com/mojombo/grit/master/lib/grit/repo.rb >
repo.rb
$ git add repo.rb
$ git commit -m 'added repo.rb'
[master 484a592] added repo.rb
 3 files changed, 709 insertions(+), 2 deletions(-)
 delete mode 100644 bak/test.txt
 create mode 100644 repo.rb
 rewrite test.txt (100%)
```

結果のツリーを見ると、`repo.rb` ファイルに対応するブロブオブジェクトのSHA-1ハッシュ値が分かります。

```
$ git cat-file -p master^{tree}
100644 blob fa49b077972391ad58037050f2a75f74e3671e92      new.txt
100644 blob 033b4468fa6b2a9547a70d88d1bbe8bf3f9ed0d5      repo.rb
100644 blob e3f094f522629ae358806b17daf78246c27c007b      test.txt
```

それから、`git cat-file` を使用すれば、そのオブジェクトの大きさも分かります。

```
$ git cat-file -s 033b4468fa6b2a9547a70d88d1bbe8bf3f9ed0d5
22044
```

ここで、このファイルに少し変更を加えて、何が起こるか見てみましょう。

```
$ echo '# testing' >> repo.rb
$ git commit -am 'modified repo a bit'
[master 2431da6] modified repo.rb a bit
 1 file changed, 1 insertion(+)
```

このコミットによって作られたツリーを見てみると、興味深いことがわかります。

```
$ git cat-file -p master^{tree}
100644 blob fa49b077972391ad58037050f2a75f74e3671e92      new.txt
100644 blob b042a60ef7dff760008df33cee372b945b6e884e      repo.rb
100644 blob e3f094f522629ae358806b17daf78246c27c007b      test.txt
```

repo.rb に対応するブロブが、別のブロブになっています。つまり、400行あるファイルの最後に1行だけ追加しただけなのに、Git はその新しいコンテンツを完全に新しいオブジェクトとして格納するのです。

```
$ git cat-file -s b042a60ef7dff760008df33cee372b945b6e884e
22054
```

これだと、ディスク上にほとんど同じ内容の22Kバイトのオブジェクトが2つあることになります。 もし、Gitが2つのうち1つは完全に格納し、2つめのオブジェクトは1つめとの差分のみを格納できたら、素晴らしいと思いませんか？

実は、それができるのです。 Gitが最初にディスク上にオブジェクトを格納する際のフォーマットは、“緩い”（“loose”）オブジェクトフォーマットと呼ばれます。 一方、容量の節約と効率化のため、Gitはときどき、緩いフォーマットのオブジェクトの中の幾つかを1つのバイナリファイルにパックします。このバイナリファイルを “packfile” と呼びます。 あまりにたくさんの緩いオブジェクトがそこら中にあるときや、git gc コマンドを手動で実行したとき、または、リモートサーバーにプッシュしたときに、Gitはパック処理を行います。 何が起こるのかを知りたいなら、 git gc コマンドを呼べば、オブジェクトをパックするよう手動でGitに指示できます。

```
$ git gc
Counting objects: 18, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (14/14), done.
Writing objects: 100% (18/18), done.
Total 18 (delta 3), reused 0 (delta 0)
```

オブジェクトディレクトリの中を見ると、大半のオブジェクトは削除され、新しいファイルが2つ作られたことがわかります。

```
$ find .git/objects -type f
.git/objects/bd/9dbf5aae1a3862dd1526723246b20206e5fc37
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
.git/objects/info/packs
.git/objects/pack/pack-978e03944f5c581011e6998cd0e9e30000905586.idx
.git/objects/pack/pack-978e03944f5c581011e6998cd0e9e30000905586.pack
```

削除されずに残ったオブジェクトは、どのコミットからも指されていないブロブです。このケースでは、以前に作成した “what is up, doc?” の例のブロブと “test content” の例のブロブが該当します。 これらのブロブはどのコミットにも加えられなかったため、宙ぶらりんになっていると見なされ、新しいpackfileにはパックされません。

残りの2つのファイルが、新しく作られたpackfileとインデックスです。 packfileには、ファイルシステムから削除されたすべてのオブジェクトの内容が1つのファイルに含まれています。 インデックスには、特定のオブジェクトを速くシークできるように、packfile中でのオフセットが記録されています。 素晴らしいことに、gc を実行する前のディスク上のオブジェクトは合計で約22Kバイトあったのに対して、新しいパックフ

ァイルはたった7Kバイトになっています。
オブジェクトをパックすることで、ディスク使用量の⍰を削減できたのです。

Gitはどうやってこれを行うのでしょうか？ Git はオブジェクトをパックするとき、似たような名前とサイズのファイルを探し出し、ファイルのあるバージョンから次のバージョンまでの差分のみを格納します。packfileの中を見ることで、容量を節約するためにGitが何を行ったのかを知ることができます。 `git verify-pack` という配管コマンドを使用して、何がパックされているのか見ることができます。

```
$ git verify-pack -v .git/objects/pack/pack-
978e03944f5c581011e6998cd0e9e30000905586.idx
2431da676938450a4d72e260db3bf7b0f587bbc1 commit 223 155 12
69bcdaff5328278ab1c0812ce0e07fa7d26a96d7 commit 214 152 167
80d02664cb23ed55b226516648c7ad5d0a3deb90 commit 214 145 319
43168a18b7613d1281e5560855a83eb8fde3d687 commit 213 146 464
092917823486a802e94d727c820a9024e14a1fc2 commit 214 146 610
702470739ce72005e2edff522fde85d52a65df9b commit 165 118 756
d368d0ac0678cbe6cce505be58126d3526706e54 tag    130 122 874
fe879577cb8cffcdf25441725141e310dd7d239b tree   136 136 996
d8329fc1cc938780ffdd9f94e0d364e0ea74f579 tree   36 46 1132
deef2e1b793907545e50a2ea2ddb5ba6c58c4506 tree   136 136 1178
d982c7cb2c2a972ee391a85da481fc1f9127a01d tree   6 17 1314 1 \
  deef2e1b793907545e50a2ea2ddb5ba6c58c4506
3c4e9cd789d88d8d89c1073707c3585e41b0e614 tree   8 19 1331 1 \
  deef2e1b793907545e50a2ea2ddb5ba6c58c4506
0155eb4229851634a0f03eb265b69f5a2d56f341 tree   71 76 1350
83baae61804e65cc73a7201a7252750c76066a30 blob   10 19 1426
fa49b077972391ad58037050f2a75f74e3671e92 blob   9 18 1445
b042a60ef7dff760008df33cee372b945b6e884e blob   22054 5799 1463
033b4468fa6b2a9547a70d88d1bbe8bf3f9ed0d5 blob   9 20 7262 1 \
  b042a60ef7dff760008df33cee372b945b6e884e
1f7a7a472abf3dd9643fd615f6da379c4acb3e3a blob   10 19 7282
non delta: 15 objects
chain length = 1: 3 objects
.git/objects/pack/pack-978e03944f5c581011e6998cd0e9e30000905586.pack: ok
```

ここで `033b4` というブロブを覚えているでしょうか。これは `repo.rb` ファイルの最初のバージョンですが、このブロブは2つ目のバージョンである `b042a` というブロブを参照しています。 出力の3つ目のカラムはpackfile中のオブジェクトのサイズを示しています。ここで、`b042a` はファイルのうち22Kバイトを占めていますが、`033b4` はたったの9バイトしか占めていないことがわかります。 さらに興味深いのは、2つめのバージョンのファイルは元のままの状態で格納されているのに対して、最初のバージョンは増分として格納されていることです。これは、直近のバージョンのファイルほど、高速にアクセスしたいだろうというのが理由です。

この機能の本当に素晴らしいのは、いつでも再パックが可能なことです。 Git は時折データベースを自動的に再パックして、常に容量をより多く節約しようと努めますが、自分で `git gc` を実行すれば、いつでも手動で再パックを行えます。

# Refspec

本書の全体に渡って、リモートブランチからローカルの参照へのシンプルなマッピングを使用してきましたが、もっと複雑な場合もあります。以下のようにリモートを追加したとしましょう。

```
$ git remote add origin https://github.com/schacon/simplegit-progit
```

`.git/config` ファイルにセクションを追加して、リモートの名前（`origin`）、リモートリポジトリのURL、そしてフェッチする対象のrefspecを指定します。

```
[remote "origin"]
    url = https://github.com/schacon/simplegit-progit
    fetch = +refs/heads/*:refs/remotes/origin/*
```

refspecの書式は `<src>:<dst>` で、その前にオプションとして `+` を指定できます。ここで `<src>` はリモート側の参照に対するパターンで、 `<dst>` はそれらの参照がローカルで書きこまれる場所を示します。 `+` は、fast-forwardでない場合でも参照を更新するようGitに指示しています。

デフォルトでは、 `git remote add` コマンドを実行すると、自動的にこの設定が書き込まれ、Gitはサーバー上の `refs/heads/` 以下にあるすべての参照をフェッチして、ローカルの `refs/remotes/origin/` に書き込みます。 そのため、サーバー上に `master` ブランチがあるとすると、ローカルでは、そのブランチのログには以下のコマンドでアクセスできます。

```
$ git log origin/master
$ git log remotes/origin/master
$ git log refs/remotes/origin/master
```

これらはすべて同じ意味を持ちます。なぜなら、どれもGitにより `refs/remotes/origin/master` に展開されるからです。

逆に、常にリモートサーバー上の `master` ブランチのみをプルして、それ以外のブランチはどれもプルしたくない場合は、fetchの行を以下のように変更します。

```
fetch = +refs/heads/master:refs/remotes/origin/master
```

これは、このリモートに対する `git fetch` のデフォルトのrefspecそのものです。 もし、設定内容とは違う内容を一度だけプルしたければ、コマンドライン上でもrefspecを指定できます。 リモートの `master` ブランチを、ローカルの `origin/mymaster` にプルするには、以下のように実行します。

```
$ git fetch origin master:refs/remotes/origin/mymaster
```

複数のrefspecを指定することも可能です。 コマンドライン上で、以下のように複数のブランチをプルできま

す。

```
$ git fetch origin master:refs/remotes/origin/mymaster \
     topic:refs/remotes/origin/topic
From git@github.com:schacon/simplegit
 ! [rejected]        master     -> origin/mymaster  (non fast forward)
 * [new branch]      topic      -> origin/topic
```

このケースでは、`master` ブランチのプルはfast-forwardの参照ではなかったため拒否されました。 refspecの先頭に `+` を指定すると、この動作を上書きできます。

さらに、設定ファイルに、フェッチ用のrefspecを複数指定することもできます。 もし、常に `master` ブランチと `experiment` ブランチをフェッチしたいならば、以下のように2行追加します。

```
[remote "origin"]
    url = https://github.com/schacon/simplegit-progit
    fetch = +refs/heads/master:refs/remotes/origin/master
    fetch = +refs/heads/experiment:refs/remotes/origin/experiment
```

パターン中で、ファイル名の一部だけをワイルドカード指定したグロブを使うことはできません。以下の指定は無効となります。

```
fetch = +refs/heads/qa*:refs/remotes/origin/qa*
```

しかし、名前空間（やディレクトリ）を使って、似たようなことは行えます。 一連のブランチをプッシュしてくれるQAチームがいたとして、masterブランチとQAチームのブランチのみを取得したいならば、該当セクションを以下のように使用すればよいでしょう。

```
[remote "origin"]
    url = https://github.com/schacon/simplegit-progit
    fetch = +refs/heads/master:refs/remotes/origin/master
    fetch = +refs/heads/qa/*:refs/remotes/origin/qa/*
```

仮に、QAチームがブランチをプッシュし、開発者チームもブランチをプッシュし、さらに統合チームもブランチをプッシュしたりリモートブランチを使って共同で作業をしたりするような複雑なワークフローに従っているとしましょう。そういった場合でも、上述のように設定しておけば簡単に名前空間を分けることができます。

## refspecへのプッシュ

このように、名前空間を分けた参照をフェッチできるのは素晴らしいことです。しかし、そもそもQAチームは、どうすれば自分たちのブランチを `qa/` という名前空間に格納できるのでしょうか? プッシュの際にrefspecを使えばそれが可能です。

QAチームが自分たちの `master` ブランチをリモートサーバー上の `qa/master` にプッシュしたい場合、以下のように実行します。

```
$ git push origin master:refs/heads/qa/master
```

QAチームが `git push origin` を実行する度に、Gitに自動的にこの処理を行ってほしいなら、設定ファイルに `push` の値を追加することもできます。

```
[remote "origin"]
    url = https://github.com/schacon/simplegit-progit
    fetch = +refs/heads/*:refs/remotes/origin/*
    push = refs/heads/master:refs/heads/qa/master
```

このように設定しておくと、 `git push origin` を実行したときに、デフォルトでローカルの `master` ブランチをリモートの `qa/master` ブランチにプッシュするようになります。

### 参照の削除

また、refspecを使ってリモートサーバーから参照を削除することもできます。削除するには以下のコマンドを実行します。

```
$ git push origin :topic
```

refspecは `<src>:<dst>` という書式なので、 `<src>` の部分を取り除くと、要するにリモート上の `topic` ブランチを空にせよという指示になり、リモート上の参照が削除されます。

## 転送プロトコル

Gitが2つのリポジトリ間でデータを転送する方法には、主に “dumb” プロトコルと “smart” プロトコルの2つがあります。 このセクションでは、これらのプロトコルがどのように機能するのかを駆け足で見ていきます。

### dumbプロトコル

HTTP経由でのリポジトリへのアクセスを読み取り専用にする場合、dumbプロトコルを使うことになると思います。 このプロトコルを “dumb” （馬鹿）と呼ぶのは、転送プロセスにおいて、サーバー側にGit専用のコードが不要だからです。フェッチのプロセスは一連のHTTP `GET` リクエストです。ここで、クライアントは、サーバー上のGitリポジトリのレイアウトを仮定してよいことになっています。

**NOTE**

dumbプロトコルは昨今ではほとんど使用されていません。 安全性や秘匿性を保つのが難しいため、多くのGitのホスト（クラウドベースでも、オンプレミスでも）では使用が禁止されています。 一般的には、もう少し後で述べるsmartプロトコルを使用することをおすすめします。

simplegitライブラリにおける `http-fetch` のプロセスを追ってみましょう。

```
$ git clone http://server/simplegit-progit.git
```

このコマンドは最初に `info/refs` ファイルをサーバから取得します。 このファイルは `update-server-info` コマンドによって出力されます。そのため、HTTPによる転送を適切に動作させるためには、このコマンドを `post-receive` フック中で呼び出す必要があります。

```
=> GET info/refs
ca82a6dff817ec66f44342007202690a93763949     refs/heads/master
```

いま、手元にはリモート参照とSHA-1のハッシュのリストがあります。 次に、HEADが指しているものを見て、終了時に何をチェックアウトするのかを調べます。

```
=> GET HEAD
ref: refs/heads/master
```

プロセスの完了時には、`master` ブランチをチェックアウトする必要があると分かりました。 これで、参照を辿るプロセスを開始する準備ができました。 開始地点は `info/refs` ファイルの中にあった `ca82a6` のコミットオブジェクトなので、まずそれを取得します。

```
=> GET objects/ca/82a6dff817ec66f44342007202690a93763949
(179 bytes of binary data)
```

すると、オブジェクトが返ってきます。これは、サーバー上にある緩いフォーマットのオブジェクトで、それを静的なHTTP GETリクエストで取得したわけです。 このオブジェクトのzlib圧縮を解除し、ヘッダを取り除けば、コミットの内容が見られます。

```
$ git cat-file -p ca82a6dff817ec66f44342007202690a93763949
tree cfda3bf379e4f8dba8717dee55aab78aef7f4daf
parent 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
author Scott Chacon <schacon@gmail.com> 1205815931 -0700
committer Scott Chacon <schacon@gmail.com> 1240030591 -0700

changed the version number
```

もう2つ、オブジェクトを取得する必要があることが分かりました。 たった今取得したコミットが指しているコンテンツのツリーである `cfda3b` と、親にあたるコミットである `085bb3` です。

```
=> GET objects/08/5bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
(179 bytes of data)
```

まずは親にあたるオブジェクトを取得しました。 続いてツリーオブジェクトを取得してみましょう。

```
=> GET objects/cf/da3bf379e4f8dba8717dee55aab78aef7f4daf
(404 - Not Found)
```

おっと、そのツリーオブジェクトは緩いフォーマットではサーバー上に存在しないようです。そのため404のレスポンスを受け取っています。 考えられる理由は2つあります。オブジェクトが代替のリポジトリにあるためか、またはこのリポジトリ内のpackfileに含まれているためです。 Gitはまず、代替のリポジトリの一覧を調べます。

```
=> GET objects/info/http-alternates
(empty file)
```

このGETリクエストに対して代替のURLのリストが返ってきた場合、Gitはその場所から緩いフォーマットのファイルとpackfileを探します。これは、プロジェクトがディスク上のオブジェクトを共有するために互いにフォークし合っている場合に適したメカニズムです。 ですが、このケースでは代替URLのリストは空だったので、オブジェクトはpackfileの中にあるに違いありません。 サーバー上のアクセス可能なpackfileの一覧は、 `objects/info/packs` ファイルに格納されているので、これを取得する必要があります（このファイルも `update-server-info` で生成されます）。

```
=> GET objects/info/packs
P pack-816a9b2334da9953e530f27bcac22082a9f5b835.pack
```

サーバー上にはpackfileが1つしかないので、探しているオブジェクトは明らかにこの中にあります。しかし念の為にインデックスファイルをチェックしてみましょう。 これにより、サーバー上にpackfileが複数ある場合でも、必要なオブジェクトがどのpackfileに含まれているか調べられます。

```
=> GET objects/pack/pack-816a9b2334da9953e530f27bcac22082a9f5b835.idx
(4k of binary data)
```

packfileのインデックスが取得できたので、これで探しているオブジェクトがpackfileの中にあるか調べられます – なぜなら、インデックスにはpackfileの中にあるオブジェクトのSHA-1ハッシュと、それらのオブジェクトに対するオフセットの一覧が格納されているからです。 探しているオブジェクトは、どうやらそこにあるようです。さあ、そのpackfileをまるごと取得してみましょう。

```
=> GET objects/pack/pack-816a9b2334da9953e530f27bcac22082a9f5b835.pack
(13k of binary data)
```

探していたツリーオブジェクトが見つかりました。さらにコミットを辿ってみましょう。 コミットはいずれも、先ほどダウンロードしたpackfileの中にあります。そのため、もうサーバーに対するリクエストは不要です。 Gitは、最初にダウンロードしたHEADが指している `master` ブランチの作業用コピーをチェックアウトします。

## smartプロトコル

dumbプロトコルはシンプルですが、少し非効率ですし、クライアントからサーバーへのデータの書き込みも行えません。 データ移行においては、smartプロトコルの方がより一般的な手段です。ただし、リモート側にGitと対話できるプロセス – ローカルのデータを読んだり、クライアントが何を持っていて何が必要としているかを判別したり、それに応じたpackfileを生成したりできるプロセス – が必要です。 データの転送には、プロセスを2セット使用します。データをアップロードするペアと、ダウンロードするペアです。

### データのアップロード

リモートプロセスにデータをアップロードする際、Gitは `send-pack` プロセスと `receive-pack` プロセスを使用します。`send-pack` プロセスはクライアント上で実行されリモート側の `receive-pack` プロセスに接続します。

#### SSH

例えば、あなたのプロジェクトで `git push origin master` を実行するとします。そして `origin` はSSHプロトコルを使用するURLとして定義されているとします。 この際、Gitは `send-pack` プロセスを起動して、あなたのサーバーへのSSH接続を開始します。 このプロセスは、以下のようなSSHの呼び出しを介して、リモートサーバー上でコマンドを実行しようとします。

```
$ ssh -x git@server "git-receive-pack 'simplegit-progit.git'"
00a5ca82a6dff817ec66f4437202690a93763949 refs/heads/master□report-status \
    delete-refs side-band-64k quiet ofs-delta \
    agent=git/2:2.1.1+github-607-gfba4028 delete-refs
0000
```

`git-receive-pack` コマンドは、今ある参照1つにつき1行の応答を、その都度返します。このケースでは、`master` ブランチとそのSHA-1ハッシュのみを返しています。 最初の行には、サーバーの持っている機能（ここでは、`report-status` や `delete-refs` など。クライアント識別子も含む）のリストも含まれています。

各行は4文字の16進数で始まっており、その行の残りがどれくらいの長さなのかを示しています。 最初の行は00a5で始まっていますが、これは16進数で165を示し、その行はあと165バイトあることを意味します。 次の行は0000であり、サーバーが参照のリストの表示を終えたことを意味します。

サーバーの状態がわかったので、これで `send-pack` プロセスは、自分の側にあってサーバー側にないコミットを判別できます。 これからこのプッシュで更新される各参照について、`send-pack` プロセスは `receive-pack` プロセスにその情報を伝えます。 例えば、 `master` ブランチの更新と `experiment` ブランチの追加をしようとしている場合、 `send-pack` のレスポンスは次のようになるでしょう。

```
0076ca82a6dff817ec66f44342007202690a93763949
15027957951b64cf874c3557a0f3547bd83b3ff6 \
    refs/heads/master report-status
006c0000000000000000000000000000000000000000
cdfdb42577e2506715f8cfeacdbabc092bf63e8d \
    refs/heads/experiment
0000
```

Gitは更新しようとしている参照のそれぞれに対して、行の長さ、古いSHA-1、新しいSHA-1、更新される参照を含む行を送信します。 最初の行にはクライアントの持っている機能も含まれています。 すべてが *0* のSHA-1ハッシュ値は、以前そこには何もなかったことを意味します。それはあなたが experiment の参照を追加しているためです。 もしもあなたが参照を削除していたとすると、逆にすべてが *0* のSHA-1ハッシュ値が右側に表示されるはずです。

次に、クライアントは、まだサーバー側にないオブジェクトすべてを含むpackfileを送信します。 最後に、サーバーは成功（あるいは失敗）を示す内容を返します。

```
000eunpack ok
```

#### HTTP(S)

このプロセスは、ハンドシェイクが少し違うだけで、HTTP経由の場合とほとんど同じです。 接続は以下のリクエストで初期化されます。

```
=> GET http://server/simplegit-progit.git/info/refs?service=git-receive-
pack
001f# service=git-receive-pack
00ab6c5f0e45abd7832bf23074a333f739977c9e8188 refs/heads/master□report-
status \
    delete-refs side-band-64k quiet ofs-delta \
    agent=git/2:2.1.1~vmg-bitmaps-bugaloo-608-g116744e
0000
```

これで初回のクライアント・サーバー間の交信は終了です。 クライアントは次に別のリクエストを作成します。この場合は `send-pack` が提供するデータをもとに `POST` リクエストを作成します。

```
=> POST http://server/simplegit-progit.git/git-receive-pack
```

この `POST` リクエストには `send-pack` の出力とpackfileがペイロードとして含まれています。 サーバーはこれに対して成功か失敗かをHTTPレスポンスで示します。

### データのダウンロード

データをダウンロードするときには、 `fetch-pack` と `upload-pack` の2つのプロセスが使用されます。 クライアントが `fetch-pack` プロセスを起動すると、リモート側の `upload-pack` プロセスに接続してネゴシエーションを行い、何のデータをダウンロードするか決定します。

#### SSH

SSHを介してフェッチを行っているなら、`fetch-pack` は以下のようなコマンドを実行します。

```
$ ssh -x git@server "git-upload-pack 'simplegit-progit.git'"
```

`fetch-pack` の接続のあと、`upload-pack` は以下のような内容を返信します。

```
00dfca82a6dff817ec66f44342007202690a93763949 HEAD□multi_ack thin-pack \
    side-band side-band-64k ofs-delta shallow no-progress include-tag \
    multi_ack_detailed symref=HEAD:refs/heads/master \
    agent=git/2:2.1.1+github-607-gfba4028
003fe2409a098dc3e53539a9028a94b6224db9d6a6b6 refs/heads/master
0000
```

これは `receive-pack` が返す内容にとても似ていますが、持っている機能は異なります。 加えて、HEADがどこを指しているか (`symref=HEAD:refs/heads/master`) を返すので、クローン処理の場合、クライアントが何をチェックアウトするのかを知ることができます。

この時点で、 `fetch-pack` プロセスは手元にあるオブジェクトを確認します。そして、必要なオブジェクトを返答するため、“want” という文字列に続けて必要なオブジェクトのSHA-1ハッシュを送ります。 また、既に持っているオブジェクトについては、“have” という文字列に続けてオブジェクトのSHA-1ハッシュを送ります。 さらに、このリストの最後には “done” を書き込んで、必要なデータのpackfileを送信する `upload-pack` プロセスを開始します。

```
003cwant ca82a6dff817ec66f44342007202690a93763949 ofs-delta
0032have 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
0009done
0000
```

#### HTTP(S)

フェッチ操作のためのハンドシェイクは2つのHTTPリクエストからなります。 1つめはdumbプロトコルで使用するのと同じエンドポイントへの `GET` です。

```
=> GET $GIT_URL/info/refs?service=git-upload-pack
001e# service=git-upload-pack
00e7ca82a6dff817ec66f44342007202690a93763949 HEAD□multi_ack thin-pack \
    side-band side-band-64k ofs-delta shallow no-progress include-tag \
    multi_ack_detailed no-done symref=HEAD:refs/heads/master \
    agent=git/2:2.1.1+github-607-gfba4028
003fca82a6dff817ec66f44342007202690a93763949 refs/heads/master
0000
```

これはSSH接続経由で `git-upload-pack` を呼び出す場合と非常によく似ていますが、2つ目の交信が個別のリクエストとして実行される点が異なります。

```
=> POST $GIT_URL/git-upload-pack HTTP/1.0
0032want 0a53e9ddeaddad63ad106860237bbf53411d11a7
0032have 441b40d833fdfa93eb2908e52742248faf0ee993
0000
```

これもまた、上と同じフォーマットです。 このリクエストに対するレスポンスは、成功したか失敗したかを示しています。また、packfileも含まれています。

### プロトコルのまとめ

このセクションでは転送プロトコルの最も基本的な概要を取り上げました。 プロトコルには他にも `multi_ack` や `side-band` など数多くの機能がありますが、それらは本書の範囲外です。 ここでは、一般的なクライアントとサーバーの間の行き来に関する感覚を捉えてもらえるよう努めました。これ以上の知識が必要な場合は、おそらくGitのソースコードを見てみる必要があるでしょう。

## メンテナンスとデータリカバリ

たまには、ちょっとしたお掃除 – リポジトリを圧縮したり、インポートしたリポジトリをクリーンアップしたり、失われた成果物をもとに戻したり – が必要になるかもしれません。 このセクションではこれらのシナリオのいくつかについて取り上げます。

### メンテナンス

Gitは時々 “auto gc” と呼ばれるコマンドを自動的に実行します。 大抵の場合、このコマンドは何もしません。 ですが、緩いオブジェクト（packfileの中に入っていないオブジェクト）やpackfileがあまりに多い場合は、Gitは完全な（full-fledged）`git gc` コマンドを起動します。 “gc” はガベージコレクト（garbage collect）を意味します。このコマンドは幾つものことを行います。すべての緩いオブジェクトを集めてpackfileに入れ、複数のpackfileをひとつの大きなpackfileに統合し、さらにどのコミットからも到達が不可能かつ数ヶ月間更新がないオブジェクトを削除します。

次のように手動でauto gcを実行することもできます。

```
$ git gc --auto
```

繰り返しますが、これは通常は何も行いません。 約7,000個もの緩いオブジェクトがあるか、または50以上のpackfileがある場合でないと、Gitは実際にgcコマンドを開始しません。 これらのリミットはそれぞれ設定ファイルの `gc.auto` と `gc.autopacklimit` で変更できます。

その他に `gc` が行うこととしては、複数の参照を1つのファイルにパックすることが挙げられます。 リポジトリに、次のようなブランチとタグが含まれているとしましょう。

```
$ find .git/refs -type f
.git/refs/heads/experiment
.git/refs/heads/master
.git/refs/tags/v1.0
.git/refs/tags/v1.1
```

`git gc` を実行すると、これらのファイルは `refs` ディレクトリからなくなります。 効率化のため、Gitはそれらのファイルの内容を、以下のような `.git/packed-refs` という名前のファイルに移します。

```
$ cat .git/packed-refs
# pack-refs with: peeled fully-peeled
cac0cab538b970a37ea1e769cbbde608743bc96d refs/heads/experiment
ab1afef80fac8e34258ff41fc1b867c702daa24b refs/heads/master
cac0cab538b970a37ea1e769cbbde608743bc96d refs/tags/v1.0
9585191f37f7b0fb9444f35a9bf50de191beadc2 refs/tags/v1.1
^1a410efbd13591db07496601ebc7a059dd55cfe9
```

ただ、ここで参照を更新しても、Gitはこのファイルを編集せず、その代わりに `refs/heads` に新しいファイルを書き込みます。 とある参照に対する適切なSHA-1ハッシュを得るために、Gitは `refs` ディレクトリ内でその参照をチェックした上で、見つからなかった場合の代替として `packed-refs` ファイルをチェックします。 一方、 `refs` ディレクトリ内で参照が見つけられない場合は、それはおそらく `packed-refs` ファイル内にあります。

ファイルの最後の行に注意してください。 `^` という文字で始まっています。 これは、この行のすぐ上にあるタグは注釈付き版のタグであり、この行はそのタグが指しているコミットであるということを意味しています。

## データリカバリ

Gitを使っていく過程のある時点で、誤ってコミットを失ってしまうことがあるかもしれません。 このようなことが起こりがちなのは、成果物が入っていたブランチをforce-deleteしたけれど、その後結局そのブランチが必要になったときか、あるいはブランチをhard-resetしたために、何か必要なものが入っているコミットがそのブランチから切り離されてしまったときです。 このようなことが起きたとして、どうやったらコミットを取り戻せるでしょうか？

以下に示す例では、testリポジトリ内のmasterブランチを古いコミットにhard-resetして、それから失ったコミットを復元します。 まず、今の時点でリポジトリがどのような状況にあるのか調べてみましょう。

```
$ git log --pretty=oneline
ab1afef80fac8e34258ff41fc1b867c702daa24b modified repo a bit
484a59275031909e19aadb7c92262719cfcdf19a added repo.rb
1a410efbd13591db07496601ebc7a059dd55cfe9 third commit
cac0cab538b970a37ea1e769cbbde608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

ここで、`master` ブランチを真ん中のコミットの時点まで戻します。

```
$ git reset --hard 1a410efbd13591db07496601ebc7a059dd55cfe9
HEAD is now at 1a410ef third commit
$ git log --pretty=oneline
1a410efbd13591db07496601ebc7a059dd55cfe9 third commit
cac0cab538b970a37ea1e769cbbde608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

これで、一番上にあった2つのコミットは、事実上失われたことになります。これらのコミットに辿り着けるブランチがないためです。 そのため、最後のコミットのSHA-1ハッシュを調べた上で、そこを指すブランチを追加する必要があります。 ここでポイントとなるのは、最後のコミットのSHA-1ハッシュを見つける方法です。ハッシュ値を記憶してます、なんてことはないですよね？

大抵の場合、最も手っ取り早いのは、`git reflog` というツールを使う方法です。 あなたが作業をしている間、HEADを変更する度に、HEADがどこを指しているかをGitは裏で記録しています。 コミットをしたり、ブランチを変更したりする度に、reflogは更新されます。 また、reflogは `git update-ref` コマンドによっても更新されます。refファイルに書かれたSHA-1ハッシュ値を直に編集せずに、このコマンドを使って編集すべき理由の1つがこれです（詳しくは Gitの参照 で取り上げました）。 `git reflog` を実行することで、ある時点で自分がどこにいたのかを知ることができます。

```
$ git reflog
1a410ef HEAD@{0}: reset: moving to 1a410ef
ab1afef HEAD@{1}: commit: modified repo.rb a bit
484a592 HEAD@{2}: commit: added repo.rb
```

このとおり、チェックアウトした2つのコミットが見つかりました。ですが、それ以上の情報は表示されていません。 同じ情報をもっと有用な形式で表示するには `git log -g` を実行します。これはreflogを通常のログ出力と同じ形式で出力してくれます。

```
$ git log -g
commit 1a410efbd13591db07496601ebc7a059dd55cfe9
Reflog: HEAD@{0} (Scott Chacon <schacon@gmail.com>)
Reflog message: updating HEAD
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri May 22 18:22:37 2009 -0700

        third commit

commit ab1afef80fac8e34258ff41fc1b867c702daa24b
Reflog: HEAD@{1} (Scott Chacon <schacon@gmail.com>)
Reflog message: updating HEAD
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri May 22 18:15:24 2009 -0700

       modified repo.rb a bit
```

一番下にあるコミットが、失われたコミットのようです。そこから新しいブランチを作成すれば、失ったコミットを取り戻せます。 例えば、そのコミット（ab1afef）を起点に `recover-branch` という名前のブランチを作成できます。

```
$ git branch recover-branch ab1afef
$ git log --pretty=oneline recover-branch
ab1afef80fac8e34258ff41fc1b867c702daa24b modified repo a bit
484a59275031909e19aadb7c92262719cfcdf19a added repo.rb
1a410efbd13591db07496601ebc7a059dd55cfe9 third commit
cac0cab538b970a37ea1e769cbbde608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

やった！ – `master` ブランチがかつて存在した場所に、 `recover-branch` という名前のブランチが作られて、最初の2つのコミットは再び到達可能になりました。 さて次は、失われたコミットが何らかの理由でreflogの中にもなかった場合を考えましょう – `recover-branch` を取り除き、reflogを削除することによって、擬似的にその状況を作り出すことができます。 これで、最初の2つのコミットは、今どこからも到達不能になりました。

```
$ git branch -D recover-branch
$ rm -Rf .git/logs/
```

reflogのデータは `.git/logs/` ディレクトリに保存されるため、これでreflogは事実上なくなりました。 この時点で、どうしたら失われたコミットを復元できるでしょうか? ひとつの方法として、 `git fsck` ユーティリティーを使用してデータベースの完全性をチェックする方法があります。 `--full` オプションを付けて実行すると、他のどのオブジェクトからも指されていないオブジェクトをすべて表示します。

```
$ git fsck --full
Checking object directories: 100% (256/256), done.
Checking objects: 100% (18/18), done.
dangling blob d670460b4b4aece5915caf5c68d12f560a9fe3e4
dangling commit ab1afef80fac8e34258ff41fc1b867c702daa24b
dangling tree aea790b9a58f6cf6f2804eeac9f0abbe9631e4c9
dangling blob 7108f7ecb345ee9d0084193f147cdad4d2998293
```

このケースでは、“dangling commit” という文字列の後に失われたコミットが表示されています。 前と同様にこのSHA-1ハッシュを指すブランチを作成すれば、失われたコミットを取り戻せます。

## オブジェクトの削除

Gitには素晴らしい点がたくさんありますが、問題となり得る特徴がひとつあります。それは、 `git clone` がすべてのファイルのすべてのバージョンを含んだプロジェクトの歴史全体をダウンロードしてしまうということです。 保存されているのがソースコードだけなら、特に問題はありません。なぜなら、Gitはそのようなデータを効率良く圧縮することに高度に最適化されているからです。 しかし、もし誰かがある時点でプロジェクトの歴史に非常に大きなファイルを1つ加えると、以降のクローンではすべて、その大きなファイルのダウンロードを強いられることになります。これは、直後のコミットでそのファイルをプロジェクトから削除したとしても変わりません。 なぜなら、そのファイルは履歴から到達可能であり、常にそこに存在し続けるためです。

SubversionやPerforceのリポジトリをGitに変換するときに、これは大きな問題になり得ます。 なぜなら、それらのシステムではすべての履歴をダウンロードする必要がないため、非常に大きなファイルを追加してもほとんど悪影響がないからです。 別のシステムからリポジトリをインポートした場合や、リポジトリがあるべき状態よりもずっと大きくなっている場合に、大きなオブジェクトを見つけて取り除く方法を以下に示します。

**注意: この操作はコミット履歴を破壊的に変更します。** この操作では、大きなファイルへの参照を取り除くため、修正が必要な一番古いツリーから、以降すべてのコミットオブジェクトを再書き込みします。 インポートの直後、そのコミットをベースとして誰かが作業を始める前にこの操作を行った場合は問題ありません。そうでない場合は、作業中の内容を新しいコミットにリベースしなければならないことを、すべての関係者に知らせる必要があります。

実演のため、testリポジトリに大きなファイルを追加して、次のコミットでそれを取り除いた上で、リポジトリからそのファイルを探し出し、そしてリポジトリからそれを完全に削除します。 まず、あなたの歴史に大きなオブジェクトを追加します。

```
$ curl https://www.kernel.org/pub/software/scm/git/git-2.1.0.tar.gz >
git.tgz
$ git add git.tgz
$ git commit -m 'add git tarball'
[master 7b30847] add git tarball
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 git.tgz
```

おっと、誤ってプロジェクトに非常に大きなtarボールを追加してしまいました。取り除いたほうがいいでしょう。

```
$ git rm git.tgz
rm 'git.tgz'
$ git commit -m 'oops - removed large tarball'
[master dadf725] oops - removed large tarball
 1 file changed, 0 insertions(+), 0 deletions(-)
 delete mode 100644 git.tgz
```

次に、データベースに対して `gc` を実行します。その後、どれくらいのスペースを使用しているのかを見てみましょう。

```
$ git gc
Counting objects: 17, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (13/13), done.
Writing objects: 100% (17/17), done.
Total 17 (delta 1), reused 10 (delta 0)
```

`count-objects` コマンドを実行すると、どれくらいのスペースを使用しているのかをすぐに見ることができます。

```
$ git count-objects -v
count: 7
size: 32
in-pack: 17
packs: 1
size-pack: 4868
prune-packable: 0
garbage: 0
size-garbage: 0
```

`size-pack` エントリにはpackfileのサイズがキロバイト単位で表示されていて、約5MB使用していることがわかります。 大きなファイルを追加するコミットの前に使用していたのは、2KB程度でした – 明らかに、直近のコミットで行ったファイルの削除では、歴史からファイルが削除されていません。 誤って大きなファイルを追加してしまったがために、誰かがこのリポジトリをクローンするたび、この小さなプロジェクトを取得するだけのために5MBすべてをクローンしなければならなくなってしまいました。 この大きなファイルを削除しましょう。

最初に、その大きなファイルを見つけなければなりません。 この例では、どのファイルがそれかは既に分かっています。 しかし、それが分からない場合、どうやって多くのスペースを占めているファイルを特定するのでしょうか？ `git gc` を実行すると、すべてのオブジェクトがpackfileに格納されます。 そのため、別の配管コマンド `git verify-pack` を実行し、その出力を3つ目のフィールド（ファイルサイズ）でソートすれば、大きなオブジェクトを特定できます。 関心の対象になるのは最も大きなファイル数個だけなので、その

出力をパイプで tail コマンドに通してもよいでしょう。

```
$ git verify-pack -v .git/objects/pack/pack-29…69.idx \
  | sort -k 3 -n \
  | tail -3
dadf7258d699da2c8d89b09ef6670edb7d5f91b4 commit 229 159 12
033b4468fa6b2a9547a70d88d1bbe8bf3f9ed0d5 blob   22044 5792 4977696
82c99a3e86bb1267b236a4b6eff7868d97489af1 blob   4975916 4976258 1438
```

探していた大きなオブジェクトは、一番下の5MBのものです。 そのオブジェクトが何のファイルなのかを知るには 特定のコミットメッセージ書式の強制 で少し使用した rev-list コマンドを使用します。 --objects を rev-list に渡すと、すべてのコミットのSHA-1ハッシュに加えて、すべてのブロブのSHA-1ハッシュと、そのブロブに関連付けられたファイルのパスを一覧表示します。 これは、ブロブの名前を特定するのに使えます。

```
$ git rev-list --objects --all | grep 82c99a3
82c99a3e86bb1267b236a4b6eff7868d97489af1 git.tgz
```

次に、過去のすべてのツリーからこのファイルを削除する必要があります。 このファイルを変更したのがどのコミットかは簡単に分かります。

```
$ git log --oneline --branches -- git.tgz
dadf725 oops - removed large tarball
7b30847 add git tarball
```

Gitリポジトリからこのファイルを完全に削除するには、 7b30847 の下流にあるすべてのコミットを修正しなければなりません。 そのためには、 歴史の書き換え で使用した filter-branch を使用します。

```
$ git filter-branch --index-filter \
  'git rm --ignore-unmatch --cached git.tgz' -- 7b30847^..
Rewrite 7b30847d080183a1ab7d18fb202473b3096e9f34 (1/2)rm 'git.tgz'
Rewrite dadf7258d699da2c8d89b09ef6670edb7d5f91b4 (2/2)
Ref 'refs/heads/master' was rewritten
```

--index-filter オプションは 歴史の書き換え で使用した --tree-filter オプションに似ていますが、ディスク上のチェックアウトされたファイルを変更するコマンドを渡すのではなく、コミット毎にステージングエリアまたはインデックスを変更する点が異なります。

ここでは、あるファイルを rm file で削除するのではなく、 git rm --cached で削除する必要があります。つまり、ディスクではなくインデックスからファイルを削除しなければなりません。 このようにする理由はスピードです。この場合、Gitがフィルタを実行する前に各リビジョンをディスク上へチェックアウトする必要がないので、プロセスをもっともっと速くすることができます。 お望みなら、同様のタスクは --tree-filter でも行えます。 git rm に渡している --ignore-unmatch オプションは、削除しようとするパ

ターンに合うファイルがない場合に、エラーを出力しないようにします。 最後に、filter-branch に、コミット 7b30847 以降の履歴のみを修正するように伝えています。なぜなら、問題が発生した場所がここだと分かっているからです。 そうでない場合は、歴史の先頭から処理を開始することになり、不必要に長い時間がかかるでしょう。

これで、歴史から大きなファイルへの参照がなくなりました。 しかし、 .git/refs/original の下で filter-branch を行ったときにGitが新しく追加したrefsには、まだ参照が含まれています。reflogについても同様です。それらを削除した上で、データベースを再パックしなければなりません。 再パックの前に、それら古いコミットへのポインタを持つものをすべて削除する必要があります。

```
$ rm -Rf .git/refs/original
$ rm -Rf .git/logs/
$ git gc
Counting objects: 15, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (11/11), done.
Writing objects: 100% (15/15), done.
Total 15 (delta 1), reused 12 (delta 0)
```

どれくらいのスペースが節約されたかを見てみましょう。

```
$ git count-objects -v
count: 11
size: 4904
in-pack: 15
packs: 1
size-pack: 8
prune-packable: 0
garbage: 0
size-garbage: 0
```

パックされたリポジトリのサイズは8KBに下がり、当初の5MBよりもずっとよくなりました。 サイズの値を見ると、緩いオブジェクトの中には大きなオブジェクトが残っており、無くなったわけではないことが分かります。ですが、プッシュや以降のクローンで転送されることはもうありません。ここが重要な点です。 お望みなら、 git prune に --expire オプションを指定すれば、オブジェクトを完全に削除することもできます。

```
$ git prune --expire now
$ git count-objects -v
count: 0
size: 0
in-pack: 15
packs: 1
size-pack: 8
prune-packable: 0
garbage: 0
size-garbage: 0
```

# 環境変数

Gitは常に bash シェル内で実行されます。また、それがどのように動作するかを決定するために、多くのシェル環境変数を使用しています。 場合によっては、これらの環境変数が何であるか、Gitを望み通りに動かすためにどんなふうに使用できるかを知っていた方が便利です。 このリストは、Gitが参照する環境変数すべてを網羅してはいませんが、もっとも有用なものを取り上げています。

## グローバルな振る舞い

Gitのコンピュータプログラムとしての一般的な動作の中には、環境変数に依存しているものがいくつかあります。

**GIT_EXEC_PATH** は、Gitがサブプログラム（git-commit や git-diff など）を探すディレクトリを決定します。 git --exec-path で現在の設定を確認できます。

**HOME** は通常カスタマイズされることを考慮されてはいません（他にこの変数に依存しているものがありすぎる）が、Gitはこのディレクトリからグローバル設定ファイルを探します。 Gitを完全にポータブルな状態でインストールしたいなら、グローバル設定をすべて埋めた上で、ポータブルなGitのシェルプロファイルで HOME を上書きできます。

**PREFIX** もそれと似ていますが、こちらはシステム全体の設定です。 Gitはこのファイルを $PREFIX/etc/gitconfig から探します。

**GIT_CONFIG_NOSYSTEM** を設定すると、システム全体の設定ファイルを無効にします。 これは、コマンドの実行にあたってシステム全体の設定が邪魔になるが、それを変更したり削除したりするアクセス権がない場合に便利です。

**GIT_PAGER** はコマンドラインで複数ページにわたる出力を表示する際に使用されるプログラムを制御します。 設定されていない場合、PAGER が代わりに使用されます。

**GIT_EDITOR** はテキスト（例えばコミットメッセージ）を編集する必要があるときにGitから起動されるエディタです。 設定されていない場合、 EDITOR が代わりに使用されます。

### リポジトリの場所

Gitは、Gitと現在のリポジトリとのインタフェース方法を決定するのに、いくつかの環境変数を使用します。

**GIT_DIR** は `.git` フォルダの場所です。 指定されていない場合、Gitはディレクトリツリーを `~` または `/` にたどり着くまで上っていき、各ディレクトリで `.git` ディレクトリを探します。

**GIT_CEILING_DIRECTORIES** は `.git` ディレクトリを探す際の動作を制御します。 読み込みが遅いディレクトリにアクセスしている場合（例えばテープドライブ上のディレクトリや、低速なネットワーク越しにアクセスしている場合）、Gitが自動で停止するのを待たずに試行を停止させたくなることもあると思います。特に、シェルプロンプトを構成している最中にGitが呼ばれた場合はそうでしょう。

**GIT_WORK_TREE** は、ベアリポジトリ以外のリポジトリで、ワーキングディレクトリのルートとなる場所です。 指定されていない場合、 `$GIT_DIR` の親ディレクトリが代わりに使用されます。

**GIT_INDEX_FILE** は、インデックスファイルのパスです（ベアリポジトリ以外でのみ使用されます）。

**GIT_OBJECT_DIRECTORY** は、通常 `.git/objects` にあるディレクトリの場所を指定するのに使用できます。

**GIT_ALTERNATE_OBJECT_DIRECTORIES** は、 `GIT_OBJECT_DIRECTORY` にオブジェクトがなかった場合にチェックに行く場所を指示するのに使います。コロン区切りのリスト（ `/dir/one:/dir/two:…` のような書式）で指定します。 大量のプロジェクトに、全く同じ内容の巨大なファイルがあるという状況で、そのファイルを大量に重複して保存したくない場合に、これが利用できます。

### Pathspec

“pathspec” とは、Gitに何かのパスを指定する方法のことで、ワイルドカードの使用法などが含まれます。 以下の環境変数は `.gitignore` ファイルだけでなく、コマンドライン（ `git add *.c` など）でも使用されます。

**GIT_GLOB_PATHSPECS および GIT_NOGLOB_PATHSPECS** は、pathspec において、ワイルドカードのデフォルトの動作を制御します。 `GIT_GLOB_PATHSPECS` に1がセットされている場合、ワイルドカード文字はワイルドカードとして働きます（これがデフォルトの挙動）。 `GIT_NOGLOB_PATHSPECS` に1がセットされている場合、ワイルドカード文字はそれ自身にのみマッチ、つまり `*.c` は `.c` で終わる名前のファイルすべてではなく、 “*.c” *という名前の* ファイルにのみマッチします。 pathspec に `:(glob)` や `:(literal)` を、 `:(glob)*.c` のように指定することで、個々のケースに対してより優先的な設定を行うこともできます。

**GIT_LITERAL_PATHSPECS** は上記の振る舞いを両方とも無効にします。ワイルドカード文字は機能を停止し、オーバーライド接頭辞も無効化されます。

**GIT_ICASE_PATHSPECS** はすべての pathspec が大文字小文字を区別せず処理するように設定します。

## コミット

Gitのコミットオブジェクトは通常、最終的に `git-commit-tree` によって作成されます。このコマンドは、以下の環境変数に設定されている情報を優先的に使用します。これらの環境変数が存在しない場合にのみ、設定ファイルの値が代わりに使用されます。

**GIT_AUTHOR_NAME** は “author” フィールドに使用される、人間に読める形式の名前です。

**GIT_AUTHOR_EMAIL** は “author” フィールドで使用するメールアドレスです。

**GIT_AUTHOR_DATE** は “author” フィールドで使用するタイムスタンプです。

**GIT_COMMITTER_NAME** は “committer” フィールドで使用する人名です。

**GIT_COMMITTER_EMAIL** は “committer” フィールドで使用するメールアドレスです。

**GIT_COMMITTER_DATE** は “committer” フィールドで使用するタイムスタンプです。

**EMAIL** は、設定値 `user.email` が設定されていない場合に代わりに使用されるメールアドレスです。 *この環境変数自体が* 設定されていない場合、Gitはシステムのユーザ名とホスト名を代わりに使用します。

## ネットワーク

Git はHTTP越しのネットワーク操作に `curl` ライブラリを使用しています。そのため、 **GIT_CURL_VERBOSE** はそのライブラリが生成するメッセージをすべて出力するようGitに指示します。 これはコマンドラインで `curl -v` を実行するのと似たようなものです。

**GIT_SSL_NO_VERIFY** は、SSL証明書の検証を行わないようにGitへ指示します。 これは、Gitリポジトリ をHTTPS経由で利用するために自己署名証明書を使っている場合や、Gitサーバーのセットアップ中で正式な 証明書のインストールが完了していない場合などに必要になります。

あるHTTP操作のデータレートが秒間 **GIT_HTTP_LOW_SPEED_LIMIT** バイトより低い状態が、 **GIT_HTTP_LOW_SPEED_TIME** 秒より長く続いた場合、Gitはその操作を中断します。 これらの環境変数は設定ファイルの `http.lowSpeedLimit` および `http.lowSpeedTime` の値より優先されます。

**GIT_HTTP_USER_AGENT** はGitがHTTPで通信する際のuser-agent文字列を設定します。 デフォルトの値は `git/2.0.0` のような内容です。

## 差分取得とマージ

**GIT_DIFF_OPTS** ですが、これは名前の付け方に少し問題ありと言えます。 有効な値は `-u<n>` または `--unified=<n>` だけです。これは、 `git diff` コマンドで表示されるコンテキスト行の行数を制御します。

**GIT_EXTERNAL_DIFF** は設定ファイルの `diff.external` の値をオーバーライドします。 設定されている場合、ここで指定したプログラムが `git diff` の実行時に呼び出されます。

**GIT_DIFF_PATH_COUNTER** および **GIT_DIFF_PATH_TOTAL** は、 `GIT_EXTERNAL_DIFF` または `diff.external` で指定したプログラムの内部で使用すると便利です。 前者は、処理中の一連のファイルの中で何番目のファイルの差分を処理しているか（1から始まる数値）、後者は処理中の一連のファイルの総数です。

- `GIT_MERGE_VERBOSITY` * は、再帰的なマージ戦略の出力を制御します。 指定できる値は以下の通りです。

- 0 は何も出力しません。例外として、エラーがあった場合はエラーメッセージを1つだけ出力します。
- 1 はマージコンフリクトのみ表示します。
- 2 はファイルの変更点のみ表示します。
- 3 は変更がなく処理をスキップしたファイルを表示します。
- 4 は処理されたパスをすべて表示します。
- 5 以上を指定すると、上記のすべてに加えて詳細なデバッグ用の情報を表示します。

デフォルト値は2です。

## デバッグ

Gitが何をしているか、*本当に* 知りたいですか？ Gitには、組み込みのトレースのほぼ完全なセットが備わっており、ユーザがする必要があるのは、それらをオンにすることだけです。 これらの環境変数に設定可能な値は次の通りです。

- “true”、“1”、“2” – 対象のカテゴリのトレースは標準エラー出力へ書き出されます。
- / から始まる絶対パス文字列 – 対象のトレースの出力はそのファイルへ書き出されます。

**GIT_TRACE** は、どの特定のカテゴリにも当てはまらない、一般的なトレースを制御します。 これには、エイリアスの展開や、他のサブプログラムへの処理の引き渡しなどが含まれます。

```
$ GIT_TRACE=true git lga
20:12:49.877982 git.c:554               trace: exec: 'git-lga'
20:12:49.878369 run-command.c:341       trace: run_command: 'git-lga'
20:12:49.879529 git.c:282               trace: alias expansion: lga =>
'log' '--graph' '--pretty=oneline' '--abbrev-commit' '--decorate' '--all'
20:12:49.879885 git.c:349               trace: built-in: git 'log' '--
graph' '--pretty=oneline' '--abbrev-commit' '--decorate' '--all'
20:12:49.899217 run-command.c:341       trace: run_command: 'less'
20:12:49.899675 run-command.c:192       trace: exec: 'less'
```

**GIT_TRACE_PACK_ACCESS** はパックファイルへのアクセスに関するトレースを制御します。 最初のフィールドはアクセスされているパックファイル、次のフィールドはそのファイル内でのオフセットです。

```
$ GIT_TRACE_PACK_ACCESS=true git status
20:10:12.081397 sha1_file.c:2088        .git/objects/pack/pack-
c3fa...291e.pack 12
20:10:12.081886 sha1_file.c:2088        .git/objects/pack/pack-
c3fa...291e.pack 34662
20:10:12.082115 sha1_file.c:2088        .git/objects/pack/pack-
c3fa...291e.pack 35175
# […]
20:10:12.087398 sha1_file.c:2088        .git/objects/pack/pack-
e80e...e3d2.pack 56914983
20:10:12.087419 sha1_file.c:2088        .git/objects/pack/pack-
e80e...e3d2.pack 14303666
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working directory clean
```

**GIT_TRACE_PACKET** はネットワーク操作におけるパケットレベルのトレースを有効にします。

```
$ GIT_TRACE_PACKET=true git ls-remote origin
20:15:14.867043 pkt-line.c:46           packet:          git< #
service=git-upload-pack
20:15:14.867071 pkt-line.c:46           packet:          git< 0000
20:15:14.867079 pkt-line.c:46           packet:          git<
97b8860c071898d9e162678ea1035a8ced2f8b1f HEAD\0multi_ack thin-pack side-
band side-band-64k ofs-delta shallow no-progress include-tag
multi_ack_detailed no-done symref=HEAD:refs/heads/master agent=git/2.0.4
20:15:14.867088 pkt-line.c:46           packet:          git<
0f20ae29889d61f2e93ae00fd34f1cdb53285702 refs/heads/ab/add-interactive-
show-diff-func-name
20:15:14.867094 pkt-line.c:46           packet:          git<
36dc827bc9d17f80ed4f326de21247a5d1341fbc refs/heads/ah/doc-gitk-config
# […]
```

**GIT_TRACE_PERFORMANCE** は性能データのログ出力を制御します。 ログには、一つ一つのGit呼び出しにかかった時間が出力されます。

```
$ GIT_TRACE_PERFORMANCE=true git gc
20:18:19.499676 trace.c:414             performance: 0.374835000 s: git
command: 'git' 'pack-refs' '--all' '--prune'
20:18:19.845585 trace.c:414             performance: 0.343020000 s: git
command: 'git' 'reflog' 'expire' '--all'
Counting objects: 170994, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (43413/43413), done.
Writing objects: 100% (170994/170994), done.
Total 170994 (delta 126176), reused 170524 (delta 125706)
20:18:23.567927 trace.c:414             performance: 3.715349000 s: git
command: 'git' 'pack-objects' '--keep-true-parents' '--honor-pack-keep'
'--non-empty' '--all' '--reflog' '--unpack-unreachable=2.weeks.ago' '--
local' '--delta-base-offset' '.git/objects/pack/.tmp-49190-pack'
20:18:23.584728 trace.c:414             performance: 0.000910000 s: git
command: 'git' 'prune-packed'
20:18:23.605218 trace.c:414             performance: 0.017972000 s: git
command: 'git' 'update-server-info'
20:18:23.606342 trace.c:414             performance: 3.756312000 s: git
command: 'git' 'repack' '-d' '-l' '-A' '--unpack-unreachable=2.weeks.ago'
Checking connectivity: 170994, done.
20:18:25.225424 trace.c:414             performance: 1.616423000 s: git
command: 'git' 'prune' '--expire' '2.weeks.ago'
20:18:25.232403 trace.c:414             performance: 0.001051000 s: git
command: 'git' 'rerere' 'gc'
20:18:25.233159 trace.c:414             performance: 6.112217000 s: git
command: 'git' 'gc'
```

**GIT_TRACE_SETUP** はGitがリポジトリや環境を操作する際に何を見ているかを表示します。

```
$ GIT_TRACE_SETUP=true git status
20:19:47.086765 trace.c:315             setup: git_dir: .git
20:19:47.087184 trace.c:316             setup: worktree:
/Users/ben/src/git
20:19:47.087191 trace.c:317             setup: cwd: /Users/ben/src/git
20:19:47.087194 trace.c:318             setup: prefix: (null)
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working directory clean
```

## その他

**GIT_SSH** は、指定されている場合、GitがSSHホストへ接続する際に `ssh` の代わりに呼び出されるプログラムです。 これは `$GIT_SSH [username@]host [-p <port>] <command>` のように呼び出されます。 注意事項として、これは `ssh` の呼び出し方をカスタマイズする最も手軽な方法というわけではありません。追加のコマンドラインパラメタがサポートされないので、ラッパースクリプトを書いて、 `GIT_SSH` がそのスクリ

プトを指すようにする必要があります。 その場合は単に `~/.ssh/config` ファイルを使用する方が簡単でしょう。

**GIT_ASKPASS** は設定ファイルの `core.askpass` の値をオーバーライドします。 これはユーザによる認証情報の入力が必要なときに呼び出されるプログラムで、コマンドライン引数としてプロンプトのテキストを受け取り、応答を標準出力へ返すようになっている必要があります。 （このサブシステムの詳細については 認証情報の保存 を参照してください）

**GIT_NAMESPACE** は名前空間内の参照へのアクセス制御を行います。これは `--namespace` フラグと同様です。 これがもっとも便利なのは、サーバーで一つのリポジトリの複数のフォークを単一のリポジトリへ格納したいが、参照だけは別々に分けておきたい場合です。

**GIT_FLUSH** は、Gitに非バッファI/Oを使用するように強制します。標準出力への書き出しを逐次的に行いたい場合に使用します。 1を設定すると、Gitは出力をより頻繁にフラッシュします。0を設定すると、すべての出力がバッファリングされます。 デフォルト（この環境変数が設定されていない場合）では、動作と出力モードに合わせて適切なバッファリングスキームが選択されます。

**GIT_REFLOG_ACTION** では、reflogへ出力される説明用のテキストを指定できます。 次に例を示します。

```
$ GIT_REFLOG_ACTION="my action" git commit --allow-empty -m 'my message'
[master 9e3d55a] my message
$ git reflog -1
9e3d55a HEAD@{0}: my action: my message
```

# まとめ

Git がバックグラウンドで行っている処理について、とてもよく理解できたことと思います。また、その実装の方法についても、ある程度の理解が得られたと思います。 この章では、各種の配管コマンド – 本書の他の章で学んだ磁器コマンドよりも、低レベルでシンプルなコマンド – を何度も使いました。 Gitがどのように機能しているのかを、より低いレベルで理解すれば、なぜそのようなことを行うのかも容易に理解できるようになるでしょうし、自分のワークフローがうまく機能するように自前のツールや補助スクリプトを書くのもより楽になるはずです。

内容アドレスファイルシステムとしてのGitは、とても強力なツールで、単なるバージョン管理システム以上のものとして使うことも簡単にできます。 この技術を使って自前の素晴らしいアプリケーションを実装したり、Gitの進んだ使い方をより気楽に利用したりするのに、この章で得たGitの内部に関する知識が役立つことを願っています。

# Appendix A: その他の環境でのGit

これまでの内容を通して、コマンドラインからGitを使用する方法について多くのことを学んできました。ローカルファイルに対して作業したり、ネットワークを介して他のリポジトリへ接続したり、他の人と効率的に作業したりできるようになったと思います。 しかし、話はそこで終わりません。Gitは通常、より大きなエコシステムの一部として使用されます。端末からの利用が常に最適解というわけではありません。 ここでは、端末以外でGitを活用できる環境の例や、そこで他の（あるいは、あなたの）アプリケーションがどのようにGitと協調動作するかを見ていきます。

## グラフィカルインタフェース

Gitは端末をネイティブ環境としています。 Gitの新機能は、まずコマンドラインから利用可能になります。また、Gitのパワーの全てを完全に思い通りに使えるのもコマンドラインからだけです。 しかし、すべてのタスクにおいてプレーンテキストが最良の選択というわけではありません。時には視覚的な表現も必要でしょうし、ポイント＆クリック方式のインタフェースの方が好みというユーザもいるでしょう。

なお、それぞれのインタフェースはそれぞれ別のワークフローに合わせて調整されているということには注意が必要です。 クライアントによっては、クライアントの作者が効率的だと考えている特定の作業手順をサポートするため、Gitの機能の中から選び抜かれた一部の機能だけを利用可能としている場合もあります。 この観点から見た場合、あるツールが他のツールと比べて “よい” よいうことはありません。各ツールは自身が想定している目的により適合している、というだけです。 また、これらグラフィカルクライアントでは可能で、コマンドラインクライアントでは不可能な処理、というものはありませんので注意してください。リポジトリに対して作業をする場合、コマンドラインが最もパワフルであることに変わりはありません。

### `gitk` と `git-gui`

Gitをインストールすると、ビジュアルツール `gitk` および `git-gui` が使えるようになります。

`gitk` はグラフィカルな歴史ビューアです。 `git log` や `git grep` をパワフルなGUIシェルから使えるようにしたようなものだと思ってください。 これは、過去に何が起こったかを検索したり、プロジェクトの歴史を視覚化しようとしているときに使うツールです。

Gitkはコマンドラインから呼び出すのが一番簡単です。 Gitのリポジトリに `cd` して、以下のようにタイプしてください。

```
$ gitk [git logのオプション]
```

Gitkには多くのコマンドラインオプションがありますが、その多くはGitkの背後にいる `git log` アクションに渡されます。 おそらく、最も便利なオプションの一つは `--all` フラグでしょう。これはgitkに対し、HEADだけではなく *任意の* 参照から到達可能なコミットを表示させるものです。 Gitkのインターフェイスは次のようになっています。

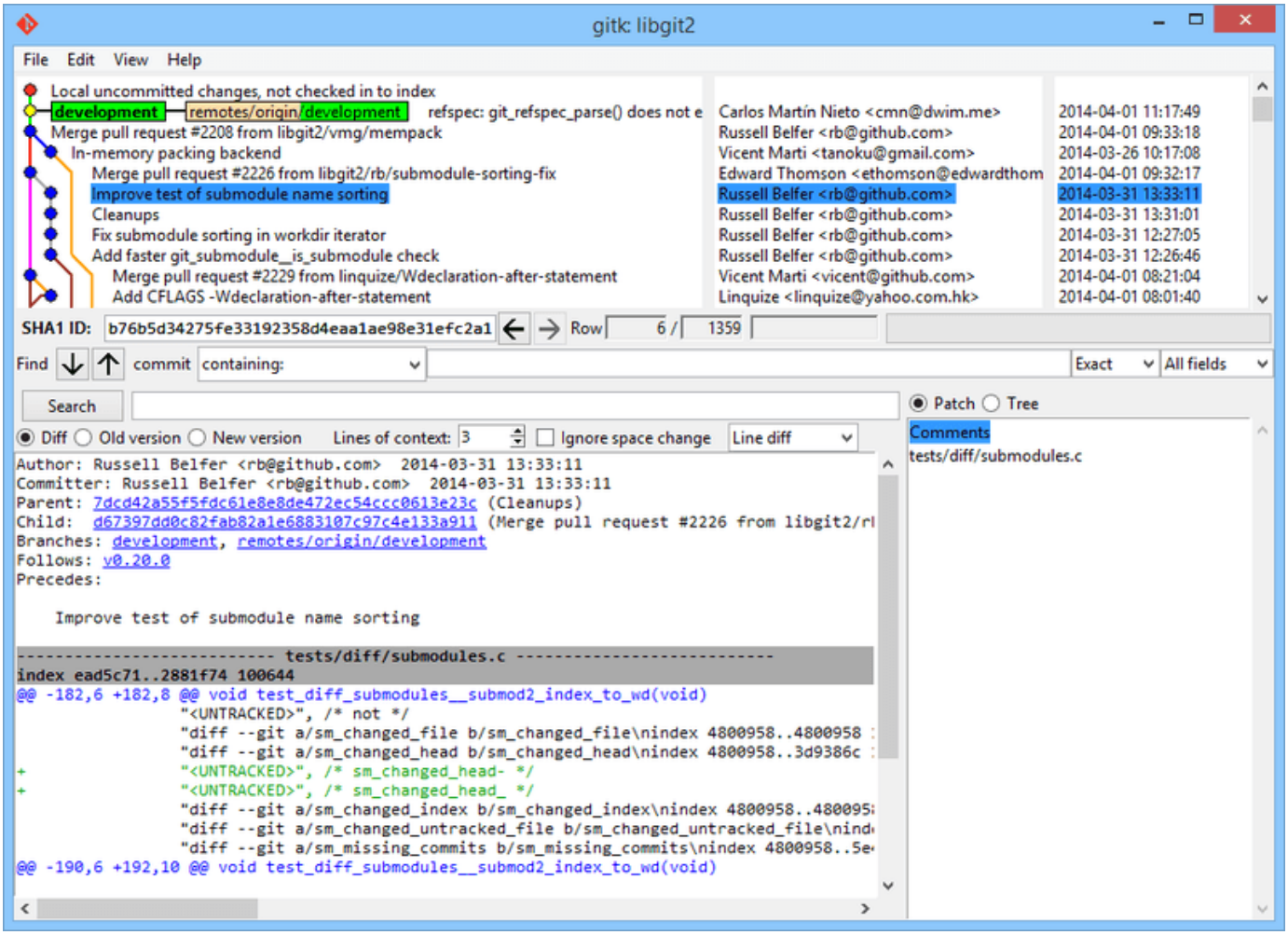



*Figure 152. `gitk`の歴史ビューアー*

上部には `git log --graph` の出力に似た内容が表示されます。各ドットはコミットを、線は親子関係を表しており、参照は色付きの箱として表示されます。 黄色の点はHEADを、赤い点はまだコミットになっていない変更を表しています。 下部には、選択されているコミットの内容が表示されます。コメントやパッチが左側に、概要が右側に表示されます。 真ん中にあるのは歴史の検索に使用するコントロール類です。

一方、 `git-gui` は主にコミットを作成するためのツールです。 これも、コマンドラインから起動するのが最も簡単です。

```
$ git gui
```

表示内容は次のようになっています。

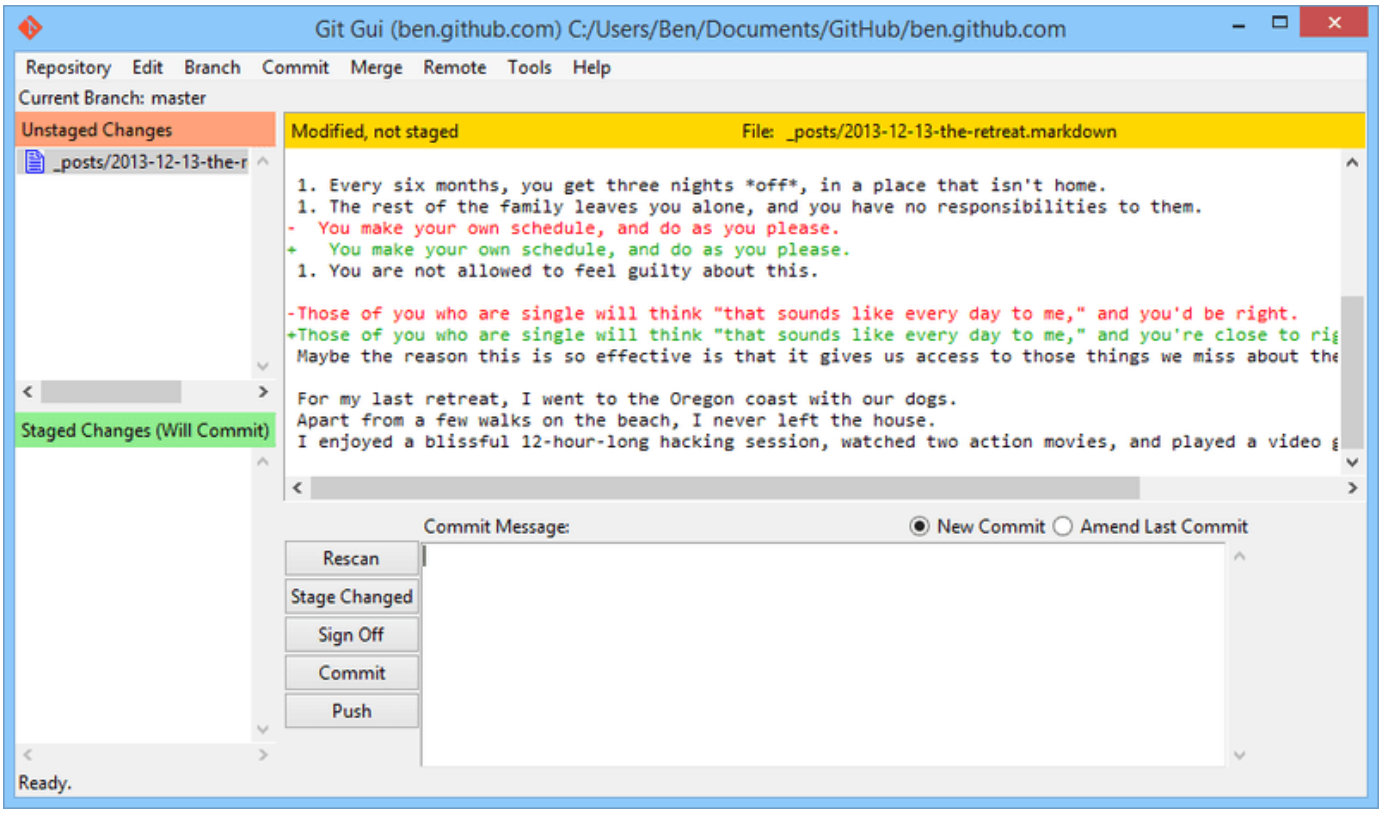



*Figure 153. `git-gui` のコミットツール*

左側はインデックスです。ステージされていない変更が上に、ステージされた変更が下に表示されます。 アイコンをクリックすると、各ファイルを2つの状態の間で行き来したり、ファイル名をクリックすることで表示するファイルを選択したりできます。

右上に表示されているのは diff で、現在選択されているファイルの変化を示しています。 この領域を右クリックすることで、個々のhunk（または個々の行）をステージングできます。

右下はメッセージとアクションの領域です。テキストボックスにメッセージを入力し、“Commit” をクリックすれば、 `git commit` と同じようなことができます。 また、 “Amend” ラジオボタンを選択すると、“Staged Changes” 領域に最後のコミットの内容が表示されるので、そのコミットを修正することもできます。 変更をステージしたり、ステージを取り消したり、コミットメッセージを変更したりしたら、“Commit” を再度クリックすれば古いコミットが新しい内容で更新されます。

`gitk` と `git-gui` はタスク指向のツールの例です。 特定の目的（それぞれ、履歴の表示と、コミットの作成）に合わせて調整されており、そのタスクに不要な機能は省略されています。

## MacとWindows用のGitHubクライアント

GitHubは、ワークフロー指向のGitクライアントを公開しています。Windows用クライアントと、Mac用クライアントがあります。 これらのクライアントは、ワークフロー指向のツールの良い例です。Gitの機能の *すべて* を公開するのではなく、よく使われる機能の中から一緒に使うと便利な機能を選択し、それにフォーカスしています。 表示内容は次のようになっています。

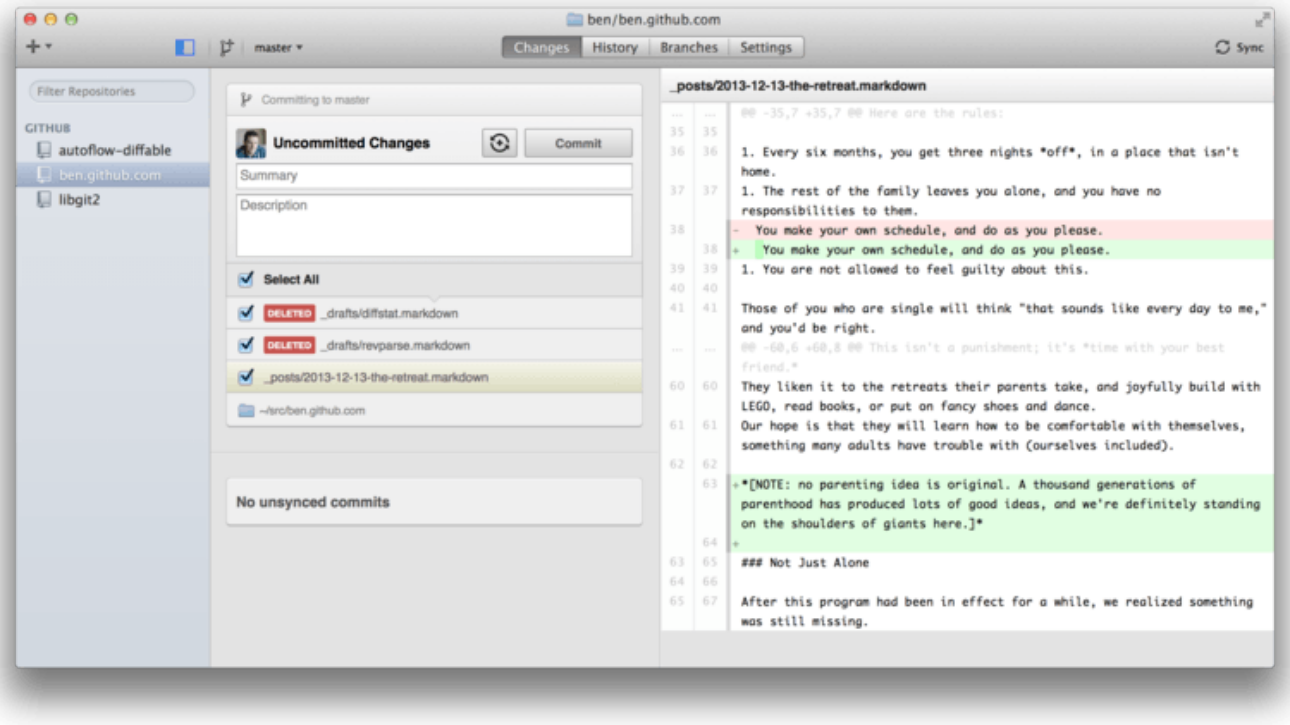



*Figure 154. GitHubのMac用クライアント*

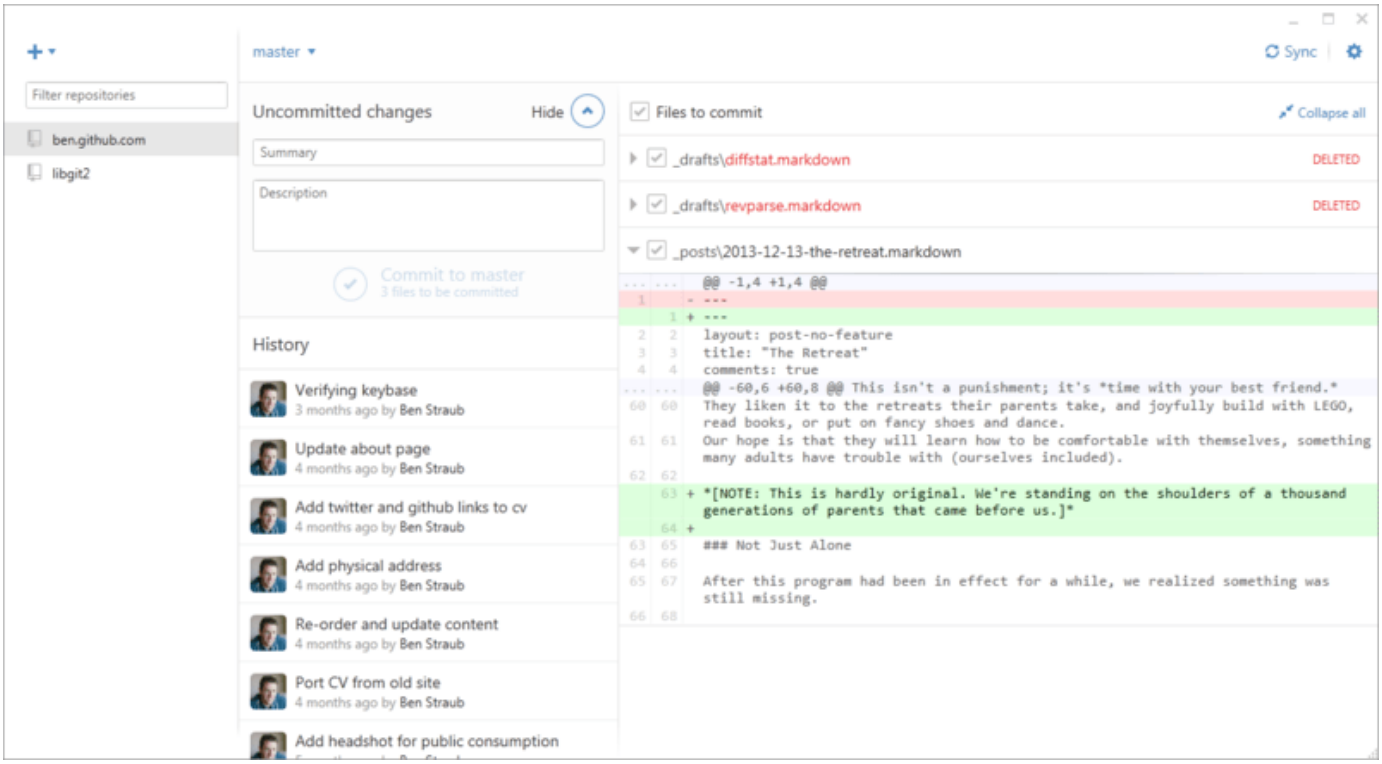



*Figure 155. GitHubのWindows用クライアント*

この2つは見た目や動作が似たような感じで設計されているので、この章ではひとつの製品として扱うことにします。 ここではツールの詳細な説明はしません（GitHubクライアントの自前のドキュメントがあります）が、“changes” ビュー（ツールの実行時間の大半はここを使うことになると思います）の内容をざっと見ていきましょう。

- 左側には、クライアントが追跡しているリポジトリのリストが表示されます。この領域の一番上の ‘`+’’ アイコンをクリックすると、（ローカルでクローンするかアタッチするかして）リポジトリを追加できます。
- 真ん中はコミット入力領域です。コミットメッセージを入力したり、コミットに含めるファイルを選択したりできます。 （Windowsでは、コミットの歴史は、この下に直接表示されます。Macの場合は、別のタブに表示されます。）
- 右側はdiffビューです。作業ディレクトリの変更内容、または、選択しているコミットに含まれている内容が表示されます。
- 最後に、右上の “Sync” ボタンは、ネットワーク経由で対話を行う主要な手段です。

**NOTE** これらのツールの使用にあたり、GitHubのアカウントは必要ありません。 これらのツールはGitHubのサービスや推奨ワークフローをハイライトするために設計されたものですが、どんなリポジトリに対しても正しく動作しますし、どんなGitのホストに対してもネットワーク操作が行えます。

### インストール

Windows用のGitHubクライアントは https://windows.github.com から、Mac用のGitHubクライアントは https://mac.github.com からダウンロードできます。 クライアントを初めて実行する際には、名前やメールアドレスの設定といったGitの初期設定がひと通り行われます。また、認証情報のキャッシュやCRLFの挙動といった、一般的なオプション設定に対して、デフォルト値が設定されます。

これらのツールはいずれも “新鮮” な状態に保たれます。つまり、アプリケーションのアップデートは、アプリケーションの実行中にバックグラウンドで自動的にダウンロードされ、インストールされます。 このアップデートには、ツールに同梱されているGitも含まれています。そのため、Gitを手動で更新する心配をする必要はおそらくないと思います。 Windowsの場合、PowerShellをPosh-gitと一緒に起動するショートカットがクライアントに同梱されています。これについてはこの章の後半で詳しく説明します。

次のステップでは、ツールに操作対象のリポジトリを設定します。 クライアントには、GitHubであなたがアクセスできるリポジトリの一覧が表示されます。クローンの作成は1ステップで行えます。 既にローカルリポジトリがある場合は、GitHubのクライアントウインドウにFinderまたはWindowsエクスプローラからそのディレクトリをドラッグすれば、クライアント左側のリポジトリのリストに追加されます。

### 推奨ワークフロー

インストールと設定が完了したら、GitHubクライアントを使って一般的なGitのタスクの多くが行えます。 このツールで想定されているワークフローは “GitHub Flow” とも呼ばれています。 この詳細は GitHub Flow で取り上げます。要点としては、（a）コミットはブランチに対して行う、（b）定期的にリモートリポジトリと同期する、といった点があります。

ブランチ管理は、2つのツールで操作が異なる点の一つです。 Mac用クライアントでは、新しいブランチを作成するためのボタンがウィンドウ上部にあります。

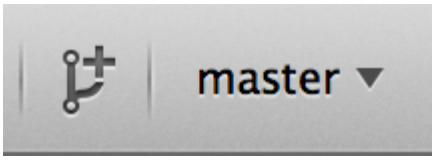



*Figure 156. Mac用クライアントの “Create Branch” ボタン*

Windows用クライアントでは、ブランチ切り替えのウィジェットで新しいブランチ名を入力すると、新しいブランチが作成されます。

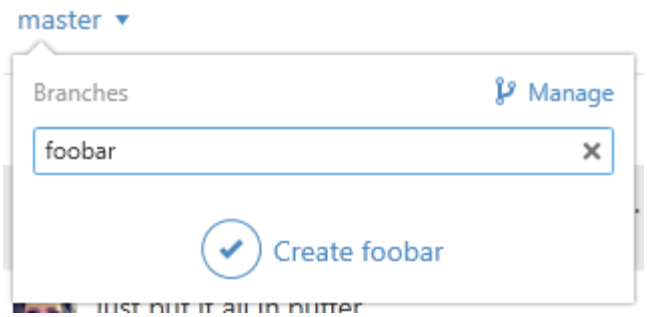



*Figure 157. Windows用クライアントでブランチを作成する*

ブランチの作成が終われば、コミットの新規作成は非常に簡単です。 作業ディレクトリで変更を行った後、GitHubクライアントのウィンドウに切り替えると、どのファイルが変更されたか表示されます。 コミットに含めたいファイルを選択し、コミットメッセージを入力したら、 “Commit” ボタンをクリックします（またはCtrl-Enterか、⌘-Enterを押下します）。

ネットワーク経由で他のリポジトリとの対話するには、主に “Sync” 機能を使用します。 Gitは内部的に、プッシュ、フェッチ、マージ、およびリベースを別々の操作としていますが、GitHubクライアントではこれら一連の処理を1つの機能で実行できるようになっています。 “Sync” ボタンをクリックすると以下の処理が実行されます。

1. `git pull --rebase` を実行します。 マージが衝突してこのコマンドが失敗したら、代わりに `git pull --no-rebase` にフォールバックします。
2. `git push` を実行します。

これは、このスタイルで作業するときに最もよく実行されるネットワークコマンドのシーケンスなので、これを1つのコマンドにまとめることで、多くの時間を節約できます。

**まとめ**

これらのツールは、その前提となっているワークフローに合わせて最適化されています。 開発者と非開発者が一つのプロジェクト上で共同作業を行う際に、双方がすぐに同じように作業を行えるようになっていますし、この種のワークフローにおける多くのベストプラクティスがツールに埋め込まれています。 しかし、あなたのワークフローがその前提と異なっている場合や、ネットワーク操作をいつどのように行うかをより細かく制御したい場合には、別のクライアントを使うか、またはコマンドラインからGitを使用することをお勧めします。

### その他のGUI

グラフィカルなGitクライアントは他にもあり、一つの目的に特化したツールから、Gitのできることは全て操作可能にしようとしているアプリケーションまで多岐に渡ります。 Gitのオフィシャルウェブサイトには、よく使われているクライアントのリストがあります。詳しくは http://git-scm.com/downloads/guis を参照してください。 また、より包括的なリストは Git wiki のサイトに掲載されています。詳しくは https://git.wiki.kernel.org/index.php/Interfaces,_frontends,_and_tools#Graphical_Interfaces を参照してください。

## Visual StudioでGitを使う

Visual Studio 2013 Update 1以降では、IDEにGitクライアントが組み込まれています。 Visual Studioには、かなり昔からソース管理システムを統合した機能が備わっていました。ただ、それは集中型の、ファイルロックベースのシステムを志向したもので、Gitはそのようなワークフローには適合していませんでした。 Visual Studio 2013におけるGitのサポートは、以前の機能とは別物です。その結果、Visual StudioとGitはよりうまく適合するようになっています。

この機能を表示するには、Gitの制御下にあるプロジェクトを開き（または既存のプロジェクトで `git init` を実行し）、メニューから[表示]>[チームエクスプローラー]を選択します。 すると、だいたいこんな感じで「接続」ビューが表示されます。

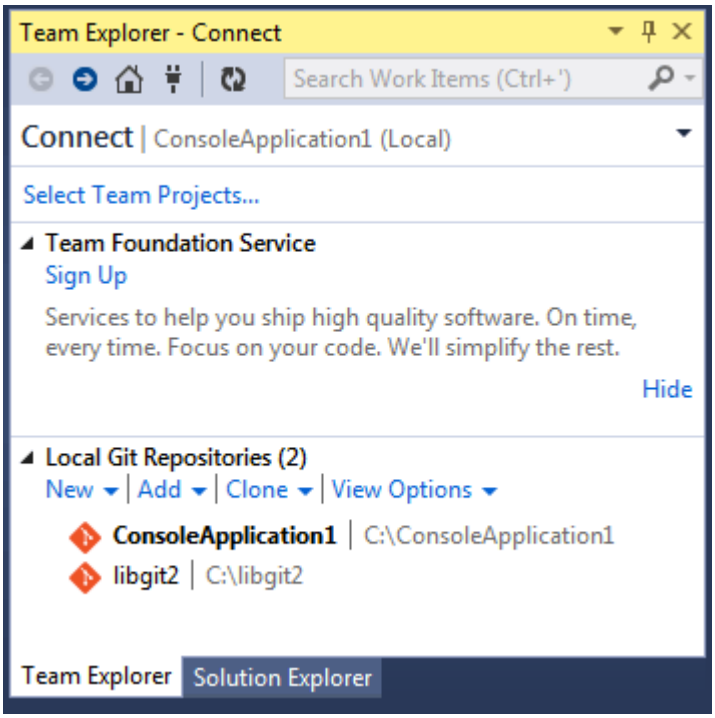



*Figure 158. チームエクスプローラからGitリポジトリへ接続する*

Visual Studioは、これまでに開いたプロジェクトのうち、Gitの制御下にあるものをすべて記憶しています。下部のリストからそれを選択できます。 開きたいプロジェクトが表示されていない場合は、「追加」リンクをクリックして作業ディレクトリへのパスを入力します。 ローカルのGitリポジトリをダブルクリックすると、Visual StudioでのGitリポジトリの"ホーム"ビュー のようなホームビューが表示されます。 これはGitのアクションを実行するためのハブとして働きます。コードを 書いて いる間は、おそらく「変更」ビューでほ

とんどの時間を費やすはずです。チームメイトが行った変更をプルするときは、「同期されていないコミット」ビューと「分岐」ビューを使用することになるでしょう。

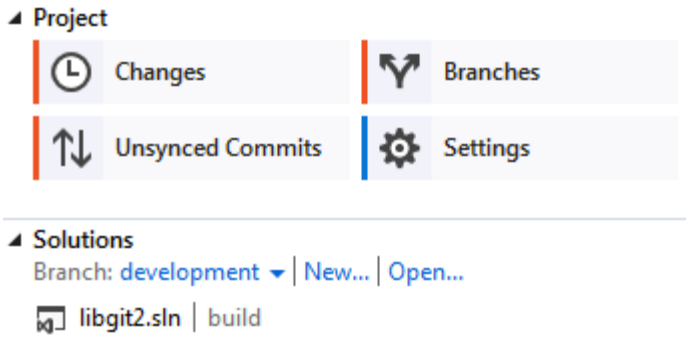



*Figure 159. Visual StudioでのGitリポジトリの"ホーム"ビュー*

Visual Studioには現在、Gitのための強力なタスク指向UIが備わっています。 これには、リニアな歴史ビュー、差分ビューア、リモートコマンドなど多くの機能が含まれています。 この機能の完全なドキュメントは（ここには書ききれないので）、 http://msdn.microsoft.com/en-us/library/hh850437.aspx を参照してください。

# EclipseでGitを使う

EclipseにはEgitというプラグインが同梱されています。Egitは、Gitのほぼすべての操作を行えるインタフェースを備えています。 Egitは、Gitパースペクティブに切り替えることで使用できます（[ウィンドウ] > [パースペクティブを開く] > [その他…] から "Git" を選択）。

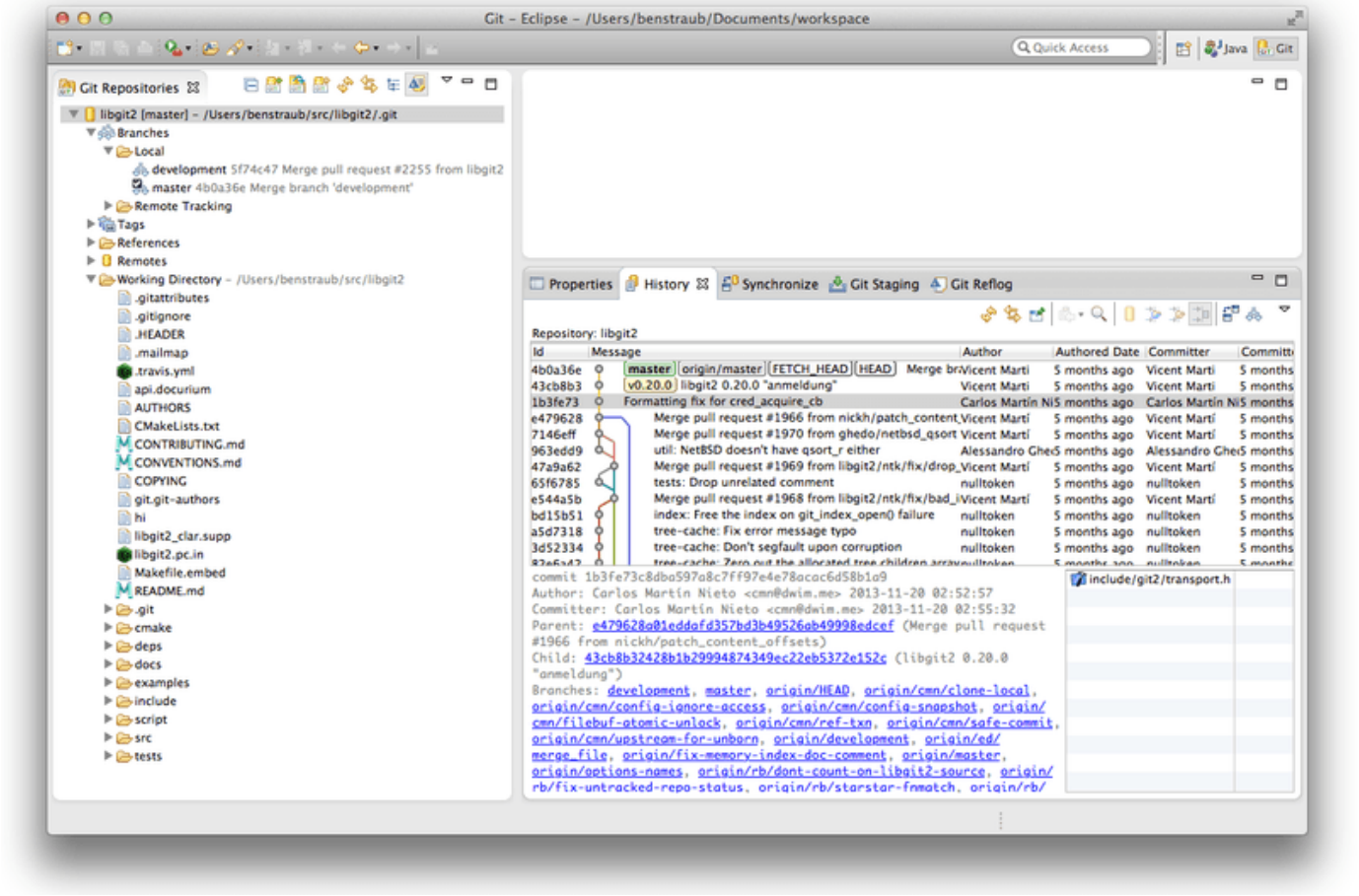



*Figure 160. EclipseのEGitの環境*

EGitには優れたドキュメントが大量に同梱されています。[ヘルプ] > [ヘルプ目次] で、リストから [EGit Documentation] を選択することで内容を参照できます。

# BashでGitを使う

あなたがBashのユーザであれば、シェルの機能を活用して、Gitをより扱いやすくすることができます。 Gitは実際、いくつかのシェルのプラグインを同梱した状態で配布されていますが、それらプラグインはデフォルトではオンになっていません。

まず、Gitのソースコードから `contrib/completion/git-completion.bash` ファイルのコピーを取得する必要があります。 取得したファイルをどこか適当な場所（例えばホームディレクトリ）へコピーした上で、 `.bashrc` に次の内容を追加します。

```
. ~/git-completion.bash
```

設定が完了したら、カレントディレクトリをgitリポジトリに変更し、次のようにタイプしてみてください。

```
$ git chec<tab>
```

……するとBashがオートコンプリートで `git checkout` まで入力してくれるはずです。 このオートコンプリートは、必要に応じて、Gitのサブコマンド、コマンドラインパラメータ、リモートおよび参照の名前に対して働きます。

プロンプトをカスタマイズして、カレントディレクトリのGitリポジトリの情報を表示するのも便利です。 表示する内容は、好みに応じてシンプルにも複雑にもできます。ですが、一般的に多くの人は、現在のブランチや作業ディレクトリの状態のような重要な情報だけを必要とします。 プロンプトにこれらを追加するには、Gitのソースリポジトリから `contrib/completion/git-prompt.sh` ファイルをあなたのホームディレクトリにコピーし、次のような内容を `.bashrc` に追加します。

```
. ~/git-prompt.sh
export GIT_PS1_SHOWDIRTYSTATE=1
export PS1='\w$(__git_ps1 " (%s)")\$ '
```

`\w` は現在の作業ディレクトリを表示するという意味、`\$` はプロンプト末尾に `$` を表示するという意味、`__git_ps1 " (%s)"` は `git-prompt.sh` が提供している関数を書式引数を指定して呼び出しています。 これで、Gitの制御下にあるプロジェクトの中に入ると、bashのプロンプトは次のようになるはずです。


*Figure 161. カスタマイズされた `bash` プロンプト*

これらのスクリプトにはいずれも役に立つドキュメントが付属しています。詳細については、 `git-completion.bash` と `git-prompt.sh` の内容を見てください。

## ZshでGitを使う

Zshには、Git用のタブ補完ライブラリも同梱されています。 `.zshrc`に`autoload -Uz compinit && compinit`という行を追加するだけで、使えるようになります。 Zshのインターフェイスは、Bashよりさらに強力です。

```
$ git che<tab>
check-attr        -- display gitattributes information
check-ref-format  -- ensure that a reference name is well formed
checkout          -- checkout branch or paths to working tree
checkout-index    -- copy files from index to working directory
cherry            -- find commits not merged upstream
cherry-pick       -- apply changes introduced by some existing commits
```

タブ補完の結果が一意に定まらない場合にできることは、候補のリスト表示だけではありません。役に立つ説明が表示されますし、繰り返しタブを押下すれば、グラフィカルにリスト内をナビゲートすることもできます。 この機能は、Gitのコマンド、Gitコマンドの引数、リポジトリ内にあるものの名前（参照やリモートなど）に対して働きます。また、ファイル名や、その他Zsh自身がタブ補完の方法を知っている要素に対しても働きます。

また、バージョン管理システムから情報を読み取るためのフレームワークがZshには同梱されています。 ブランチ名をプロンプトの右端に表示するには、 `~/.zshrc` ファイルに次の内容を追加します。

```
autoload -Uz vcs_info
precmd_vcs_info() { vcs_info }
precmd_functions+=( precmd_vcs_info )
setopt prompt_subst
RPROMPT=\$vcs_info_msg_0_
# PROMPT=\$vcs_info_msg_0_'%# '
zstyle ':vcs_info:git:*' formats '%b'
```

これで、シェルがGitリポジトリ内にいるときには、ターミナルウィンドウの右側に現在のブランチ名が表示されるようになります。 （左側に表示することももちろん可能です。そうしたければ、PROMPT 部分のコメントを外してください。） 見た目は次のようになります。


*Figure 162. カスタマイズされた zsh のプロンプト*

vcs_infoについての詳細は、`zshcontrib(1)`マニュアルにあるドキュメントか、オンラインであれば http://zsh.sourceforge.net/Doc/Release/User-Contributions.html#Version-Control-Information を確認してみてください。

一方、Gitに同梱されている`git-prompt.sh`というスクリプトでも、プロンプトをカスタマイズすることができます。vcs_infoよりも気に入るかもしれませんね。詳しくは http://git-prompt.sh を確認してみてください。 `git-prompt.sh`はBashとZshの両方に対応しています。

Zshは非常にパワフルであり、Zshには自身を改善するためのフレームワークも備わっています。 そのフレームワークの一つに "oh-my-zsh" があります。これは https://github.com/robbyrussell/oh-my-zsh にあります。 oh-my-zshのプラグインシステムには、強力なGit用タブ補完機能が付属しています。また、各種のプロンプトの「テーマ」が付属していて、バージョン管理に関するデータをプロンプトに表示できます。 oh-my-zshのテーマの例 は、このシステムでできることの一例に過ぎません。

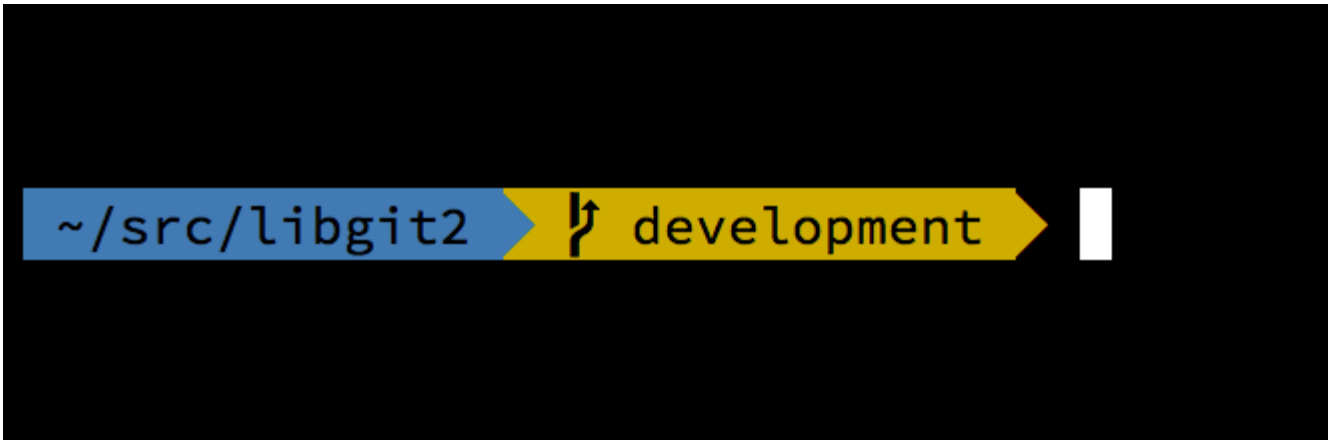



*Figure 163. oh-my-zshのテーマの例*

## PowershellでGitを使う

Windows標準のコマンドライン端末（`cmd.exe`）では、Git向けにユーザ経験をカスタマイズすることができません。一方、PowerShellを使用しているならラッキーです。 Posh-Git (https://github.com/dahlbyk/posh-git) というパッケージが、強力なタブ補完機能や、リポジトリの状態を把握するのに役立つプロンプト表示を提供しています。 表示は次のようになります。

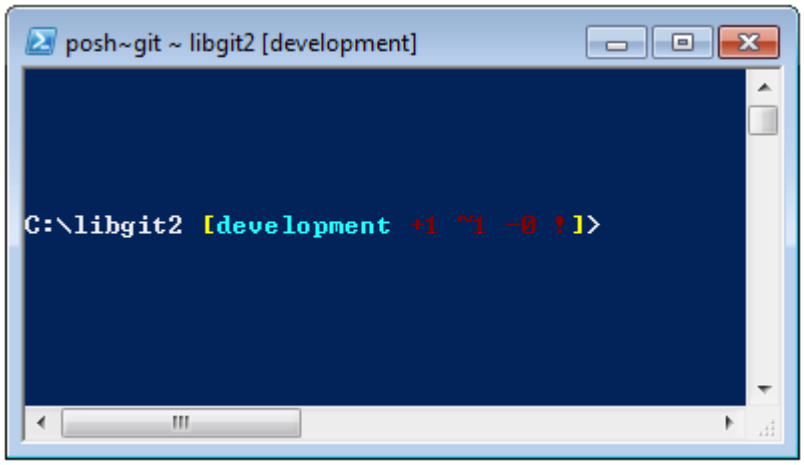



*Figure 164. PowershellでPosh-gitを使う*

Windows用のGitHubクライアントをインストールしている場合は、Posh-Gitがデフォルトで含まれています。必要な作業は、 `profile.ps1` （通常 `C:\Users\<username>\Documents\WindowsPowerShell` に配置されます）に次の内容を追加するだけです。

```
. (Resolve-Path "$env:LOCALAPPDATA\GitHub\shell.ps1")
. $env:github_posh_git\profile.example.ps1
```

Windows用GitHubクライアントのユーザでない場合は、 (https://github.com/dahlbyk/posh-git) からPosh-Gitのリリースをダウンロードし、 `WindowsPowershell` ディレクトリに解凍してください。 その後、管理

者権限で PowerShell プロンプトを開き、次の操作を行います。

```
> Set-ExecutionPolicy RemoteSigned -Scope CurrentUser -Confirm
> cd ~\Documents\WindowsPowerShell\posh-git
> .\install.ps1
```

これで `profile.ps1` ファイルに適切な内容が追加されます。次にプロンプトを開いた時に、 posh-git が有効になります。

# まとめ

この章では、日々の作業で使うツールからGitのパワーを活用する方法、および自作のプログラムからGitリポジトリにアクセスする方法を学びました。

# Appendix B: Gitをあなたのアプリケーションに組み込む

開発者向けのアプリケーションを開発しているなら、ソース管理機能を統合することで利益が得られる可能性は高いでしょう。 また、非開発者向けのアプリケーション（例えばドキュメントエディタなど）であっても、バージョン管理機能から利益を享受できる可能性があります。Gitのモデルは、様々なシナリオに上手く適合します。

Gitをアプリケーションに統合する場合、やり方は大きく分けて3種類あります。1つ目はシェルのプロセスを生成してGitのコマンドラインツールを使う方法、2つ目はLibgit2を使う方法、3つ目はJGitを使う方法です。

## Gitのコマンドラインツールを使う方法

1つ目の方法は、シェルのプロセスを生成して、Gitのコマンドラインツールを使って処理を行うやり方です。この方法には、標準的な方法であるという利点がありますし、Gitのすべての機能がサポートされています。また、ほとんどの実行環境には、比較的簡単にコマンドライン引数つきでプロセスを呼び出す機能が備わっているため、非常に簡単でもあります。 ただし、この方法にはいくつか欠点があります。

一つ目は、出力が全てプレインテキストであるという点です。 これはつまり、処理の進捗や結果を取得したければ、Gitの出力フォーマット（ちょくちょく変わる）を自前でパースする必要があるということです。これは非効率的ですし、エラーも発生しやすくなります。

2つ目は、エラーから回復する方法がないという点です。 リポジトリが何らかの形で壊れていたり、ユーザが設定に不正な値を指定していた場合でも、Gitは単に多くの操作を受け付けなくなるだけです。

3つ目は、プロセス管理です。 シェル環境を別プロセスとして管理する必要があるため、処理が不必要に複雑になります。 複数のGitのプロセスを協調動作させるのは（特に、複数のプロセスが同じリポジトリへアクセスする可能性がある場合は）、時に相当な困難を伴います。

## Libgit2を使う方法

あなたが取れる2つ目のオプションは、Libgit2を使用することです。 Libgit2は、他のプログラムへの依存性のないGitの実装であり、プログラムから使いやすいAPIを提供することにフォーカスしています。 Libgit2は http://libgit2.github.com から取得できます。

まずは、C言語用のAPIがどのようなものか見てみましょう。 ここは駆け足で行きます。

```
// リポジトリを開く
git_repository *repo;
int error = git_repository_open(&repo, "/path/to/repository");

// HEADへの参照を解決してコミットを取得
git_object *head_commit;
error = git_revparse_single(&head_commit, repo, "HEAD^{commit}");
git_commit *commit = (git_commit*)head_commit;

// コミットのプロパティのうちいくつかを出力
printf("%s", git_commit_message(commit));
const git_signature *author = git_commit_author(commit);
printf("%s <%s>\n", author->name, author->email);
const git_oid *tree_id = git_commit_tree_id(commit);

// クリーンアップ
git_commit_free(commit);
git_repository_free(repo);
```

最初の2行で、Gitのリポジトリを開いています。 `git_repository` 型は、メモリにキャッシュされているリポジトリへのハンドルを表しています。 リポジトリの作業ディレクトリか、または `.git` フォルダの正確なパスが分かっている場合、これがリポジトリを開く最もシンプルな方法です。 他の方法としては、 `git_repository_open_ext` を使って検索オプション付きで開く方法、 `git_clone` とその仲間を使ってリモートリポジトリのローカルなクローンを作る方法、 `git_repository_init` を使って全く新規にリポジトリを作る方法があります。

2番目のコードのかたまりは、 rev-parse 文法（詳細は ブランチの参照 を参照）を使って、HEADが最終的に指しているコミットを取得しています。 戻り値は `git_object` 型のポインタで、これはリポジトリのGitオブジェクトデータベースに存在する何かを表しています。 `git_object` 型は、実際には数種類のオブジェクトの “親” にあたります。 “子” にあたる型のメモリレイアウトは `git_object` 型と同じになっているので、正しい型へのキャストは安全に行えます。 上記の場合では、 `git_object_type(commit)` が `GIT_OBJ_COMMIT` を返すので、 `git_commit` 型のポインタへ安全にキャストできます。

次のかたまりは、コミットのプロパティにアクセスする方法を示しています。 ここの最後の行では `git_oid` 型を使用しています。これは、 Libgit2 において SHA-1 ハッシュを表現する型です。

このサンプルからは、いくつかのパターンが見て取れます。

- ポインタを宣言して、 Libgit2 の呼び出しにそのポインタへの参照を渡すと、その呼び出しは多くの場合 int 型のエラーコードを返す。 値 `0` は成功を表す。それより小さい値はエラーを表す。
- Libgit2 がポインタへ値を入れて返したら、解放は自前で行わなければならない。
- Libgit2 の呼び出しが `const` ポインタを返した場合、開放する必要はない。ただし、それがそれが属するオブジェクトが解放されたら、ポインタは無効になる。
- Cでコードを書くのはちょっとキツい。

最後の1つは、 Libgit2 を使用するときに、C言語でコードを書こうということはまずないだろう、というくらいの意味です。 幸いなことに、様々な言語用のバインディングが利用可能です。これを使えば、あなたの使っている特定の言語や環境から、Gitリポジトリに対する作業を非常に簡単に行えます。 Libgit2 の Ruby 向けバインディングを使って上記の例を書いたものを見てみましょう。Libgit2 の Ruby 向けバインディングは Rugged という名前で、 https://github.com/libgit2/rugged から取得できます。

```
repo = Rugged::Repository.new('path/to/repository')
commit = repo.head.target
puts commit.message
puts "#{commit.author[:name]} <#{commit.author[:email]}>"
tree = commit.tree
```

ご覧のように、コードがだいぶすっきりしました。 第一に、 Rugged は例外を使用します。エラーの状態を知らせるのに、 `ConfigError` や `ObjectError` のような例外を raise できます。 第二に、リソースの明示的な解放処理がありません。これは、 Ruby がガベージコレクションをしてくれるためです。 それではもう少し複雑な例を見てみましょう。次の例では、コミットをゼロから作成しています。

```
blob_id = repo.write("Blob contents", :blob) ①

index = repo.index
index.read_tree(repo.head.target.tree)
index.add(:path => 'newfile.txt', :oid => blob_id) ②

sig = {
    :email => "bob@example.com",
    :name => "Bob User",
    :time => Time.now,
}

commit_id = Rugged::Commit.create(repo,
    :tree => index.write_tree(repo), ③
    :author => sig,
    :committer => sig, ④
    :message => "Add newfile.txt", ⑤
    :parents => repo.empty? ? [] : [ repo.head.target ].compact, ⑥
    :update_ref => 'HEAD', ⑦
)
commit = repo.lookup(commit_id) ⑧
```

① 新しいファイルの内容を含む新しい blob を作成します。

② インデックスとHEADのコミットのツリーを取得し、パス `newfile.txt` にある新しいファイルを追加します。

③ ODBに新しいツリーを作成し、それを新しいコミット用に使用しています。

④ author フィールドと committer フィールドに同じ署名を使います。

⑤ コミットメッセージです。

⑥ コミットを作成するときには、新しいコミットの親を指定する必要があります。 ここではHEADの先端を単一の親として指定しています。

⑦ Rugged （およびLibgit2）では、コミットを作成する際に、必要に応じて参照を更新することもできます。

⑧ 戻り値は新しいコミットオブジェクトの SHA-1 ハッシュです。これは後で `Commit` オブジェクトを取得するために使用できます。

このRubyのコードは単純明快です。また、重い処理はLibgit2が行っているので、非常に高速に実行できます。 Rubyist でない方のために、 その他のバインディング では他のバインディングにも触れています。

## 高度な機能

Libgit2 には、Git のコアがスコープ外としている機能がいくつか備わっています。 一つの例がプラグイン機能です。 Libgit2 では、一部の機能に対し、カスタム “バックエンド” を指定できます。これにより、Git が行うのとは別の方法でデータを保存することができます。 Libgit2 では設定、refストレージ、オブジェクトデータベースなどに対してカスタムバックエンドを指定できます。

バックエンドがどのように機能するか見てみましょう。 次のコードは、Libgit2チームが提供しているサンプル（ https://github.com/libgit2/libgit2-backends から取得できます）から拝借しています。 オブジェクトデータベース用のカスタムバックエンドを設定する方法を示しています。

```
git_odb *odb;
int error = git_odb_new(&odb); ①

git_odb_backend *my_backend;
error = git_odb_backend_mine(&my_backend, /*…*/); ②

error = git_odb_add_backend(odb, my_backend, 1); ③

git_repository *repo;
error = git_repository_open(&repo, "some-path");
error = git_repository_set_odb(odb); ④
```

*(ここで、エラーの捕捉はしていますが、エラー処理は行っていないことに注意してください。あなたのコードが私たちのものより優れていることを願っています。)*

① 空のオブジェクトデータベース(ODB) “フロントエンド” を初期化します。これは、実際の処理を行う “バックエンド” のコンテナとして機能します。

② カスタムODBバックエンドを初期化します。

③ フロントエンドにバックエンドを追加します。

④ リポジトリを開きます。作成したODBを、オブジェクトの検索に使うように設定します。

さて、この `git_odb_backend_mine` というのは何でしょうか？
そう、これは自作のODB実装のコンストラクタです。この中では、 `git_odb_backend` 構造体へ適切に値を設定しさえしていれば、どんな処理でも行えます。 処理は *例えば* 以下のようになります。

```
typedef struct {
    git_odb_backend parent;

    // Some other stuff
    void *custom_context;
} my_backend_struct;

int git_odb_backend_mine(git_odb_backend **backend_out, /*…*/)
{
    my_backend_struct *backend;

    backend = calloc(1, sizeof (my_backend_struct));

    backend->custom_context = …;

    backend->parent.read = &my_backend__read;
    backend->parent.read_prefix = &my_backend__read_prefix;
    backend->parent.read_header = &my_backend__read_header;
    // …

    *backend_out = (git_odb_backend *) backend;

    return GIT_SUCCESS;
}
```

ここで、非常に分かりにくい制約として、 `my_backend_struct` の最初のメンバ変数は `git_odb_backend` 構造体である必要があります。これによって、Libgit2 のコードが期待している通りのメモリレイアウトになることが保証されます。 構造体の残りの部分は任意です。この構造体は必要に合わせて大きくしたり小さくしたりして構いません。

この初期化関数では、構造体にメモリを割り当て、カスタムコンテキストを設定し、それがサポートしている `parent` 構造体のメンバーへデータを設定しています。 その他の呼び出しのシグネチャについては、Libgit2 のソースの `include/git2/sys/odb_backend.h` ファイルを見てみてください。ユースケースがはっきりしていれば、シグネチャのうちどれをサポートすればよいかを判断するのに役立つでしょう。

## その他のバインディング

Libgit2 には各種の言語向けのバインディングがあります。 ここでは、これを書いている時点で利用できるバインディングの中から、その一部を使用して、小さなサンプルプログラムを示していきます。他にも、C++、Go、Node.js、Erlang、JVMなど多くの言語向けのライブラリがあり、成熟度合いも様々です。 バインディングの公式なコレクションは、 https://github.com/libgit2 にあるリポジトリを探せば見つかります。 以降で示すコードはいずれも、最終的にHEADが指しているコミットのコミットメッセージを返します (`git log -1` のようなものです)。

### LibGit2Sharp

*.NET や Mono でアプリケーションを書いているなら、 LibGit2Sharp (https://github.com/libgit2/libgit2sharp) をお探しでしょう。*

バインディングは C# で書かれていて、生の Libgit2 の呼び出しを、ネイティブ感のある CLR API でラップすることに細心の注意が払われています。 サンプルプログラムは次のようになります。

```
new Repository(@"C:\path\to\repo").Head.Tip.Message;
```

Windows向けのデスクトップアプリケーション向けにはNuGetパッケージもあります。これは、すぐに作業を始めようという時に役立ちます。

### objective-git

Apple のプラットフォーム向けのアプリケーションを書いているなら、おそらく実装には Objective-C を使用しているものと思います。 Objective-Git (https://github.com/libgit2/objective-git) は、そういった環境向けの Libgit2 のバインディングです。 サンプルプログラムは次のようになります。

```
GTRepository *repo =
    [[GTRepository alloc] initWithURL:[NSURL fileURLWithPath:
@"/path/to/repo"] error:NULL];
NSString *msg = [[[repo headReferenceWithError:NULL] resolvedTarget]
message];
```

Objective-git は Swift に対しても完全な相互運用性があるので、 Objective-C を捨てたとしても怖くありません。

### pygit2

Libgit2 の Python 向けバインディングは Pygit2 という名前で、 http://www.pygit2.org/ から取得できます。サンプルプログラムは次のようになります。

```
pygit2.Repository("/path/to/repo") # リポジトリを開く
    .head                          # 現在のブランチを取得
    .peel(pygit2.Commit)           # HEADが指すコミットまで移動
    .message                       # メッセージを読む
```

### 参考文献

もちろん、 Libgit2 の機能の扱い方すべてを取り上げるのは、本書の範囲外です。 Libgit2 自体についてより多くの情報が必要な場合は、 API ドキュメントが https://libgit2.github.com/libgit2 にあります。また、ガイドが https://libgit2.github.com/docs にあります。 他のバインディングについては、同梱されている README やテストを見てみてください。ちょっとしたチュートリアルや、参考文献へのポインタが書かれていることがあります。

## JGit

JavaのプログラムからGitを使いたい場合、十分な機能を備えたGitのライブラリであるJGitが利用できます。JGitは、ネイティブJavaによるGitの実装です。Gitのほぼ全機能を備えており、Javaコミュニティで広く使われています。 JGitはEclipse傘下のプロジェクトで、ホームページは http://www.eclipse.org/jgit です。

### セットアップする

JGitをあなたのプロジェクトへ追加して、コードを書き始めるには、いくつかの方法があります。 おそらく最も簡単なのはMavenを使う方法です。次のスニペットを pom.xml の `<dependencies>` タグに追加すれば、統合が行えます。

```
<dependency>
    <groupId>org.eclipse.jgit</groupId>
    <artifactId>org.eclipse.jgit</artifactId>
    <version>3.5.0.201409260305-r</version>
</dependency>
```

皆さんがこれを読んでいる時には、おそらく `version` の番号はもっと進んでいるでしょうから、http://mvnrepository.com/artifact/org.eclipse.jgit/org.eclipse.jgit で最新のリポジトリの情報を確認してください。 このステップが完了したら、以降は必要なJGitライブラリの取得と使用をMavenが自動的に行ってくれます。

バイナリの依存関係を自前で管理したい場合は、ビルド済みのJGitのバイナリが http://www.eclipse.org/jgit/download から取得できます。 JGitをプロジェクトへ組み込むには、次のようなコマンドを実行します。

```
javac -cp .:org.eclipse.jgit-3.5.0.201409260305-r.jar App.java
java -cp .:org.eclipse.jgit-3.5.0.201409260305-r.jar App
```

### Plumbing

JGitのAPIには大きく2 つのレベルがあり、それぞれ配管（Plumbing）および磁器（Porcelain）と呼ばれています。 これらはGit由来の用語で、JGitでもだいたいGitと同じように区分けされています。Porcelain APIは、使いやすいフロントエンドで、一般的なユーザレベルの処理（普通のユーザがGitのコマンドラインツールを使って行うような処理）を行います。一方、Plumbing APIでは、低レベルなリポジトリオブジェクトを直接

操作します。

JGitセッションでは多くの場合、`Repository` クラスを開始点とします。この場合、まず最初に行いたい処理は `Repository` クラスのインスタンスの作成です。 ファイルシステムベースのリポジトリなら（そう、JGitでは他のストレージモデルも扱えます）、これは `FileRepositoryBuilder` を使って行います。

```
// 新しくリポジトリを作成する。存在するパスを指定すること
Repository newlyCreatedRepo = FileRepositoryBuilder.create(
    new File("/tmp/new_repo/.git"));

// 既存のリポジトリを開く
Repository existingRepo = new FileRepositoryBuilder()
    .setGitDir(new File("my_repo/.git"))
    .build();
```

`FileRepositoryBuilder` は洗練されたAPIが備わっており、リポジトリの場所が分かっているにしろいないにしろ、Gitのリポジトリを見つけるのに必要な処理はすべて提供されています。 ここでは、環境変数を使う (`.readEnvironment()`)、作業ディレクトリ中のどこかを起点として検索をする(`.setWorkTree(…).findGitDir()`)、上の例のように単に既知の `.git` ディレクトリを開く、といった方法が使用できます。

`Repository` インスタンスを取得したら、そこを起点にあらゆる種類の処理が行えます。 簡単なサンプルプログラムを次に示します。

```
// 参照を取得する
Ref master = repo.getRef("master");

// 参照の指すオブジェクトを取得する
ObjectId masterTip = master.getObjectId();

// Rev-parse文法を使う
ObjectId obj = repo.resolve("HEAD^{tree}");

// オブジェクトの生の内容をロードする
ObjectLoader loader = repo.open(masterTip);
loader.copyTo(System.out);

// ブランチを作成する
RefUpdate createBranch1 = repo.updateRef("refs/heads/branch1");
createBranch1.setNewObjectId(masterTip);
createBranch1.update();

// ブランチを削除する
RefUpdate deleteBranch1 = repo.updateRef("refs/heads/branch1");
deleteBranch1.setForceUpdate(true);
deleteBranch1.delete();

// 設定値
Config cfg = repo.getConfig();
String name = cfg.getString("user", null, "name");
```

ここでは数多くの処理が行われています。1セクションずつ順に見て行きましょう。

最初の行では `master` 参照へのポインタを取得しています。 JGitは `refs/heads/master` にある *実際の* master参照を自動的に取得してオブジェクトを返します。このオブジェクトを使えば、参照についての情報を取得できます。 ここでは、名前 (`.getName()`) と、直接参照のターゲットオブジェクト (`.getObjectId()`) またはシンボリック参照の指す参照 (`.getTarget()`) のいずれかを取得できます。 参照オブジェクトは、タグ参照やオブジェクトを表すのにも使われるので、タグが “peeled” か問い合わせられるようになっています。つまり、参照がタグオブジェクトの（ひょっとすると長い）列の最後のターゲットを指しているか問い合わせることができます。

2行目では、 `master` 参照の指す先を取得して、ObjectIdインスタンスの形で返します。 ObjectIdはGitのオブジェクトデータベース中にある（または、データベース中にない）オブジェクトのSHA-1ハッシュを表しています。 3行目は似たような処理ですが、ここではJGitがrev-parse文法（詳細は ブランチの参照 を参照）を処理する方法を示しています。Gitが解釈できる任意のオブジェクト指定子を渡すことができ、JGitはそのオブジェクトのvalidなObjectIdか `null` のどちらかを返します。

次の2行はオブジェクトの生の内容をロードする方法を示しています。 このサンプルでは `ObjectLoader.copyTo()` を使ってオブジェクトの内容を標準出力へ直接流し込んでいますが、ObjectLoaderにはオブジェクトの型やサイズを返すメソッド、オブジェクトの内容をbyte型配列として返すメソッドもあります。 大きいオブジェクト（`.isLarge()` が `true` を返すようなオブジェクト）に対し

ては、 `.openStream()` を使えば、InputStream
のようなオブジェクトを取得でき、データ全体をメモリ上に置くことなく、生のデータを読み込めます。

次の数行は、新しいブランチを作成するために必要な処理を示しています。 ここではRefUpdateのインスタンスを作成し、パラメータを設定した上で、`.update()` を呼んで変更を適用しています。 続く数行は同じブランチを削除するコードです。 なお、この処理では `.setForceUpdate(true)` が必須です。さもなくば、`.delete()` を呼んでも `REJECTED` が返り、何も変更されません。

最後の例は、Gitの設定ファイルから `user.name` の値を取得する方法を示しています。 このConfigインスタンスは、ローカル設定のために前に開いたリポジトリを使用しますが、グローバル設定ファイルやシステム設定ファイルからも自動的に値を読み込みます。

ここで示したサンプルは、Plumbing APIのごく一部であり、利用可能なメソッドやクラスは他にもたくさんあります。 ここで取り上げなかった内容としては、他にJGitのエラー処理があります。エラー処理は例外を通じて行われます。 JGitのAPIからthrowされる例外には、Java標準の例外（`IOException` など）の他にも、JGit固有の各種例外（`NoRemoteRepositoryException`, `CorruptObjectException`, `NoMergeBaseException` など）があります。

## Porcelain

Plumbing APIは網羅的ではありますが、その機能を繋ぎ合わせて一般的な作業（インデックスにファイルを追加したり、新しくコミットを作成したり）を遂行するのは、場合によっては面倒です。 JGitは、この点を手助けする高いレベルのAPIを提供しています。これらのAPIへのエントリポイントは、 `Git` クラスです。

```
Repository repo;
// repoオブジェクトの作成……
Git git = new Git(repo);
```

Gitクラスは、洗練された高レベルの *builder* スタイルのメソッドを備えています。これは、非常に複雑な処理を組み立てる際に利用できます。 それでは例を見てみましょう。ここでは `git ls-remote` のような処理を行っています。

```
CredentialsProvider cp = new
UsernamePasswordCredentialsProvider("username", "p4ssw0rd");
Collection<Ref> remoteRefs = git.lsRemote()
    .setCredentialsProvider(cp)
    .setRemote("origin")
    .setTags(true)
    .setHeads(false)
    .call();
for (Ref ref : remoteRefs) {
    System.out.println(ref.getName() + " -> " + ref.getObjectId().name());
}
```

これはGitクラスを使うときによくあるパターンです。メソッドがコマンドオブジェクトを返すので、パラメータを設定するメソッドチェーンを繋げていき、最後に `.call()` メソッドを呼び出すとそれらがまとめて

実行されます。 このケースでは、タグを取得する際に、HEADではなく`origin`リモートを要求しています。 また、`CredentialsProvider` オブジェクトを使って認証を行っていることにも注意してください。

Gitクラスからは `add`、`blame`、`commit`、`clean`、`push`、`rebase`、`revert`、`reset` を含め、他にも多くのコマンドが使用できます。

## 参考文献

この節で示したのは、JGitの機能のごく一部です。 興味が湧いた、もっと知りたいということなら、情報は次の場所から探せます。

- オフィシャルなJGit APIドキュメントは http://download.eclipse.org/jgit/docs/latest/apidocs で参照できます。 標準的なJavadocなので、ローカルにインストールして、好きなJVM IDEから参照することもできます。
- JGit Cookbook (https://github.com/centic9/jgit-cookbook) には、特定の作業をJGitでどうやるかのサンプルプログラムが数多くあります。
- http://stackoverflow.com/questions/6861881 で、優れたリソースへのポインタがいくつか示されています。

# Appendix C: Gitのコマンド

本書を通して、数多くのGitコマンドを紹介してきました。コマンドの説明にあたっては、物語風に、使うコマンドを少しずつ増やしていくように心掛けてきました。しかしその結果、コマンドの使用例が本書の全体に散在する形になってしまいました。

この付録では、本書で扱ったすべてのGitコマンドを見ていきます。コマンドはその用途ごとに大まかにまとめてあります。ここでは、それぞれのコマンドが一般的に何を行うのか、そして本書のどこでそのコマンドが使われていたかについて述べていきます。

## セットアップと設定

最初のGitの呼び出しから、日々行われる微調整や参照に至るまで、非常によく使われるコマンドが2つあります。 `config` コマンドと `help` コマンドです。

### git config

Gitでは、様々な処理についてデフォルトの処理方法があります。その多くでは、デフォルトの処理方法を変えたり、あなた好みの設定をしたりできます。これには、Gitにあなたの名前を教えることから、端末の色の変更や使用するエディタの設定まですべてが含まれます。 このコマンドが読み書きするファイルはいくつかあり、それによってグローバルな設定をしたり特定のリポジトリだけの設定をしたりできるようになっています。

`git config` コマンドは、本書のほぼすべての章で使用されています。

最初のGitの構成 では、Gitを使い始める前に、名前、メールアドレス、エディタの設定を行う際に使用しました。

Git エイリアス では、大量のオプションを毎回打ち込まなくてもいいように、`git config` を使って省略形のコマンドを作成する方法を示しました。

リベース では、`git pull` の実行時に `--rebase` をデフォルトにするのに使用しました。

認証情報の保存 では、HTTPパスワードのデフォルトの保存先を設定するのに使用しました。

キーワード展開 では、Gitへ入ってくるコンテンツ、Gitから出ていくコンテンツに対して、smudgeとcleanフィルタを設定する方法を示しました。

最後に、Git の設定 の内容は基本的にすべてこのコマンドに関するものです。

### git help

`git help` コマンドは、任意のコマンドについて、Gitに同梱されているあらゆるドキュメントを表示します。一般的なものの多くはこの付録でも概要を示していきますが、各コマンドで指定可能なオプションとフラグすべての一覧は、いつでも `git help <command>` で表示できます。

ヘルプを見る では、 `git help` コマンドの紹介を行いました。 サーバーのセットアップ では、 `git shell` に関する詳細情報を見る方法を示しました。

# プロジェクトの取得と作成

Gitリポジトリを取得するには、2つの方法があります。一つはネットワーク上かどこかにある既存のリポジトリをコピーしてくる方法、もう一つは既存のディレクトリに新しくリポジトリを作成する方法です。

## git init

ディレクトリを選んで、そこをGitのリポジトリにしてバージョン管理を始められるようにするには、単に `git init` を実行します。

これを最初に紹介したのは Git リポジトリの取得 で、作業の開始にあたり新規にリポジトリを作成しました。

リモートブランチ では、デフォルトのブランチを “master” から変更する方法を簡単に述べました。

ベアリポジトリのサーバー上への設置 では、サーバ用に空のベアリポジトリを作成するのにこのコマンドを使用しました。

最後に、 配管（Plumbing）と磁器（Porcelain） では、舞台裏で実際に行われていることの詳細について、その一部を見てきました。

## git clone

`git clone` は、実際にはいくつかの他のコマンドのラッパーのようなものです。新しいディレクトリを作成し、そこに入って `git init` を実行して空のGitリポジトリを作成し、渡したURLをリモートに追加（`git remote add`）し（デフォルトでは `origin` という名前が付けられる）、そのリモートリポジトリから `git fetch` し、最後に `git checkout` でワーキングディレクトリに最新のコミットをチェックアウトします。

`git clone` コマンドは本書全体を通して何十回も使用されてきましたが、ここでは興味深い数カ所を列挙します。

基本的には 既存のリポジトリのクローン でコマンドの紹介と説明を行いました。ここではいくつかの例を示しました。

サーバー用の Git の取得 では、 `--bare` オプションを使用してワーキングディレクトリのないGitリポジトリのコピーを作成する方法を見てきました。

バンドルファイルの作成 では、Gitリポジトリのバンドルファイルを展開するのに使用しました。

最後に、 サブモジュールを含むプロジェクトのクローン では、 `--recursive` オプションを使って、サブモジュールつきのリポジトリのクローンを簡単に行う方法を学びました。

このコマンドは本書の他の多くの箇所でも使われていますが、ここではちょっとユニークだったり、他とは違う使われ方をしている箇所を挙げました。

# 基本的なスナップショット

コンテンツをステージングしたり、それを歴史に対してコミットしたりする基本的なワークフローについては、基本的なコマンドが少しあるだけです。

## git add

`git add` コマンドは、次のコミット用に、ワーキングディレクトリからステージングエリア（または“インデックス”）へコンテンツを追加します。`git commit` コマンドの実行時、デフォルトではこのステージングエリアしか参照しません。そのため、 `git add` は、次のコミットのスナップショットが正確にはどのようになっていて欲しいのかを組み立てるのに使用されます。

このコマンドは、Gitの中でも非常に重要なコマンドです。本書の中でも、何十回も言及したり使用したりしています。ここでは、その中で見られるユニークな使用法をいくつか取り上げます。

`git add` の紹介と詳細な説明を最初に行ったのは 新しいファイルの追跡 でした。

マージ時のコンフリクト では、マージの衝突を解決するためにこのコマンドを使用する方法について言及しました。

対話的なステージング では、このコマンドを使って、変更されたファイルのうち、特定の部分だけを対話的にステージングする方法について細かく見てきました。

最後に、 ツリーオブジェクト では、舞台裏で何を行っているかを理解するため、このコマンドを低レベルで模倣しました。

## git status

`git status` コマンドは、作業ディレクトリとステージングエリアとに対して、ファイルの状態について別々の内容を表示します。それぞれ、変更されているがステージングされていないファイルはどれか、ステージングされているがまだコミットされていないファイルはどれかを表示します。通常の使用法では、これらのステージの間でファイルを移動する方法について、基本的なヒントも表示します。

最初に `status` を取り上げたのは ファイルの状態の確認 で、基本的な使用法と単純化された使用法の両方を取り上げました。本書の全体を通してこのコマンドを使ってきましたが、`git status` コマンドで行えることのほぼすべてをここで取り上げています。

## git diff

`git diff` は、2つの任意のツリーで確認したい場合に使用します。これは、作業中の環境とステージングエリアの差異(`git diff` だけで表示される)でもよいですし、ステージングエリアと最後のコミットの差異(`git diff --staged`)でもよいですし、2つのコミットの差異(`git diff master branchB`)でもよいです。

`git diff` の基本的な使用法を最初に見たのは ステージされている変更 / されていない変更の閲覧 でした。ここでは、どの変更がステージングされていて、どの変更がまだステージングされていないのか見る方法を示しました。

コミットの指針 では、`--check` オプションを使って、問題となる空白文字がないかをコミット前に探すのに使いました。

何が変わるのかの把握 では、`git diff A...B` という文法を使って、ブランチ間の差異をより効率的にチェックする方法を見てきました。

高度なマージ手法 では、`-b` を使って空白文字の差異をフィルタしました。また、 `--theirs`、 `--ours`、

`--base` を使って、衝突しているファイルのいろいろな段階を比較しました。

最後に、 サブモジュールの作り方 では、サブモジュールの変更を効率的に比較するために `--submodule` を使いました。

## git difftool

`git difftool` コマンドは、単に外部ツールを起動して、2つのツリーの差異を表示します。これは、ビルトインの `git diff` 以外のコマンドを使いたい場合に使用します。

このコマンドについては、 ステージされている変更 / されていない変更の閲覧 で簡単に言及しただけです。

## git commit

`git commit` コマンドは、`git add` でステージングされたすべてのファイルの内容を取得し、データベースに新しく永続的なスナップショットを記録し、最後に現在のブランチのブランチポインタをそこまで進めます。

コミットの基本を最初に取り上げたのは 変更のコミット でした。そこではまた、 `-a` フラグを使って、日々のワークフローで `git add` を行うステップを省略する方法の実例を示しました。また、`-m` フラグを使って、コミットメッセージを、エディタを起動するのではなくコマンドラインから渡す方法についても説明しました。

作業のやり直し では、`--amend` オプションを使って最新のコミットを取り消す方法を取り上げました。

ブランチとは では、`git commit` が何を行っているか、なぜそのようなことをするのかについて非常に細かいところまで説明しました。

コミットへの署名 では、`-S` フラグを使って、コミットに暗号を使って署名を行う方法を見てきました。

最後に、 コミットオブジェクト では、 `git commit` コマンドがバックグラウンドで何を行っているのか、またそれが実際どのように実装されているのかを簡単に見てきました。

## git reset

`git reset` コマンドは、その動詞から分かるかも知れませんが、主に物事を元に戻すのに使われます。このコマンドは、 `HEAD` ポインタをあちこち動かし、必要に応じて インデックス またはステージングエリアに変更を加えます。 `--hard` を使えば作業ディレクトリを変更することもできます。この最後のオプションは、誤って使用すると作業結果を失う可能性があるので、必ずその点を理解した上で使用してください。

`git reset` の最も単純な使用法を実質的に初めて取り上げたのは ステージしたファイルの取り消し でした。ここでは、 `git add` したファイルのステージを解除するのに使いました。

リセットコマンド詳説 は、全体がこのコマンドの説明に費やされており、このコマンドについてかなり詳細に取り上げています。

[_abort_merge] では、 `git reset --hard` を使用してマージを中断しました。また、`git reset` コマンドのちょっとしたラッパーである `git merge --abort` も使用しました。

### git rm

`git rm` コマンドは、ステージングエリアおよびGitの作業ディレクトリからファイルを削除するのに使用されます。これは、次のコミット用に `git add` でファイルの削除をステージングするのに似ています。

ファイルの削除 では、 `git rm` コマンドの詳細を取り上げました。ファイルを再帰的に削除する方法、`--cached` を使って作業ディレクトリにファイルを残しつつステージングエリアからファイルを削除する方法などについて取り上げました。

これ以外の方法で `git rm` を使用したのは オブジェクトの削除 だけです。ここでは、 `git filter-branch` を実行した際に、 `--ignore-unmatch` について簡単に説明しました。これは、削除しようとしているファイルが存在しなかった場合でもエラーとしないオプションで、スクリプトを作成する際に役立ちます。

### git mv

`git mv` は簡単な便利コマンドで、ファイルを移動した上で、新しいファイルを `git add` し、古いファイルを `git rm` します。

このコマンドについては、 ファイルの移動 で簡単に言及しただけです。

### git clean

`git clean` コマンドは、作業ディレクトリから不要なファイルを削除するのに使用されます。これには、ビルド時の一時ファイルやマージ衝突ファイルの削除が含まれます。

作業ディレクトリの掃除 では、cleanコマンドのオプションの多くや、cleanコマンドを使用する シナリオについて取り上げました。

## ブランチとマージ

Gitのブランチとマージの機能は、その大半がほんの一握りのコマンドで実装されています。

### git branch

`git branch` コマンドは、実際にはブランチ管理ツールのようなものです。あなたの持っているブランチを一覧表示したり、新しいブランチを作成したり、ブランチを削除したり、ブランチの名前を変更したりできます。

Git のブランチ機能 のほとんどは `branch` コマンドに費やされており、この章の全体に渡って`branch` コマンドが使用されています。最初にこのコマンドを紹介したのは 新しいブランチの作成 で、そこで扱った以外の機能（一覧表示と削除）のほとんどは ブランチの管理 で見てきました。

追跡ブランチ では、 `git branch -u` を使用して追跡ブランチを設定しました。

最後に、Gitの参照 では、このコマンドがバックグラウンドで行っていることについて見てきました。

## git checkout

`git checkout` コマンドは、ブランチを切り替える際と、コンテンツを作業ディレクトリへチェックアウトするのに使用されます。

このコマンドは、 ブランチの切り替え で、 `git branch` コマンドとともに初めて登場しました。

追跡ブランチ では、`--track` フラグを使用して、ブランチの追跡を開始する方法を見てきました。

コンフリクトのチェックアウト では、 `--conflict=diff3` を使用して、ファイルの衝突部分を再表示しました。

リセットコマンド詳説 では、`git checkout` と `git reset` の関係の詳細を見てきました。

最後に、HEAD では、実装の詳細の一部を見てきました。

## git merge

`git merge` は、チェックアウト中のブランチに、1つまたは複数のブランチをマージする際に使用されるツールです。 このコマンドは、現在のブランチをマージの結果まで進めます。

`git merge` コマンドを最初に紹介したのは ブランチの基本 でした。このコマンドは本書の様々な場所で使用されていますが、 `merge` コマンドにはごく少数のバリエーションしかありません⍰—⍰その多くは、単に `git merge <branch>` でマージする単一のブランチの名前を指定しているだけです。

フォークされた公開プロジェクト の最後では、マージの際にコミットをひとつにまとめる（Gitがマージを行う際に、マージするブランチの歴史を記録せず、あたかも新しくコミットされたかのようにする）方法について取り上げました。

高度なマージ手法 では、マージのプロセスとコマンドについて多くを見てきました。これには、-Xignore-space-change` コマンドや、 `--abort` フラグを使って問題のあるマージを中断する方法が含まれます。

コミットへの署名 では、あなたのプロジェクトでGPG署名を使っている場合に、マージする前に署名を確認する方法を学びました。

最後に、サブツリーマージ では、サブツリーマージについて学びました。

## git mergetool

`git mergetool` コマンドは、Gitのマージに問題があった場合に、単に外部のマージ補助ツールを起動するコマンドです。

マージ時のコンフリクト では、このコマンドについて簡単に言及しました。また、外部のマージツールおよび diff ツール では、独自の外部マージツールを実装する方法について詳細に見てきました。

## git log

`git log` コマンドは、プロジェクトに記録されている歴史を、最新のコミットのスナップショットから後ろ向きに走査して到達可能な歴史を表示するのに使用されます。デフォルトでは現在のブランチの歴史だけを表

示しますが、別のブランチ、または複数のブランチのHEADを与えて走査させることもできます。また、コミットレベルで複数のブランチ間の差異を表示するために使用されることもあります。

このコマンドは、本書のほぼすべての章で、プロジェクトの歴史の実例を表示するのに使用されています。

コミット履歴の閲覧 では、このコマンドを紹介し、ある程度深く説明を行いました。ここでは、各コミットで何が取り込まれたかを `-p` および `--stat` オプションを使って知る方法、 `--pretty` および `--oneline` オプションで歴史をより簡潔な形で見る方法、および日付や作者で簡単なフィルタリングを行うオプションを見てきました。

新しいブランチの作成 では、 `--decorate` オプションを使用して、ブランチポインタがどこを指しているかを簡単に可視化しました。また、 `--graph` オプションを使用して、分岐した歴史がどのようになっているかを見てきました。

非公開な小規模のチーム および コミットの範囲指定 では、 `git log` コマンドで `branchA..branchB` 形式の構文を使用して、 他のブランチと比較して、あるブランチに固有のコミットはどれかを見る方法を取り上げました。コミットの範囲指定 では、かなり広範囲に渡ってこのコマンドを見てきました。

マージの履歴 および トリプルドット では、どちらか一方のブランチにだけ入っているものは何かを見るための `branchA...branchB` 形式および `--left-right` 構文の使い方を扱いました。マージの履歴 では、マージ衝突のデバッグを支援するための `--merge` オプションの使い方と、歴史の中のマージコミットの衝突を見るための `--cc` オプションの使い方を見てきました。

参照ログの短縮形 では、ブランチを走査する代わりにこのツールを使用してGit reflogを見るために `-g` オプションを使いました。

検索 では、ある機能の歴史など、コードの歴史上で起こった出来事を検索するために、 `-S` および `-L` オプションを使用して非常に洗練された検索を行う方法を見てきました。

コミットへの署名 では、 `--show-signature` を使って、各コミットが正当に署名されているかどうかに基づいて、 `git log` の出力に対してバリデーション文字列を付け加える方法を見てきました。

## git stash

`git stash` コマンドは、未コミットの作業を一時的に保存する際に使用されます。これは、ワーキングディレクトリをきれいにしたいが、作業中の内容をブランチにコミットしたくないという場合に使用されます。

このコマンドの機能は基本的にすべて 作業の隠しかたと消しかた で取り上げました。

## git tag

`git tag`コマンドは、コードの歴史の中で、特定のポイントに永続的なブックマークを付与するのに使用されます。一般的には、このコマンドはリリース作業などで使用されます。

タグ では、このコマンドの紹介と詳細な説明を行いました。また、リリース用のタグ付け では、実際にこのコマンドを使用しました。

作業内容への署名 では、 `-s` フラグを使ってGPGで署名されたタグを作成する方法、および `-v` フラグを使ってタグの署名を検証する方法を取り上げました。

# プロジェクトの共有とアップデート

Gitにおいて、ネットワークにアクセスするコマンドはそれほど多くありません。ほぼ全てのコマンドはローカル・データベース上で動作します。成果物を共有したり、他の場所から変更点をプルする準備ができたら、リモートリポジトリを扱うほんの一握りのコマンドを使います。

## git fetch

`git fetch` コマンドは、リモートリポジトリと通信し、そのリポジトリにあって現在のリポジトリにない情報を全て取得します。またその上で、取得した情報をローカル・データベースへ保存します。

このコマンドを最初に見たのは リモートからのフェッチ、そしてプル でした。続いて、その使用例を リモートブランチ で見てきました。

プロジェクトへの貢献 では、例のいくつかでこのコマンドを使用しました。

プルリクエストの参照 では、デフォルトの範囲の外側から、特定の単一の参照を取得するのにこのコマンドを使用しました。また、 バンドルファイルの作成 では、バンドルからフェッチする方法を見てきました。

Refspec では、 `git fetch` にデフォルトとは少し違った動きをさせるために、高度にカスタマイズされたrefspecをセットアップしました。

## git pull

`git pull` コマンドは、基本的には `git fetch` コマンドと `git merge` コマンドの組み合わせです。Gitは指定したリモートからフェッチを行い、続けて現在のブランチへそれをマージするよう試みます。

リモートからのフェッチ、そしてプル では、このコマンドについて簡単に紹介しました。 リモートの調査 では、このコマンドを実行した場合に何がマージされるのかを見る方法を示しました。

リベースした場合のリベース では、リベースの際の問題に対する支援としてこのコマンドを使用する方法を見てきました。

リモートブランチのチェックアウト では、このコマンドにURLを指定して、一回限りのやり方で変更点をプルする方法を示しました。

最後に、コミットへの署名 では、このコマンドに `--verify-signatures` オプションを使用して、pullの対象のコミットがGPGで署名されていることを検証できることに、ごく簡単に言及しました。

## git push

`git push` コマンドは、他のリポジトリと通信し、自分のローカル・データベースにあって通信先のリポジトリにないものは何かを計算した上で、差分を通信先のリポジトリへプッシュします。このコマンドは、通信先のリポジトリへの書き込みアクセスを必要とするので、通常は何らかの形で認証が行われます。

最初に `git push` コマンドについて見たのは リモートへのプッシュ でした。ここでは、ブランチをリモートリポジトリへプッシュする基本に触れました。プッシュ では、特定のブランチをプッシュする方法について少し詳細に見てきました。 追跡ブランチ では、自動的にプッシュをするために、追跡ブランチを設定する方法を見てきました。リモートブランチの削除 では、`git push` でサーバ上のブランチを削除するために、`--

delete`フラグを使用しました。

プロジェクトへの貢献 では、全体を通して、複数のリモートとブランチ上の成果物を共有する際の `git push` を使用した例をいくつか見てきました。

タグの共有 では、 `--tags` オプションで作成したタグを共有するためにこのコマンドを使用する方法を見てきました。

サブモジュールに加えた変更の公開 では、サブモジュールのサブプロジェクトをプッシュする前に、`--recurse-submodules` オプションを使用して、サブモジュールの成果物が全て公開されているかをチェックしました。これは、サブモジュールを使用している場合に非常に役立ちます。

その他のクライアントフック では、 `pre-push` フックについて簡単に述べました。これは、プッシュが完了する前に実行するよう設定できるスクリプトで、プッシュしてよいかを検査します。

最後に、 refspecへのプッシュ では、普段使用されるショートカットの代わりに、完全なrefspecを使用したプッシュを見てきました。これは、共有したい成果物を厳密に指定する際の助けになります。

## git remote

`git remote` コマンドは、リモートリポジトリの記録を管理するツールです。このコマンドでは、長いURLを毎回タイプしなくて済むように、URLの短縮形（例えば “origin” ）を保存できます。短縮形は複数持つことができます。 `git remote` コマンドは、短縮形の追加、変更、削除に使用されます。

このコマンドは、短縮形の一覧表示、追加、削除、リネームなどを含め、 リモートでの作業 で詳しく取り上げられています。

また、それ以降のほぼすべての章でもこのコマンドは使用されていますが、そこでは常に標準的な `git remote add <name> <url>` の形式で使用されています。

## git archive

`git archive` コマンドは、プロジェクトの特定のスナップショットのアーカイブファイルを作成するのに使用されます。

リリースの準備 では、`git archive` を使用して、プロジェクトの共有用のtarballを作成しました。

## git submodule

`git submodule` コマンドは、通常のリポジトリ内で、外部のリポジトリを管理するのに使用されます。外部リポジトリの内容は、ライブラリだったり、その他の共有リソースだったりします。 `submodule` コマンドには、これらのリソースを管理するために、いくつかのサブコマンド（`add`、`update`、`sync` など）があります。

このコマンドについて言及しているのは サブモジュール だけです。この節だけで、このコマンドのすべてを取り上げています。

# 検査と比較

## git show

`git show` コマンドは、Gitオブジェクトを、人間が読める単純な形で表示します。このコマンドは通常、タグまたはコミットに関する情報を表示するのに使用されます。

最初に 注釈付きのタグ では、このコマンドを使用して、注釈つきのタグの情報を表示しました。

その後 リビジョンの選択 では、各種のリビジョン選択が解決するコミットを示すために、このコマンドを何度も使用しました。

マージの手動再実行 では、`git show` でできることの中でもう一つ興味深いこととして、マージが衝突した際に、様々な状態の中から特定のファイルの内容を抽出しました。

## git shortlog

`git shortlog` コマンドは、`git log` の出力を要約するのに使用されます。このコマンドは、`git log` と同じオプションの多くを受け取りますが、すべてのコミットを一覧表示する代わりに、コミットの作者の単位でまとめた概要を表示します。

短いログ では、このコマンドを使用して、すてきな変更履歴を作成する方法を示しました。

## git describe

`git describe` コマンドは、あるコミットを指し示す何らかの文字列を受け取って、人間が読めてかつ不変であるような文字列を生成します。この文字列は、コミットのSHA-1と同様にひとつのコミットを特定できますが、より理解しやすい形式になっています。

ビルド番号の生成 および リリースの準備 では、後でリリースファイルに名前をつけるために、`git describe` を使用して文字列を取得しました。

# デバッグ

Gitには、コードのデバッグを支援するためのコマンドが2つあります。このコマンドは、どこに問題が入り込んだのかを明らかにするところから、誰がそれを入れ込んだのかを明らかにするところまでを支援してくれます。

## git bisect

`git bisect` は非常に便利なデバッグツールです。バグや問題が最初に入り込んだのがどのコミットか、二分探索を自動的に行って調査します。

このコマンドは 二分探索 でくまなく取り上げました。このコマンドに言及しているのはこの節だけです。

## git blame

`git blame` コマンドは、任意のファイルの各行に対して、注釈を付与して表示します。注釈には、ファイル

の各行を最後に変更したのはどのコミットか、そのコミットの作者は誰かが含まれます。これは、コードの特定の行について質問したいときに、誰に聞いたらいいか調べるのに役立ちます。

このコマンドは ファイルの注記 で取り上げました。このコマンドについて言及しているのはこの節だけです。

### git grep

`git grep` コマンドは、任意の文字列や正規表現でソースコード内を検索することができます。検索は、古いバージョンのプロジェクトに対して行うこともできます。

このコマンドは Git Grep で取り上げました。このコマンドについて言及しているのはこの節だけです。

## パッチの適用

Gitのコマンドのうちいくつかは、コミットとはそれによっておこる修正のことであるという観点で捉え、また一連のコミットを一続きのパッチの集まりであるとみなす考え方を中心としています。これらのコマンドは、この考え方に従ってブランチを管理するのに役立ちます。

### git cherry-pick

`git cherry-pick` コマンドは、あるコミットで行われた変更を取得して、それを現在のブランチへ新しいコミットとして取り込む場合に使用されます。これは、あるブランチをマージしてすべての変更を取り込むのではなく、そのブランチから一つか二つのコミットだけを個別に取り込みたい場合に役立ちます。

リベースとチェリーピックのワークフロー では、チェリーピックの説明を行い、実例を示しました。

### git rebase

`git rebase` コマンドは、基本的には `cherry-pick` を自動化したものです。対象となる一連のコミットを決めた上で、それらを一つずつ、元と同じ順序となるように、どこか別の場所へチェリーピックします。

リベース では、リベースについて詳しく取り上げました。ここでは、公開済みのブランチのリベースに関連した、共同作業の際の課題についても取り上げました。

Git オブジェクトの置き換え では、歴史を分割して、二つの別々のリポジトリへ格納する例を通して、このコマンドの実用的な使い方を示しました。またその際に `--onto` フラグを使用しました。

Rerere では、リベース中に発生するマージコンフリクトについて見てきました。

複数のコミットメッセージの変更 では、 `-i` オプションを指定して、対話的スクリプティングモードを使用しました。

### git revert

`git revert` コマンドは、本質的には `git cherry-pick` コマンドの逆です。このコマンドは、コマンドの対象となるコミットで取り込まれた変更に対して、本質的にはそれを元に戻したり取り消したりすることで、そのコミットとは逆の変更を行うコミットを新規に作成します。

コミットの打ち消し では、マージコミットを元に戻すのにこのコマンドを使用しました。

# メール

Git自体を含め、多くのGitプロジェクトは、もっぱらメーリングリスト上で管理されています。Gitには、メールで簡単に送れるパッチを生成したり、メールボックスからパッチ当てをしたりといった、このプロセスを補助するツールがいくつか組み込まれています。

## git apply

`git apply` コマンドは、`git diff` コマンドまたはGNU diffコマンドで作成したパッチを適用します。ほんの少しの違いを除けば、これは`patch` コマンドが行うであろう処理と同様のものです。

メールで受け取ったパッチの適用 では、このコマンドの使い方と、それを行うであろう状況を例示しました。

## git am

`git am` コマンドは、メールの受信トレイ（特にmboxフォーマットのもの）からパッチを適用するのに使用されます。これは、パッチをメールで受け取った上で、それを簡単にプロジェクトへ適用するのに役立ちます。

am でのパッチの適用 では、`--resolved`、, `-i` および `-3` オプションの使い方を含め、`git am` コマンドの使い方とワークフローを取り上げました。

`git am` に関連したワークフローを便利にするのに使用できるフックは数多くあります。それらのフックはすべて Eメールワークフローフック で取り上げました。

メールでの通知 では、GitHubのプルリクエストの変更点をpatch形式にフォーマットしたものを、このコマンドを使用して適用しました。

## git format-patch

`git format-patch` コマンドは、一連のパッチをmbox形式にフォーマットし、適切にフォーマットされた形式でメーリングリストへ送信できるようにします。

メールを使った公開プロジェクトへの貢献 では、`git format-patch` ツールを使用してプロジェクトへ貢献する例を見てきました。

## git imap-send

`git imap-send` コマンドを使うと、`git format-patch` コマンドによって生成された mailbox ファイルをIMAP サーバのドラフトフォルダにアップロードしてくれます。

プロジェクトへの貢献方法として、`git imap-send` を使ってパッチを送る例を メールを使った公開プロジェクトへの貢献 で紹介しています。

### git send-email

`git send-email` コマンドは、`git format-patch` コマンドで生成したパッチをメールで送信する際に使用されます。

メールを使った公開プロジェクトへの貢献 では、`git send-email` ツールを使用して、パッチを送信してプロジェクトへ貢献する例を見てきました。

### git request-pull

`git request-pull` コマンドは、単に誰かへのメールの本文の例を生成するのに使用されます。公開サーバにブランチがあり、メールでパッチを送信することなしにその変更点を取り込んでもらう方法を誰かに知ってもらいたい場合、このコマンドを実行して、変更点を取り込んでもらいたい人にその出力を送ることができます。

フォークされた公開プロジェクト では、`git request-pull` を使用してプルメッセージを生成する実例を示しました。

## 外部システム

Gitには、他のバージョン管理システムと連携するためのコマンドがいくつか付属しています。

### git svn

`git svn` コマンドは、Subversionバージョン管理システムに対して、クライアントとして通信をするのに使用されます。これは、Subversionサーバに対してチェックアウトしたりコミットしたりするのにGitを使用できることを意味しています。

Git と Subversion では、このコマンドについて詳細に取り上げました。

### git fast-import

他のバージョン管理システム、または他の任意のフォーマットからのインポートにおいては、`git fast-import` を使用して、他のフォーマットをGitが容易に記録できるフォーマットへ対応付けることができます。

A Custom Importer では、このコマンドについて詳細に取り上げました。

## システム管理

Gitリポジトリのシステム管理をしていたり、大々的に何かを修正したい場合、Gitにはそれを支援するシステム管理用コマンドがいくつかあります。

### git gc

`git gc` コマンドは、リポジトリ上で “ガベージコレクション” を実行し、データベース上の不要なファイルを削除するとともに、残ったファイルをより効率的なフォーマットへ詰め込み直します。

このコマンドは通常バックグラウンドで自動的に実行されますが、お望みなら手動で実行することもできます。メンテナンス では、このコマンドのいくつかの例を見てきました。

### git fsck

`git fsck` コマンドは、内部データベースに問題や不整合がないかチェックするのに使用されます。

このコマンドは、データリカバリ で宙ぶらりんのオブジェクトを検索する際に一度使用しただけです。

### git reflog

`git reflog` コマンドは、歴史を書き換える際に失われた可能性のあるコミットを探すため、ブランチのすべてのHEADがあった場所のログを見ていきます。

このコマンドは主に 参照ログの短縮形 で取り上げました。通常の使い方と、`git log -g` を使用して `git log` の出力で同じ情報を見る方法を示しました。

データリカバリ では、失ったブランチの回復など実用的な例を見てきました。

### git filter-branch

`git filter-branch` コマンドは、大量のコミットを、特定のパターンに従って書き換える際に使用されます。例えば、あるファイルを全てのコミットから削除する場合や、プロジェクトを抽出するためにリポジトリ全体を単一のサブディレクトリへフィルタリングする場合に使用します。

全コミットからのファイルの削除 では、このコマンドの説明を行いました。また、`--commit-filter`、`--subdirectory-filter`、`--tree-filter` などいくつかのオプションの使い方を見てきました。

Git-p4 および TFS では、インポートした外部リポジトリの修正にこのコマンドを使用しました。

## 配管コマンド

本書では、低レベルな配管コマンドが何度も出てきています。

最初に `ls-remote` コマンドが出てきたのは プルリクエストの参照 でした。ここでは、サーバ上の生の参照を見るためにこのコマンドを使用しました。

マージの手動再実行、Rerere および インデックス では、`ls-files` を使用して、ステージングエリアがどのようになっているかをより生に近い状態で見てきました。

ブランチの参照 では `rev-parse` について言及しました。ここでは、任意の文字列を受け取ってオブジェクトのSHA-1へ変換するのに使用しました。

しかし、低レベルの配管コマンドのほとんどは、多かれ少なかれ、そこにフォーカスした章である Gitの内側 で取り上げられています。 また本書の他の大部分では、これらのコマンドを使用しないように努めました。