

Pro Git

目次

1. 使い始める

1. バージョン管理に関して
2. Git 略史
3. Git の基本
4. Git のインストール
5. 最初のGitの構成
6. ヘルプを見る
7. おわりに

2. Git の基本

1. Git リポジトリの取得
2. 変更内容のリポジトリへの記録
3. コミット履歴の閲覧
4. 作業のやり直し
5. リモートでの作業
6. タグ
7. ヒントと裏技
8. まとめ

3. Git のブランチ機能

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

4. Git サーバ

1. プロトコル
2. サーバ用の Git の取得
3. SSH 公開鍵の作成
4. サーバのセットアップ
5. 一般公開
6. GitWeb
7. Gitis
8. Git デーモン
9. Git のホスティング
10. まとめ

5. Git での分散作業

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

6. Git のさまざまなツール

1. リビジョンの選択
2. Interactive Staging
3. Stashing
4. Rewriting History
5. Debugging with Git
6. Submodules
7. Subtree Merging
8. Summary

7. Gitの内側

1. Plumbing and Porcelain
2. Git Objects
3. Git References
4. Packfiles
5. The Refspec
6. Transfer Protocols
7. Maintenance and Data Recovery
8. Summary

使い始める

この章は、Gitを使い始めることにに関してになります。まずはバージョン管理システムの背景に触れ、その後にGitをあなたのシステムで動かす方法、そして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が広まっているか、なぜGitを使うべきなのか、それをするための準備が全て整っているだろうということを、あなたはきっと理解しているでしょう。

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、変更を一つのファイル、もしくは時間を通じたファイルの集合に記録するシステムで、そのため後で特定バージョンを呼び出すことができます。現実にはコンピュータ上のほとんどあらゆるファイルのタイプでバージョン管理を行なう事ができますが、本書の中の例では、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利用します。

もしあなたが、グラフィックス・デザイナー、もしくはウェブ・デザイナーであって、（あなたが最も確実に望んでいるであろう）画像もしくはレイアウトの全てのバージョンを管理したいのであれば、バージョン管理システム（VCS）はとても賢く利用できるものです。それは、ファイルを以前の状態まで戻し、プロジェクト丸ごとを以前の状態に戻し、時間を通じた変化を比較し、誰が最後に問題を引き起こすだろう何かを修正したか、誰が、何時、課題を導入したかを知り、それ以上のことを可能にします。VCSを使うということはまた、一般的に、何かをもみくちゃにするか、ファイルを失うとしても、簡単に復活させることができることを意味します。加えて、とても僅かな諸経費で、それら全てを得ることができます。

ローカル・バージョン管理システム

多くの人々の選り抜きのバージョン管理手法は、他のディレクトリ（もし彼らが賢いのであれば、恐らく日時が書かれたディレクトリ）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、とても単純なためにすごく一般的ですが、信じられない間違い傾向もあります。どのディレクトリにいるのか忘れやすいですし、偶然に間違ったファイルに書き込んだり、意図しないファイルに上書きしたりします。

この問題を扱うため、大昔にプログラマは、バージョン管理下で全ての変更をファイルに保持するシンプルなデータベースを持つ、ローカルなバージョン管理システムを開発しました（図1-1参照）。

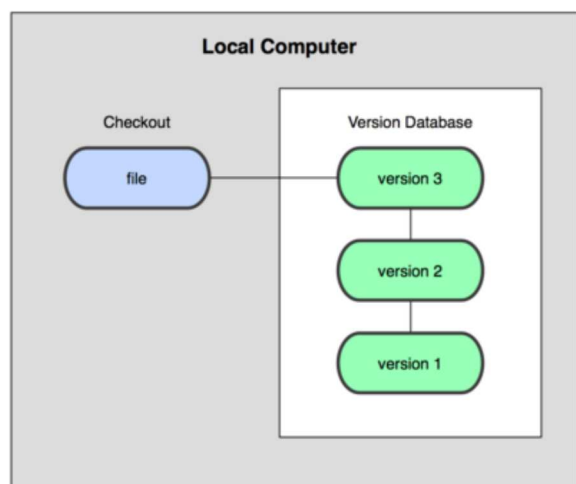


図1-1. ローカル・バージョン管理図解

もっとも有名なVCSツールの一つが、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・システムさえも、開発者ツールをインストールしたときは、rcsコマンドを含みます。このツールは基本的に、ディスク上に特殊フォーマットで、一つの変更からもう一つの変更へのパッチ（これはファイル間の差分です）の集合を保持することで稼動します。そういうわけで、全てのパッチを積み上げることで、いつかは、あらゆる時点の、あらゆるファイルのように見えるものを再生成する事ができます。

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、他のシステムの開発者と共同制作をする必要があることです。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（CVCSs）が開発されました。CVSやSubversion、Perforceのような、これらのシステムは、全てのバージョン管理されたファイルと、その中央の場所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を含む単一のサーバーを持ちます。長年の間、これはバージョン管理の標準となって来ました（図1-2参照）。

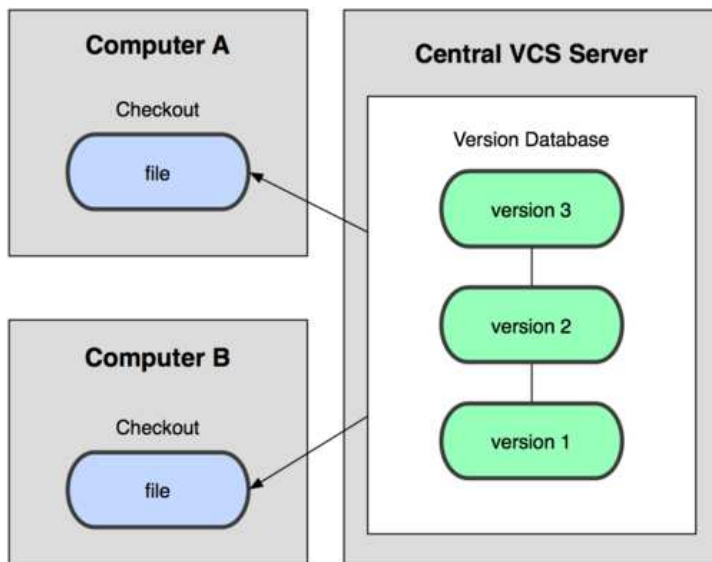


図1-2. 集中バージョン管理図解

この構成は、特にローカルVCSと比較して、多くの利点を提供します。例えば、全ての人は、プロジェクトのその他の全ての人が何をしているのか、一定の程度は知っています。管理者は、誰が何をできるのかについて、きめ細かい統制手段を持ちます。このため、一つのCVCSを管理するという事は、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、はるかに容易です。

しかしながら、この構成はまた、深刻な不利益も持ちます。もっとも明白なのは、中央サーバーで発生する単一障害点です。もし、そのサーバーが1時間の間停止すると、その1時間の間は誰も全く、共同作業や、彼らが作業を進めている全てに対してバージョン変更の保存をすることができなくなります。もし中央データベースがのってい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いとすると、人々が偶然にローカル・マシンに持ってい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）を除いた、プロジェクト全体の履歴を失うことになります。ローカルVCSシステムも、これと同じ問題に悩まされま

分散バージョン管理システム

ここから分散バージョン管理システム(DVCS)に入ります。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、更新履歴など一群）全体をミラーリングします。故にどのサーバーが故障したとして、故障したサーバーを介してそれらのDVCSが共同作業をしていたとしても、あらゆるクライアント・リポジトリは修復のためにサーバーにコピーして戻す事ができます。そのサーバーを介してコラボレーションしていたシステムは、どれか一つのクライアントのリポジトリからサーバー復旧の為バックアップをコピーすることができます。全てのチェックアウトは、実は全データの完全バックアップなのです(図1-3を参照)。

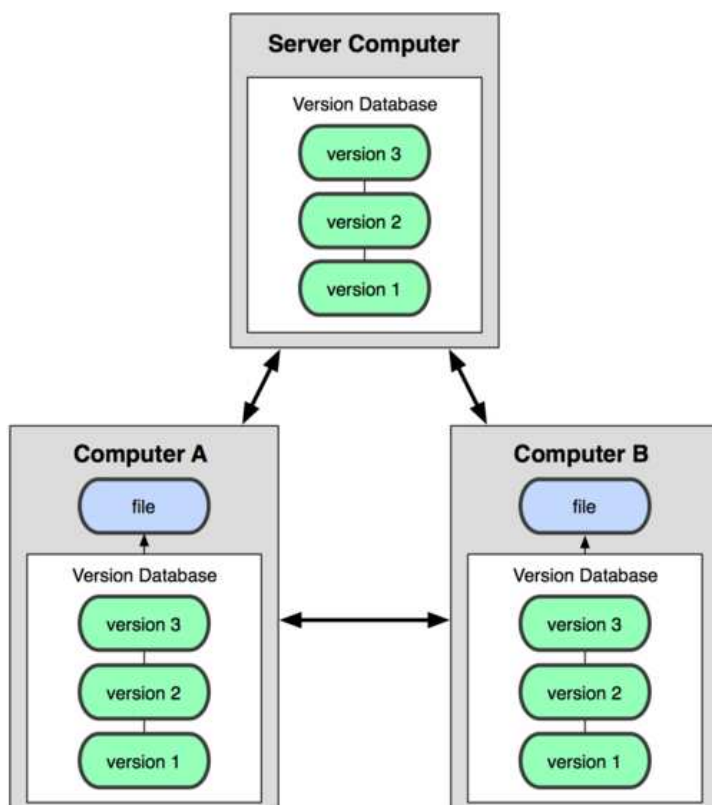


図1-3. 分散バージョン管理システムの図解

そのうえ、これらのDVCSの多くは、連携する複数のリモート・リポジトリを扱いながら大変よく機能するため、同一のプロジェクト内において、同時に異なった方法で、異なる人々のグループと共同作業が可能です。このことは、集中システムでは不可能であった階層モデルのような、幾つかの様式のワークフローを始めることを許します。

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な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（第3章参照）。

Gitの基本

では、要するにGitとは何なのでしょう。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がどうやって稼働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VCSs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。Gitは、ユーザー・インターフェイスがとてもよく似ているのにも関わらず、それら他のシステムとは大きく異なって、情報を格納して取り扱います（訳者注：「取り扱う」の部分はthinksなので、「見なします」と訳す方が原語に近い）。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。

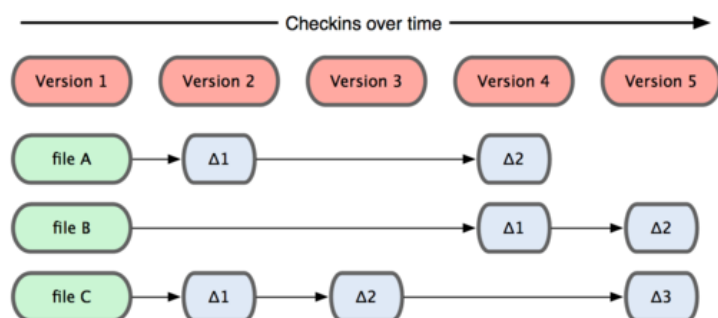


図1-4. 他のシステムは、データをそれぞれのファイルの基本バージョンへの変更として格納する傾向があります。

Gitは、この方法ではデータを考えたり、格納しません。代わりに、Gitはデータをミニ・ファイルシステムのスナップショットの集合のように考えます。Gitで全てのコミット（訳注：commitとは変更を記録・保存するGitの操作。詳細は後の章を参照）をするとき、もしくはプロジェクトの状態を保存するとき、Gitは基本的に、その時の全てのファイルの状態のスナップショットを撮り（訳者注：意識）、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、むしろデータを図1-5のように考えます。

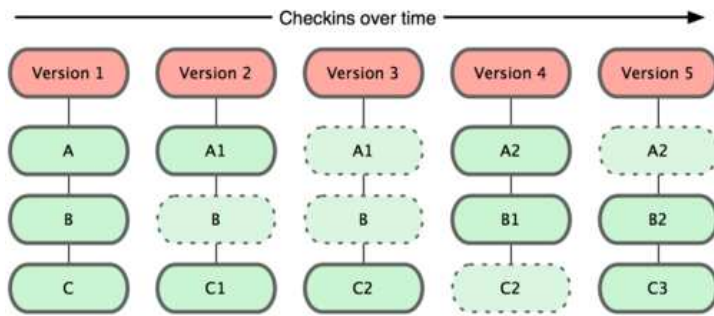


図1-5. Gitは時間を通じたプロジェクトのスナップショットとしてデータを格納します。

これが、Gitと類似の全ての他のVCSs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を意識）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、第3章のGit branching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アドレスが呼び出される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、大変難しいです。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や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、第9章の"Under the Covers"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。

Local Operations

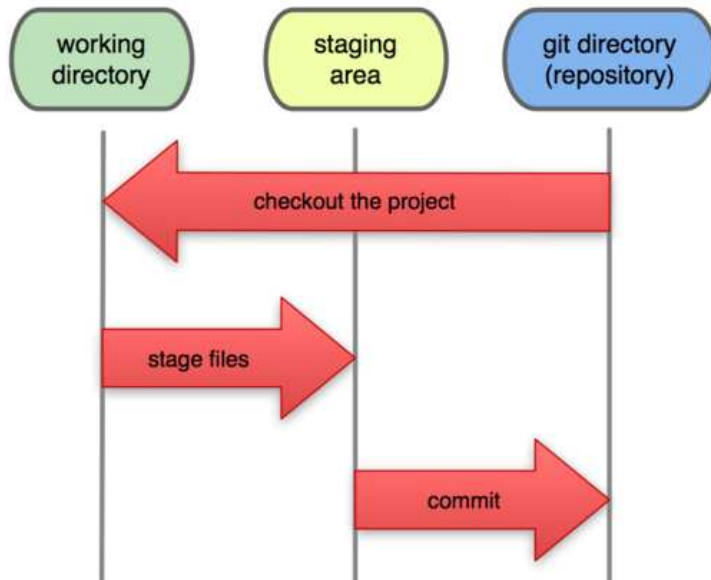


図1-6. 作業ディレクトリ、ステージング・エリア、Gitディレクトリ

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

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

ステージング・エリアは、普通はGitディレクトリに含まれる、次のコミットに何が含まれるかに関する情報を蓄えた一つの単純なファイルです。ときどきインデックスのように引き合いにだされますが、ステージング・エリアと呼ばれることが基本になりつつあります。

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。第2章では、これらの状態と、どうやってこれらを利用するか、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。

Gitのインストール

少しGitを使う事に入りましょう。何よりも最初に、Gitをインストールしなければなりません。幾つもの経路で入手することができ、主要な二つの方法のうちの一つはソースからインストールすることで、もう一つはプラットフォームに応じて存在するパッケージをインストールすることです。

ソースからのインストール

もし可能であれば、もっとも最新のバージョンを入手できるので、一般的にソースからGitをインストールするのが便利です。Gitのそれぞれのバージョンは、実用的なユーザー・インターフェイスの向上が含まれており、もしソースからソフトウェアをコンパイルすることに違和感を感じないのであれば、最新バージョンを入手することは、大抵は最も良い経路になります。また、多くのLinuxディストリビューションがとても古いパッケージを収録している事は良くあることであり、最新のディストリビューションを使っているか、バックポート（訳者注：最新のパッケージを古いディストリビューションで使えるようにする事）をしていない限りは、ソースからのインストールがベストな選択になるでしょう。

Gitをインストールするためには、Gitが依存するライブラリーである、curl、zlib、openssl、expat、libiconvを入手する必要があります。例えば、もし（Fedoraなどで）yumか（Debianベースのシステムなどで）apt-getが入ったシステムを使っているのであれば、これらのコマンドの一つを依存対象の全てをインストールするのに使う事ができます：

```
$ yum install curl-devel expat-devel gettext-devel \
    openssl-devel zlib-devel

$ apt-get install libcurl4-gnutls-dev libexpat1-dev gettext \
    libz-dev libssl-dev
```

全ての必要な依存対象を持っているのであれば、先に進んでGitのウェブサイトから最新版のスナップショットを持ってくる事ができます：

```
http://git-scm.com/download
```

そして、コンパイルしてインストールします：

```
$ tar -zxf git-1.6.0.5.tar.gz
$ cd git-1.6.0.5
$ make prefix=/usr/local all
$ sudo make prefix=/usr/local install
```

また、Gitのインストール後、アップデートでGitを通して最新版のGitを得ることができます。

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

Linuxにインストール

バイナリのインストーラーを通じてLinux上にGit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：

```
$ yum install git-core
```

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

```
$ apt-get install git-core
```

Macにインストール

MacにGitをインストールするには2つの簡単な方法があります。もっとも簡単な方法は、グラフィカルなGitインストーラーを使うことで、このGitインストーラーはGoogle Codeのページ（図1-7参照）からダウンロードすることができます：

```
http://code.google.com/p/git-osx-installer
```

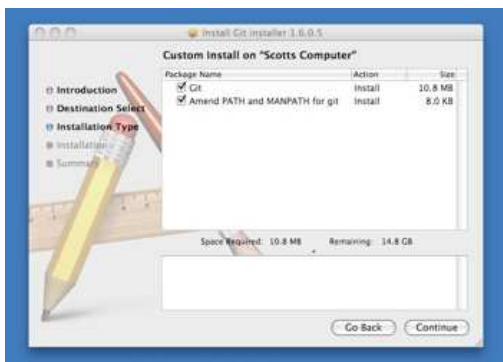


Figure 1-7. Git OS X installer

もう一つの主要な方法は、MacPorts (<http://www.macports.org>) からGitをインストールすることです。MacPortsをインストールした状態であれば、Gitを以下のようにインストールできます。

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

全てのvariantsを追加する必要はありませんが、SubversionのリポジトリでGitを使う必要がまだあるなら、恐らく+svnを含めないといけなでしょう（第8章参照）。

Windowsにインストール

WindowsにGitをインストールするのはとても簡単です。msysGitプロジェクトは、より簡単なインストール手続きの一つを備えています。Google Codeのページから、単純にインストーラーのexeファイルをダウンロードをし、実行してください：

```
http://code.google.com/p/msysgit
```

インストール後、コマンドライン版（後で役に立つSSHクライアントを含む）とスタンダードGUI版の両方を使う事ができます。

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。アップグレードの度についてまわるので、たった一度でそれらを終わらすべきでしょう。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り、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：

- /etc/gitconfig file: 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。
- ~/.gitconfig file: 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。
- 現在使っている、あらゆるリポジトリのGitディレクトリの設定ファイル(.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環境下では、Gitは\$HOMEディレクトリ（ほとんどのユーザーはC:\Documents and Settings\%\$USER）（訳者注：環境変数USERPROFILEで指定される）の中の.gitconfigファイルを検索に行きます。また、インストーラー時にWindowsシステムにGitをインストールすると決めたところにある、MSysのルートとの相対位置であったとしても、/etc/gitconfigも見に行きます。

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E-mail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、次々とまわす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E-mail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。

エディター

今や、個人の識別情報が設定され、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。標準では、Gitはシステムのデフォルト・エディターを使います。これは大抵の場合、ViかVimです。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
```

diffツール

設定したいと思われる、その他の便利なオプションは、マージ（訳者注：複数のリポジトリを併合すること）時の衝突を解決するために使う、標準のdiffツールです。vimdiffを使いたいとします：

```
$ git config --global merge.tool vimdiff
```


Gitはkdiff3、tkdiff、meld、xxdiff、emerge、vimdiff、gvimdiff、ecmerge、opendiffを確かなマージ・ツールとして扱えます。カスタム・ツールもまた設定できますが、これをする事に関しての詳細な情報は第7章を参照してください。

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=Scott Chacon
user.email=schacon@gmail.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}`をタイプすることで確認することができます：

```
$ git config user.name
Scott Chacon
```

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、素晴らしいです。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、全員が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、システムに移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、第 9 章を参照ください)。

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コミットを行います。

```
$ git add *.c
$ git add README
$ 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。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、`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復元することができます (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。第 4 章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、Ruby の Git ライブラリである Grit をクローンする場合は次のようになります。

```
$ git clone git://github.com/schacon/grit.git
```

これは、まず "grit" 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 `grit` ディレクトリに入ると、プロジェクトのファイルをぐらんいただけます。もし `grit` 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 git://github.com/schacon/grit.git mygrit
```

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 `mygrit` 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 `git://` プロトコルを使用しましたが、`http(s)://` や `user@server:/path.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。第 4 章で、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「追跡されていない(untracked)」ものの二通りがあることを覚えておきましょう。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。ここまでの流れを図 2-1 にまとめました。

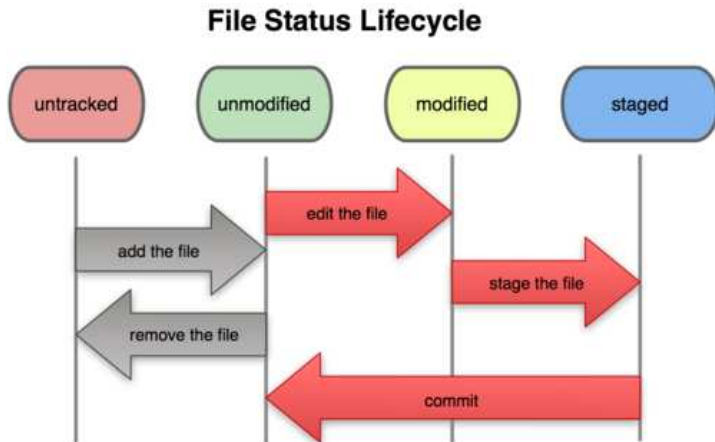


図 2-1. ファイルの状態の流れ

ファイルの状態の確認

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

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

これは、クリーンな作業コピーである（つまり、追跡されているファイルの中に変更されているものがない）ことを意味します。また、追跡されていないファイルも存在しません（もし追跡されていないファイルがあれば、Git はそれを表示します）。最後に、このコマンドを実行するとあなたが今どのブランチにいるのかを知ることができます。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次の章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。

```
$ vim 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り、ステージされ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 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 benchmarks.rb を変更して status コマンドを実行すると、結果はこうになります。

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#   new file:   README
#
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#
#   modified:   benchmarks.rb
#
```

benchmarks.rb ファイルは“Changed but not updated”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（このコマンド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）。では、git add で benchmarks.rb 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 benchmarks.rb
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#   new file:   README
#   modified:   benchmarks.rb
#
```

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 benchmarks.rb にちょっとした変更を加えてからコミットしたくなつたとしましょう。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 benchmarks.rb
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#   new file:   README
#   modified:   benchmarks.rb
#
```

```
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#
#   modified:   benchmarks.rb
#
```

これはどうということでしょうか？ benchmarks.rb が、ステージされているほうにもステージされていないほうにも登場しています。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 benchmarks.rb 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 benchmarks.rb
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#   new file:   README
#   modified:   benchmarks.rb
#
```

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る。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
*.log
*~
```

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えて 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
- 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。

では、.gitignore ファイルの例をもうひとつ見てみましょう。

```
# コメント。これは無視されます
*.a      # .a ファイルは無視
!lib.a   # しかし、lib.a ファイルだけは .a であっても追跡対象とします
/TODO    # ルートディレクトリの TODO ファイルだけを無視し、サブディレクトリの TODO は無視しません
build/   # build/ ディレクトリのすべてのファイルは無視します
doc/*.txt # doc/notes.txt は無視しますが、doc/server/arch.txt は無視しません
```

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?」もちろん git status でも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 benchmarks.rb 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 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)
#
#   new file:   README
#
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#
#   modified:   benchmarks.rb
#
```

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/benchmarks.rb b/benchmarks.rb
index 3cb747f..da65585 100644
--- a/benchmarks.rb
+++ b/benchmarks.rb
@@ -36,6 +36,10 @@ def main
     @commit.parents[0].parents[0].parents[0]
   end

+  run_code(x, 'commits 1') do
+    git.commits.size
+  end
+
   run_code(x, 'commits 2') do
     log = git.commits('master', 15)
     log.size
```

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 --cached を使用します (Git バージョン 1.6.1 以降では git diff --staged も使えます。こちらのほうが覚えやすいでしょう)。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 --cached
diff --git a/README b/README
new file mode 100644
index 0000000..03902a1
--- /dev/null
+++ b/README2
@@ -0,0 +1,5 @@
+grit
+ by Tom Preston-Werner, Chris Wanstrath
+ http://github.com/mojombo/grit
+
+Grit is a Ruby library for extracting information from a Git repository
```

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。benchmarks.rb 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。

```
$ git add benchmarks.rb
$ echo '# test line' >> benchmarks.rb
$ git status
```



```
# On branch master
#
# Changes to be committed:
#
#   modified:   benchmarks.rb
#
# Changed but not updated:
#
#   modified:   benchmarks.rb
#
```

ここで `git diff` を使うと、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/benchmarks.rb b/benchmarks.rb
index e445e28..86b2f7c 100644
--- a/benchmarks.rb
+++ b/benchmarks.rb
@@ -127,3 +127,4 @@ end
main()

##pp Grit::GitRuby.cache_client.stats
+# 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。

```
$ git diff --cached
diff --git a/benchmarks.rb b/benchmarks.rb
index 3cb747f..e445e28 100644
--- a/benchmarks.rb
+++ b/benchmarks.rb
@@ -36,6 +36,10 @@ def main
  @commit.parents[0].parents[0].parents[0]
end

+  run_code(x, 'commits 1') do
+    git.commits.size
+  end
+
  run_code(x, 'commits 2') do
    log = git.commits('master', 15)
    log.size
```

変更のコミット

ステージングエリアの準備ができたなら、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デディスク上に残ります。今回の場合は、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ています。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、それ以外にも第 1 章で説明した `git config --global core.editor` コマンドで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:
#   (use "git reset HEAD <file>..." to unstage)
```

```
#
#      new file:   README
#      modified:  benchmarks.rb
~
~
~
".git/COMMIT_EDITMSG" 10L, 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行ったの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 はそのメッセージ付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` コマンドの後に `-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]: created 463dc4f: "Fix benchmarks for speed"
 2 files changed, 3 insertions(+), 0 deletions(-)
 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
#
# Changed but not updated:
#
#   modified:   benchmarks.rb
#
$ git commit -a -m 'added new benchmarks'
[master 83e38c7] added new benchmarks
 1 files changed, 5 insertions(+), 0 deletions(-)
```

この場合、コミットする前に `benchmarks.rb` を `git add` する必要がないことに注意しましょう。

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は残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 grit.gemspec
$ git status
# On branch master
#
# Changed but not updated:
#   (use "git add/rm <file>..." to update what will be committed)
#
```

```
#      deleted:    grit.gemspec
#
```

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

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` ファイルが追加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.txt
```

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が、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 `status` 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.txt README
$ git status
# On branch master
# Your branch is ahead of 'origin/master' by 1 commit.
#
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#      renamed:    README.txt -> 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.txt README
$ git rm README.txt
$ 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付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、私が説明用によく用いているプロジェクトで、次のようにして取得できます。

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

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 code

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パ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、よく用いられる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、これは各コミットの diff 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.version = "0.1.0"
+   s.version = "0.1.1"
   s.author = "Scott Chacon"

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

    removed unnecessary test code

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 files changed, 1 insertions(+), 1 deletions(-)

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

    removed unnecessary test code

lib/simplegit.rb | 5 ----
1 files changed, 0 insertions(+), 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(+), 0 deletions(-)

```

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 code
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, 11 months ago : changed the version number
085bb3b - Scott Chacon, 11 months ago : removed unnecessary test code
a11bef0 - Scott Chacon, 11 months ago : first commit
```

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

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

表: 書式オプション

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に行った人、*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第 5 章でも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、ちょっとした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。Grit プロジェクトのリポジトリならこのようになります。

```
$ 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。表 2-2 に、今まで取り上げたオプションとそれ以外によく使われるオプション、そしてそれぞれがログの出力をどのように変更のかをまとめました。

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

表: `git log` のオプション

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 を両方指定したい場合は --all-match を追加しないとはいけません。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)。

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。

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

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 指定した日付以降のコミットのみに制限する
- until, --before 指定した日付以前の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

表: git log のフィルタオプション

たとえば、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（マージは除く）で 2008 年 10 月に行われたものを知りたい場合に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 attribute
acd3b9e - Enhance hold_lock_file_for_{update,append}()
f563754 - demonstrate breakage of detached checkout wi
d1a43f2 - reset --hard/read-tree --reset -u: remove un
51a94af - Fix "checkout --track -b newbranch" on detac
b0ad11e - pull: allow "git pull origin $something:$cur
```

約 20,000 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。

GUI による歴史の可視化

もう少しグラフィカルなツールでコミットの歴史を見たい場合は、Tcl/Tk のプログラムである gitk を見てみましょう。これは Git に同梱されています。gitk は、簡単に言うとビジュアルな git log ツールです。git log で使えるフィルタリングオプションにはほぼすべて対応しています。プロジェクトのコマンドラインで gitk と打ち込むと、図 2-2 のような画面があらわれるでしょう。

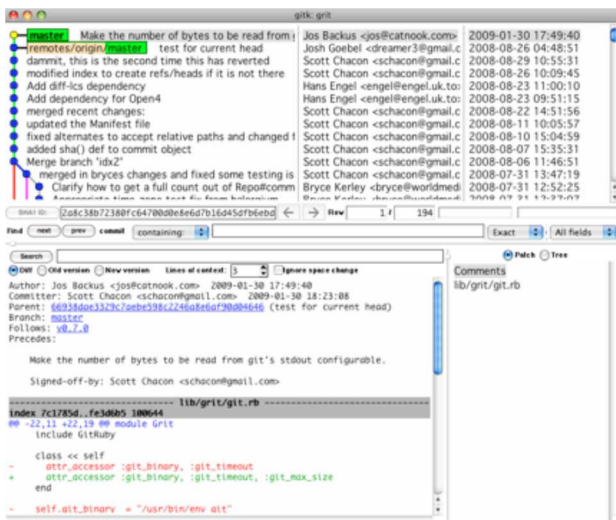


図 2-2. gitk history visualizer

ウィンドウの上半分に、コミットの歴史がきれいな家系図とともに表示されます。ウィンドウの下半分には diff ビューアがあり、任意のコミットをクリックしてその変更内容を確認することができます。

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。注意しなければならないのは、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。

直近のコミットの変更

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 '初期コミット'
$ git add 忘れてたファイル
$ git commit --amend
```

この 3 つのコマンドの結果、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 commit コマンドが、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。ふたつのうち的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)
#
#       modified:   README.txt
#       modified:   benchmarks.rb
#
```

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

```
$ git reset HEAD benchmarks.rb
benchmarks.rb: locally modified
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   README.txt
#
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   benchmarks.rb
#
```

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。benchmarks.rb 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。

ファイルへの変更の取り消し

benchmarks.rb 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、まともな 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。

```
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   benchmarks.rb
#
```

とても明確に、変更を取り消す方法が書かれています（少なくとも、バージョン 1.6.1 以降の新しい Git ではこのようになります。もし古いバージョンを使用しているのなら、アップグレードしてこのすばらしい機能を活用することをおすすめします）。ではそのとおりにし

てみましょう。

```
$ git checkout -- benchmarks.rb
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   README.txt
#
```

変更が取り消されたことがわかります。また、これが危険なコマンドであることも知っておかねばなりません。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。単にファイルを片付けたいだけなら、次の章で説明する 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第 9 章を参照ください)。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たり読み書き可能にした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 git://github.com/schacon/ticgit.git
Initialized empty Git repository in /private/tmp/ticgit/.git/
remote: Counting objects: 595, done.
remote: Compressing objects: 100% (269/269), done.
remote: Total 595 (delta 255), reused 589 (delta 253)
Receiving objects: 100% (595/595), 73.31 KiB | 1 KiB/s, done.
Resolving deltas: 100% (255/255), done.
$ cd ticgit
$ git remote
origin
```

-v を指定すると、その名前に対応する URL を表示します。

```
$ git remote -v
origin  git://github.com/schacon/ticgit.git
```

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、私の Grit リポジトリの場合はこのようになっています。

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

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。ここで、origin リモートだけが SSH の URL であることに注目しましょう。私がプッシュできるのは origin だけだということになります (なぜそうなるのかについては第 4 章で説明しま

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 git://github.com/paulboone/ticgit.git
$ git remote -v
origin  git://github.com/schacon/ticgit.git
pb      git://github.com/paulboone/ticgit.git
```

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: 58, done.
remote: Compressing objects: 100% (41/41), done.
remote: Total 44 (delta 24), reused 1 (delta 0)
Unpacking objects: 100% (44/44), done.
From git://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。

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 (ブランチとは何なのか、どのように使うのかについては、第 3 章でより詳しく説明します)。

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、`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 (次のセクションと第 3 章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。またデフォルトで、`git clone` コマンドはローカルの master ブランチが (取得元サーバー上の) リモートの master ブランチを追跡するよう自動設定します (リモートに master ブランチが存在することを前提としています)。`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。master ブランチの内容を origin サーバー (何度も言いますが、クローンした地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、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第 3 章を参照ください。

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
  URL: git://github.com/schacon/ticgit.git
  Remote branch merged with 'git pull' while on branch master
    master
  Tracked remote branches
    master
    ticgit
```

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「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。また、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。

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: git@github.com:defunkt/github.git
  Remote branch merged with 'git pull' while on branch issues
    issues
  Remote branch merged with 'git pull' while on branch master
    master
  New remote branches (next fetch will store in remotes/origin)
    caching
  Stale tracking branches (use 'git remote prune')
    libwalker
    walker2
  Tracked remote branches
    acl
    apiv2
    dashboard2
    issues
    master
    postgres
  Local branch pushed with 'git push'
    master:master
```

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、新しいバージョンの Git では `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
```

これは、リモートブランチ名も変更す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の参照を削除したい場合（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。一般に、この機能は (バージョン 1.0 などの) リリースポイントとして使われています。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、240 以上のタグが登録されています。その中で 1.4.2 系のタグのみを見たい場合は、このようにします。

```
$ git tag -l 'v1.4.2.*'
v1.4.2.1
v1.4.2.2
v1.4.2.3
v1.4.2.4
```

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: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Feb 9 14:45:11 2009 -0800

my version 1.4
commit 15027957951b64cf874c3557a0f3547bd83b3fff6
Merge: 4a447f7... a6b4c97...
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sun Feb 8 19:02:46 2009 -0800

Merge branch 'experiment'
```

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。

署名付きのタグ

GPG 秘密鍵を持っていれば、タグに署名をすることができます。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: "Scott Chacon <schacon@gee-mail.com>"
1024-bit DSA key, ID F721C45A, created 2009-02-09
```

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: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Feb 9 15:22:20 2009 -0800

my signed 1.5 tag
-----BEGIN PGP SIGNATURE-----
Version: GnuPG v1.4.8 (Darwin)

iEYEABECAAYFAkmQurIACgkQON3DxfchxFr5cACeIMN+ZxLKggJQf0QYiQBwgySN
Ki0An2JeAVUCAiJ70x6ZEtK+NvZAJ82/
=WryJ
-----END PGP SIGNATURE-----
commit 15027957951b64cf874c3557a0f3547bd83b3ff6
Merge: 4a447f7... a6b4c97...
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sun Feb 8 19:02:46 2009 -0800

    Merge branch 'experiment'
```

タグの署名を検証する方法については後ほど説明します。

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 15027957951b64cf874c3557a0f3547bd83b3ff6
Merge: 4a447f7... a6b4c97...
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sun Feb 8 19:02:46 2009 -0800

    Merge branch 'experiment'
```

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'
```

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'
0b7434d86859cc7b8c3d5e1dddfed66ff742fcbb 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: 50, done.
Compressing objects: 100% (38/38), done.
Writing objects: 100% (44/44), 4.56 KiB, done.
Total 44 (delta 18), reused 8 (delta 1)
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: 50, done.
Compressing objects: 100% (38/38), done.
Writing objects: 100% (44/44), 4.56 KiB, done.
Total 44 (delta 18), reused 8 (delta 1)
To git@github.com:schacon/simplegit.git
* [new tag]          v0.1 -> v0.1
* [new tag]          v1.2 -> v1.2
* [new tag]          v1.4 -> v1.4
* [new tag]          v1.4-lw -> v1.4-lw
* [new tag]          v1.5 -> v1.5
```

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。

ヒントと裏技

Git の基本を説明した本章を終える前に、ほんの少しだけヒントと裏技を披露しましょう。これを知っておけば、Git をよりシンプルかつお手軽に使えるようになり、Git になじみやすくなることでしょう。ほとんどの人はこれらのことを知らずに Git を使っています。別にどうでもいいことですが、本書の後半でこれらの技を使うわけでもないのですが、その方法ぐらいは知っておいたほうがよいでしょう。

自動補完

Bash シェルを使っているのなら、Git にはよくできた自動補完スクリプトが付属しています。Git のソースコードをダウンロードし、`contrib/completion` ディレクトリを見てみましょう。`git-completion.bash` というファイルがあるはずで、このファイルをホームディレクトリにコピーし、それを `.bashrc` ファイルに追加しましょう。

```
source ~/.git-completion.bash
```

すべてのユーザーに対して Git 用の Bash シェル補完を使わせたい場合は、Mac なら `/opt/local/etc/bash_completion.d` ディレクトリ、Linux 系なら `/etc/bash_completion.d/` ディレクトリにこのスクリプトをコピーします。Bash は、これらのディレクトリにあるスクリプトを自動的に読み込んでシェル補完を行います。

Windows で Git Bash を使用している人は、`msysGit` で Windows 版 Git をインストールした際にデフォルトでこの機能が有効になっています。

Git コマンドの入力中にタブキーを押せば、補完候補があらわれて選択できるようになります。

```
$ git co<tab><tab>
commit config
```

ここでは、`git co` と打ち込んだ後にタブキーを二度押してみました。すると `commit` と `config` という候補があらわれました。さらに `m<tab>` と入力すると、自動的に `git commit` と補完されます。

これは、コマンドのオプションに対しても機能します。おそらくこっちのほうがより有用でしょう。たとえば、`git log` を実行しようとしてそのオプションを思い出せなかった場合、タブキーを押せばどんなオプションを使えるのかがわかります。

```
$ git log --s<tab>
```

```
--shortstat --since= --src-prefix= --stat --summary
```

この裏技を使えば、ドキュメントを調べる時間を節約できるでしょう。

Git エイリアス

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ます。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か？ 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もささしくありません。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。第 1 章で説明したように、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わけではありません。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 (第 1 章で説明した SHA-1 ハッシュ) を計算し、そのパ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 'initial commit of my project'
```

git commit を実行してコミットを作成すると、Git は各サブディレクトリ (この場合はプロジェクトのルートディレクトリのみ) のチェックサムを計算し、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、メタデータおよびルートオブジェクトツリーへのポインタを含むコミットオブジェクトを作成します。これで、必要に応じてこの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。Git リポジトリ内のデータを概念図であらわすと、図 3-1 のようになります。

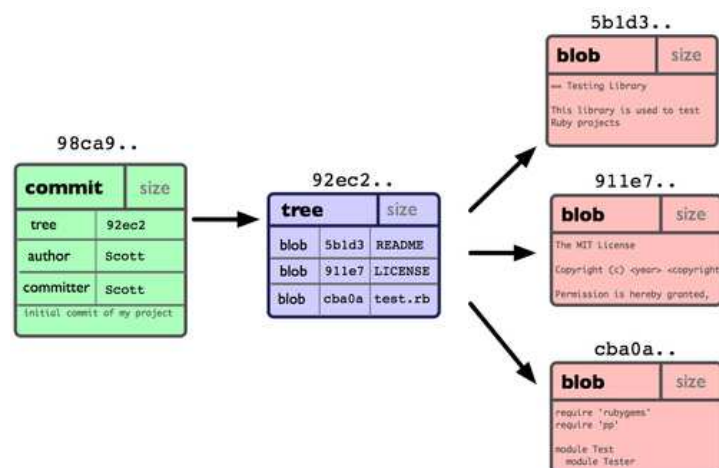


図 3-1. ひとつのコミットをあらわすリポジトリ上のデータ

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。さらに 2 回のコミットを終えた後の履歴は、図 3-2 のようになるでしょう。

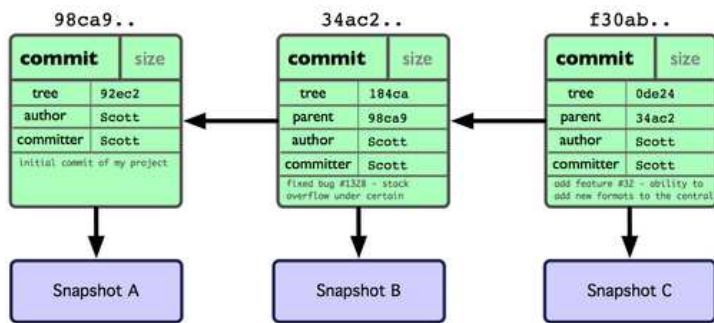


図 3-2. 複数のコミットに対応する Git オブジェクト

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の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。

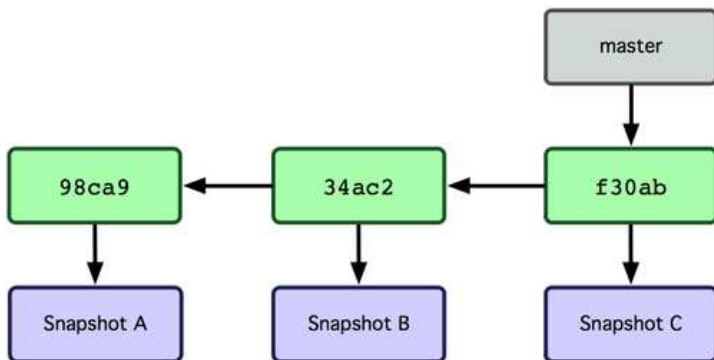


図 3-3. コミットデータの歴史を指すブランチ

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 (図 3-4 を参照ください)。

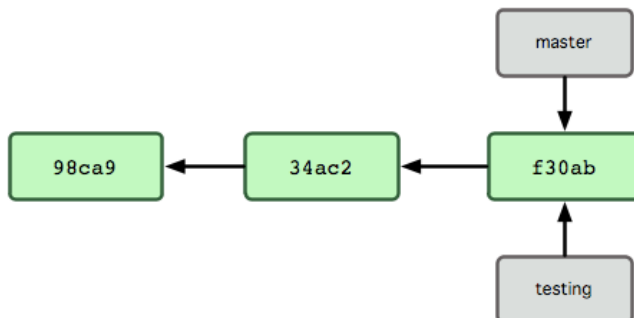


図 3-4. 複数のブランチがコミットデータの履歴を指す例

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くか？ 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 (図 3-5 を参照ください)。

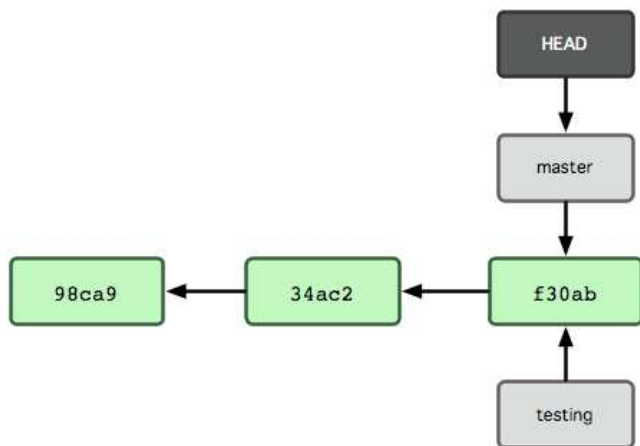


図 3-5. 現在作業中のブランチを指す HEAD

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 (図 3-6 を参照ください)。

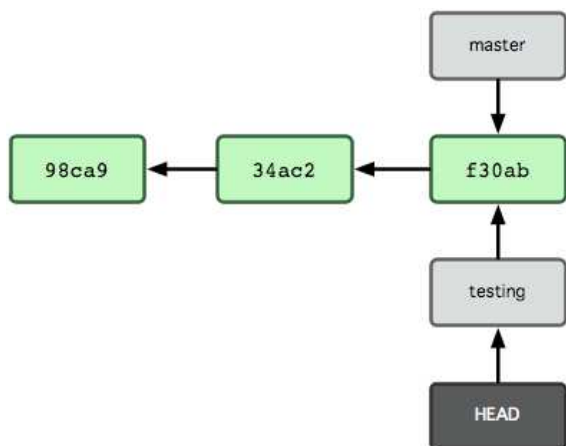


図 3-6. ブランチを切り替えると、HEAD の指す先が移動する

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'
```

図 3-7 にその結果を示します。

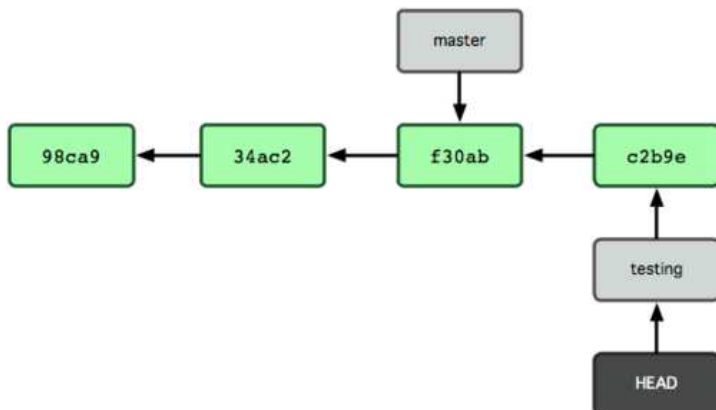


図 3-7. HEAD が指すブランチが、コミットによって移動する

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

```
$ git checkout master
```

図 3-8 にその結果を示します。

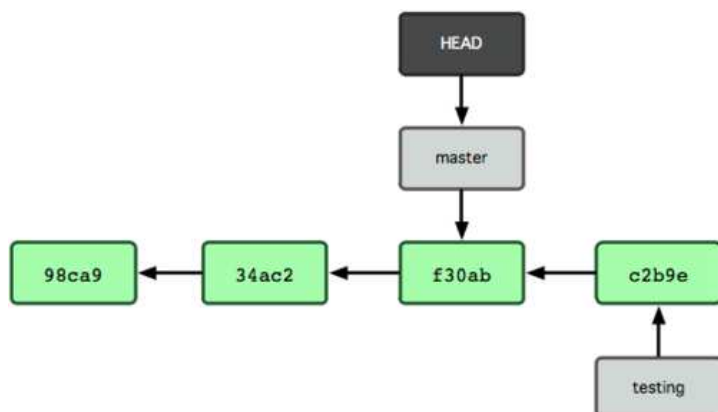


図 3-8. チェックアウトによって HEAD が別のブランチに移動する

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。

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 (図 3-9 を参照ください)。新たなブランチを作成してそちらに切り替え、何らかの作業を行い、メインブランチに戻って別の作業をした状態です。どちらの変更も、ブランチごとに分離しています。ブランチを切り替えつつそれぞれの作業を進め、必要に応じてマージすることができます。これらをすべて、シンプルに branch コマンドと checkout コマンドで行えるのです。

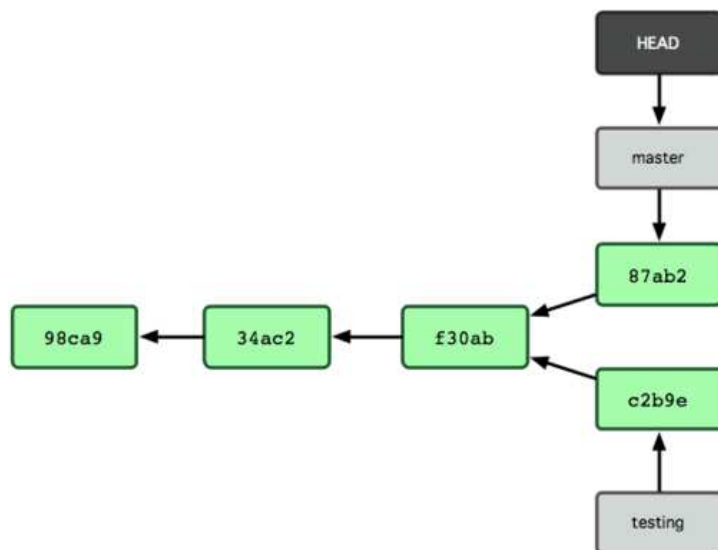


図 3-9. ブランチの歴史が分裂した

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、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 (図 3-10 を参照ください)。

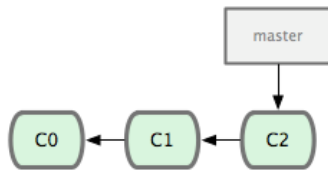


図 3-10. 短くて単純なコミットの歴史

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。念のために言っておくと、Git は何かの問題追跡システムと連動しているわけではありません。しかし、今回の作業はこの問題番号 53 に対応するものであるため、作業用に新しいブランチを作成します。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
```

図 3-11 に結果を示します。

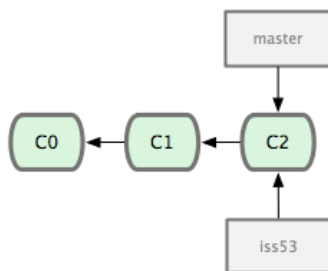


図 3-11. 新たなブランチポイントの作成

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 iss53 ブランチを指しているということです。図 3-12 を参照ください)。

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

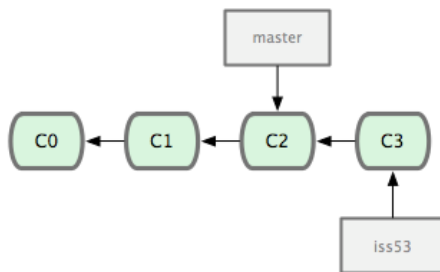


図 3-12. 作業した結果、iss53 ブランチが移動した

ここで、ウェブサイトに別の問題が発生したという連絡を受けました。そっちのほうを優先して対応する必要があるということです。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、また後ほど説明します。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、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 (図 3-13 を参照ください)。

```
$ git checkout -b 'hotfix'
Switched to a new branch "hotfix"
$ vim index.html
$ git commit -a -m 'fixed the broken email address'
[hotfix]: created 3a0874c: "fixed the broken email address"
1 files changed, 0 insertions(+), 1 deletions(-)
```

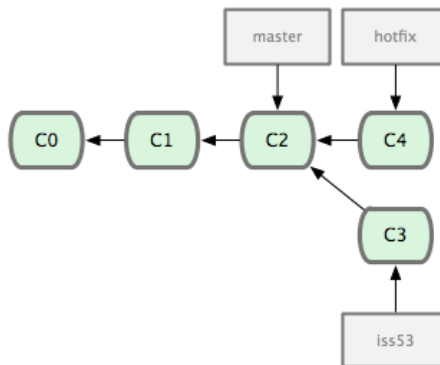


図 3-13. master ブランチから新たに作成した hotfix ブランチ

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

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 (図 3-14 を参照ください)。

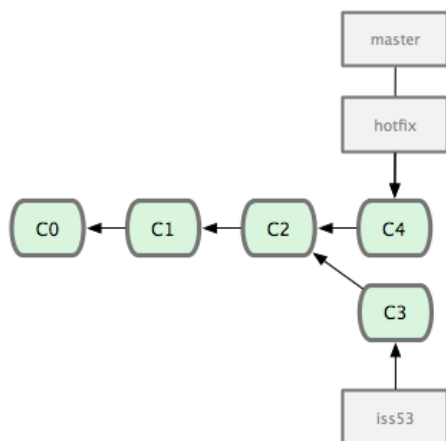


図 3-14. マージした結果、master ブランチの指す先が hotfix ブランチと同じ場所になった

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も、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 (図 3-15 を参照ください)。

```
$ git checkout iss53
Switched to branch "iss53"
$ vim index.html
$ git commit -a -m 'finished the new footer [issue 53]'
[iss53]: created ad82d7a: "finished the new footer [issue 53]"
1 files changed, 1 insertions(+), 0 deletions(-)
```

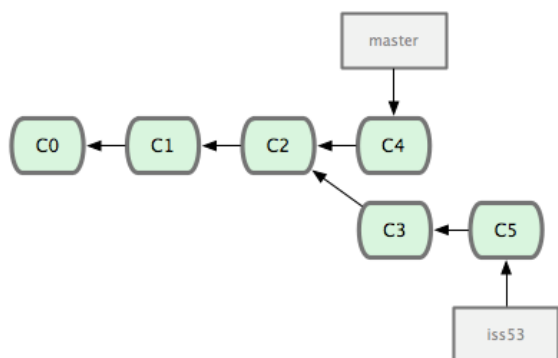


図 3-15. iss53 ブランチは独立して進めることができる

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
$ git merge iss53
Merge made by recursive.
 README | 1 +
1 files changed, 1 insertions(+), 0 deletions(-)
```

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。図 3-16 に、今回のマージで使用した三つのスナップショットを示します。

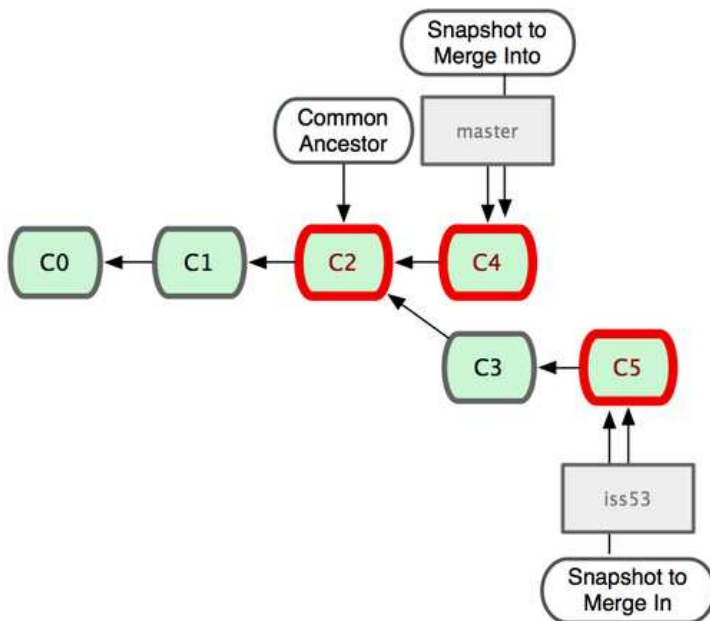


図 3-16. Git が共通の先祖を自動的に見つけ、ブランチのマージに使用する

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 (図 3-17 を参照ください)。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。

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。

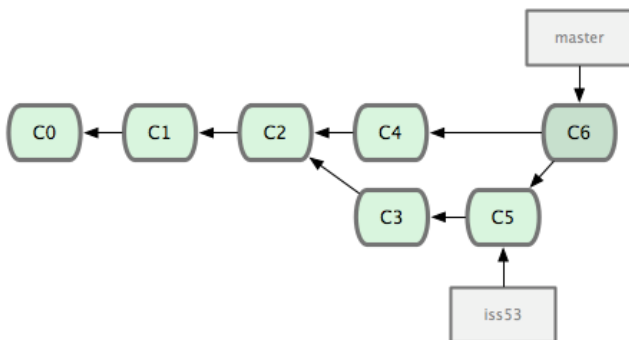


図 3-17. マージ作業の結果から、Git が自動的に新しいコミットオブジェクトを作成する

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、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。

```
[master*]$ git status
index.html: needs merge
# On branch master
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#   unmerged:   index.html
#
```

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―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なります。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
merge tool candidates: kdiff3 tkdiff xxdiff meld gvimdiff opendiff emerge vimdiff
Merging the files: index.html

Normal merge conflict for 'index.html':
  {local}: modified
  {remote}: modified
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、“merge tool candidates”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。第 7 章で、環境にあわせてこのデフォルトを変更する方法を説明します。

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 が聞いてきます。成功したと伝えらると、ファイルを自動的にステージしてコンフリクトが解決したことを示します。

再び git status を実行すると、すべてのコンフリクトが解決したことを確認できます。

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

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
```

各ブランチの状態を知るために便利なもうひとつの機能として、現在作業中のブランチにマージ済みかそうでないかによる絞り込みができるようになっています。そのための便利なオプション `--merged` と `--no-merged` が、Git バージョン 1.5.6 以降で使用可能となりました。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、もはや何も失うことはありません。

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 an ancestor of your current HEAD.
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か？ このセクションでは、気軽にブランチを切れることでこ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（図 3-18 を参照ください）。

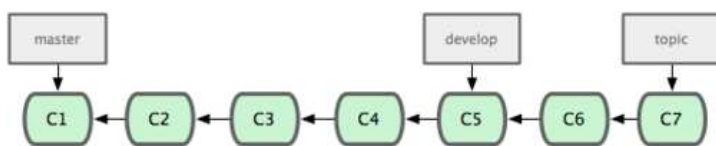


図 3-18. 安定したブランチほど、一般的にコミット履歴の奥深くに存在する

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、完全にテストを通り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（図 3-19 を参照ください）。

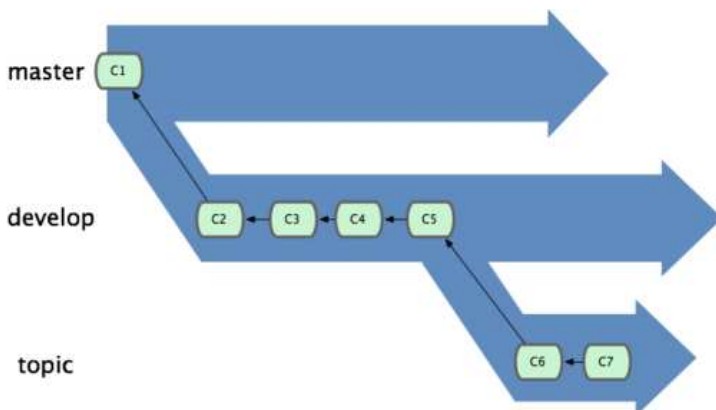


図 3-19. ブランチをサイロとして考えるとわかりやすいかも

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、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、コミットの歴史は図 3-20 のようになります。

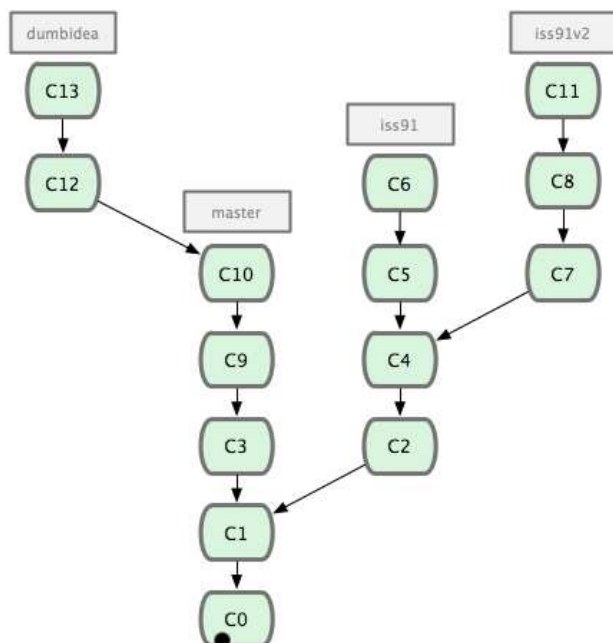


図 3-20. 複数のトピックブランチを作成した後のコミットの歴史

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、歴史は図 3-21 のようになっています。

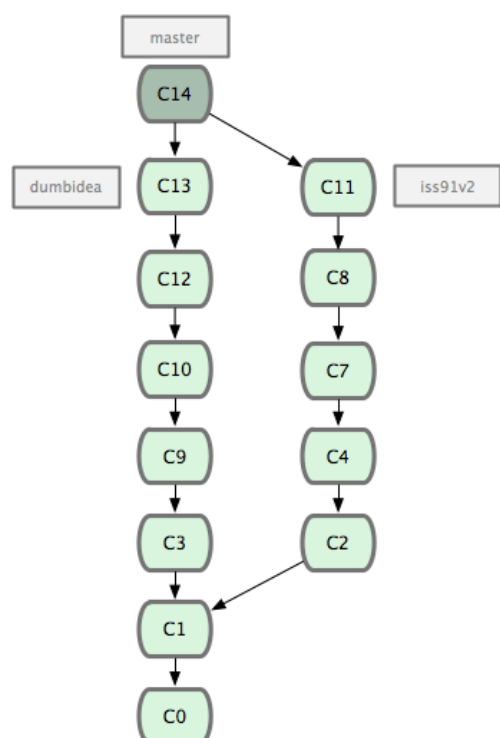


図 3-21. dumbidea と iss91v2 をマージした後の歴史

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

リモートブランチは、リモートリポジトリ上のブランチの状態を指すものです。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いければ origin/master ブランチをチェックします。誰かと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 はそれに `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。それを自分で移動させることはできません。Git はまた、`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 (図 3-22 を参照ください)。

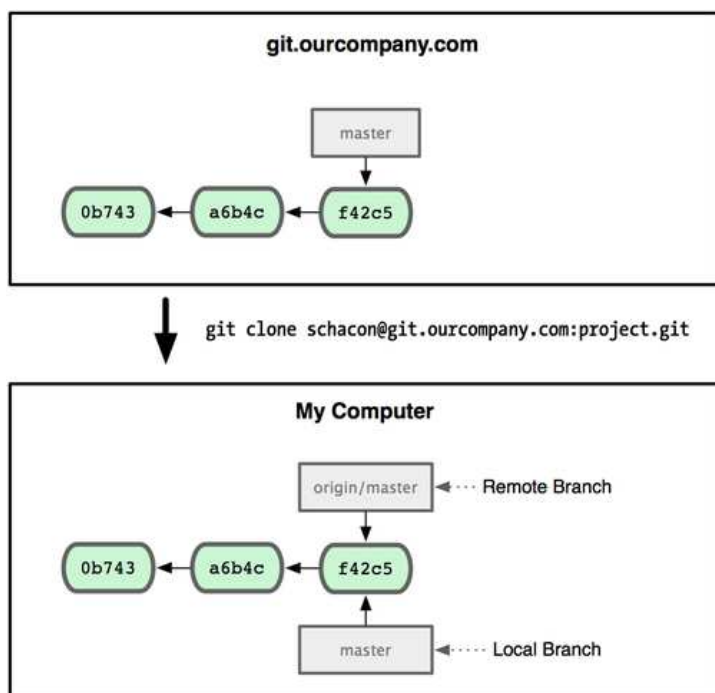


図 3-22. `git clone` により、ローカルの `master` ブランチのほかに `origin` の `master` ブランチを指す `origin/master` が作られる

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こと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 (図 3-23 を参照ください)。

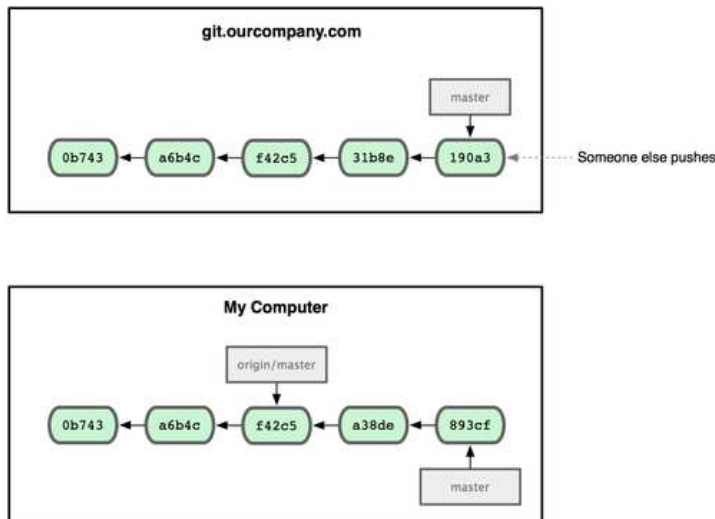


図 3-23. ローカルで作業している間に誰かがリモートサーバーにプッシュすると、両者の歴史が異なるものとなる

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、まだ手元には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 (図 3-24 を参照ください)。

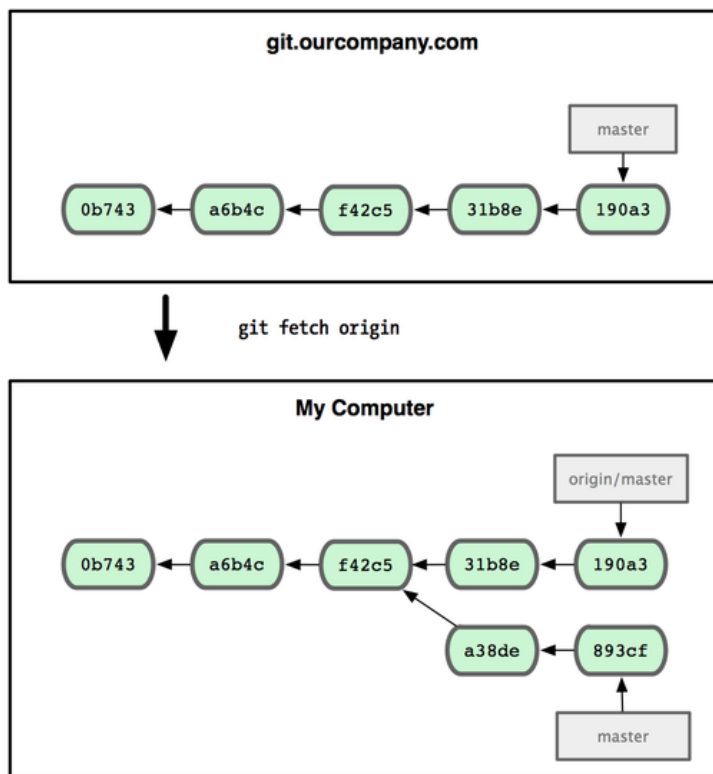


図 3-24. git fetch コマンドによるリモートへの参照の更新

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な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、第 2 章で紹介した `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 (図 3-25 を参照ください)。

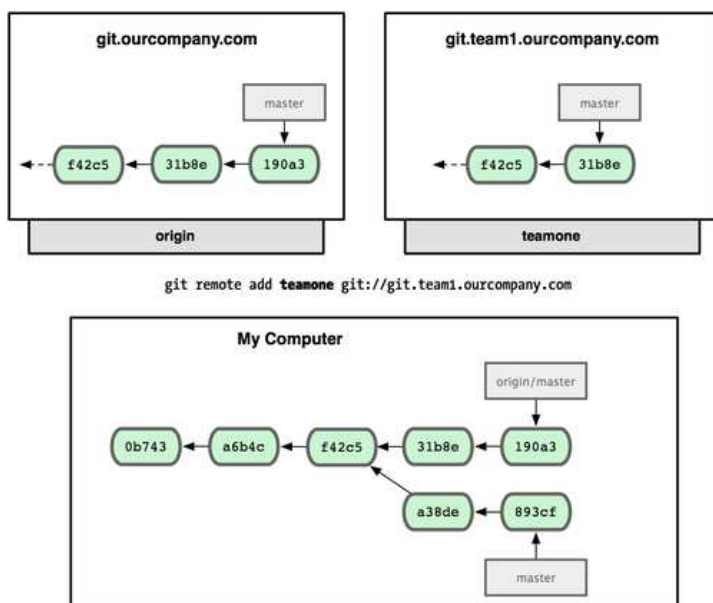


図 3-25. 別のサーバーをリモートとして追加

`git fetch teamone` を実行すれば、まだ手元にないデータをリモートの `teamone` サーバーからすべて取得できるようになりました。このサーバーは `origin` サーバーの現在の状態のサブセットなので、実際のところ Git は何のデータも取得しません。 `teamone/master` というリモートブランチの指す先を、`teamone` の `master` ブランチが指す先にあわせませす (図 3-26 を参照ください)。

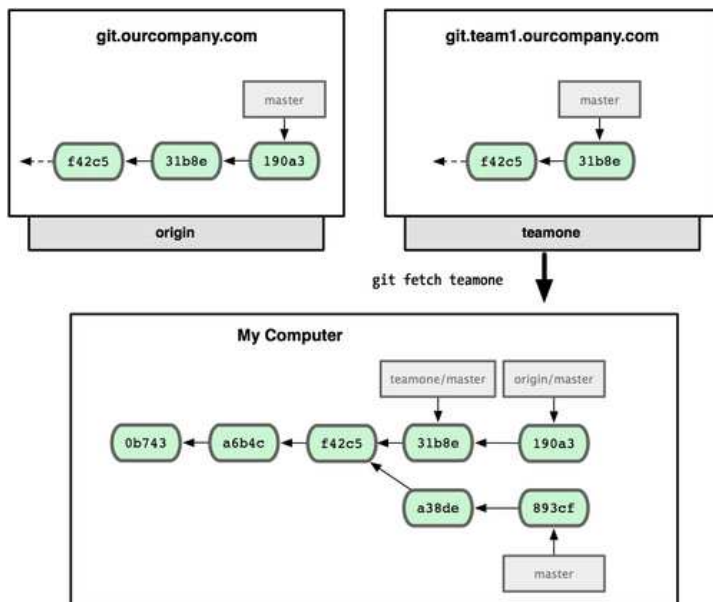


図 3-26. teamone の master ブランチの位置をローカルに取得する

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ができます。

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: 20, done.
Compressing objects: 100% (14/14), done.
Writing objects: 100% (15/15), 1.74 KiB, done.
Total 15 (delta 5), reused 0 (delta 0)
To git@github.com:schacon/simplegit.git
 * [new branch]      serverfix -> serverfix
```

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第 9 章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` という名前前のブランチ名でプッシュすることができます。

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: 20, done.
remote: Compressing objects: 100% (14/14), done.
remote: Total 15 (delta 5), reused 0 (delta 0)
Unpacking objects: 100% (15/15), done.
From git@github.com:schacon/simplegit
 * [new branch]      serverfix -> origin/serverfix
```

注意すべき点は、新しいリモート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ことです。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` ポイントだけであり、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 refs/remotes/origin/serverfix.
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 (tracking branch)* というブランチが自動的に作成されます。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 push を実行すると、Git は自動的にプッシュ先のサーバーとブランチを判断します。また、追跡ブランチ上で git pull を実行すると、リモートの参照先からすべてのデータを取得し、対応するリモートブランチの内容を自動的にマージします。

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。これが、git push や git pull が引数なしでもうまく動作する理由です。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。Git バージョン 1.6.2 以降では、より簡単に --track を使うことができます。

```
$ git checkout --track origin/serverfix
Branch serverfix set up to track remote branch refs/remotes/origin/serverfix.
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 refs/remotes/origin/serverfix.
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。

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コマンドは、少しわかりにくい構文ですが git push [remotename] :[branch] となります。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 :serverfix
To git@github.com:schacon/simplegit.git
- [deleted]          serverfix
```

ドッカーン。これでブランチはサーバーから消えてしまいました。このページの端を折っておいたほうがいいかもしれませんね。実際にこのコマンドが必要になったときには、おそらくこの構文を忘れてしまっているでしょうから。このコマンドを覚えるコツは、少し前に説明した構文 git push [remotename] [localbranch]:[remotebranch] を思い出すことです。[localbranch] の部分をそのまま残して考えると、これは基本的に「こっちの (何もなし) で、向こうの [remotebranch] を更新しろ」と言っていることになります。

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振り返ってみましょう (図 3-27 を参照ください)。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。

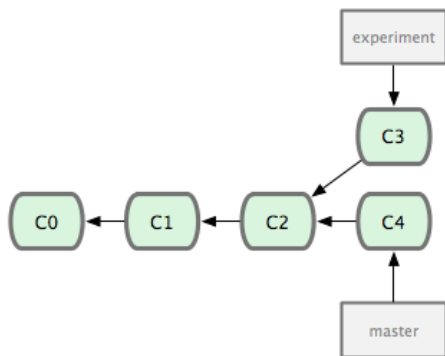


図 3-27. 分岐したコミットの歴史

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。その結果は図 3-28 のようになります。

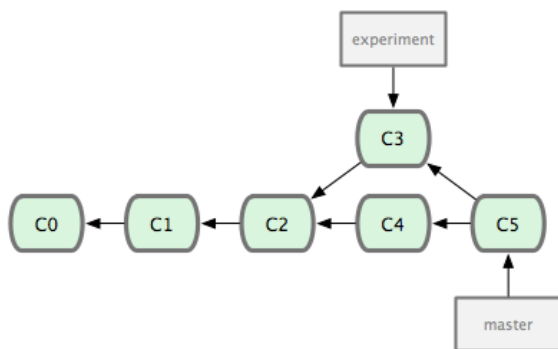


図 3-28. 分岐した作業履歴をひとつに統合する

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

今回の例では、次のように実行します。

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

これは、まずふたつのブランチ (現在いるブランチとリベース先のブランチ) の共通の先祖に移動し、現在のブランチ上の各コミットの diff を取得して一時ファイルに保存し、現在のブランチの指す先をリベース先のブランチと同じコミットに移動させ、そして先ほどの変更を順に適用していきます。図 3-29 にこの手順をまとめました。

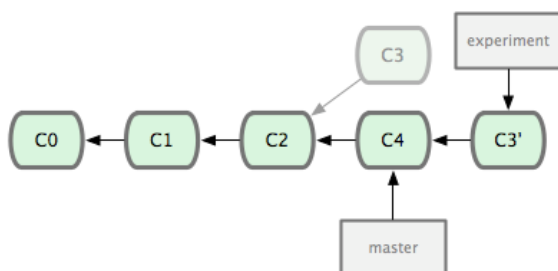


図 3-29. C3 での変更の C4 へのリベース

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 (図 3-30 を参照ください)。

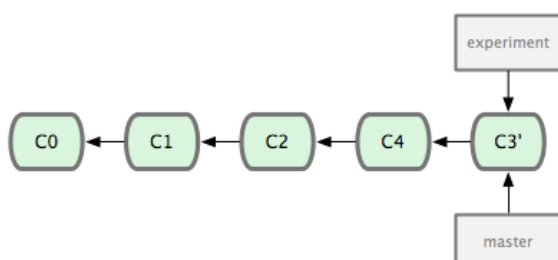


図 3-30. master ブランチの Fast-forward

これで、C3 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。元々平行移動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ならそれを 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。たとえば図 3-31 のような歴史を考えてみましょう。トピックブランチ (server) を作成してサーバー側の機能をプロジェクトに追加し、それをコミットしました。その後、そこからさらにクライアント側の変更用のブランチ (client) を切って数回コミットしました。最後に、server ブランチに戻ってさらに何度かコミットを行いました。

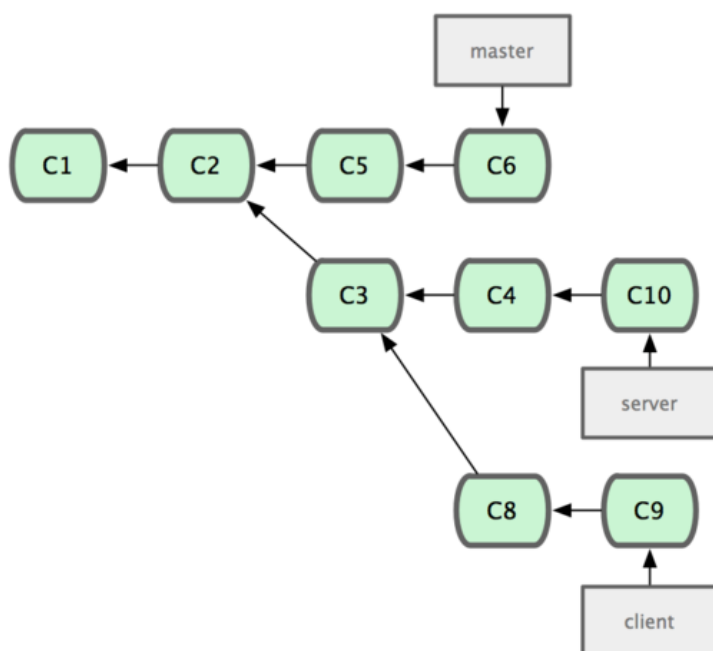


図 3-31. トピックブランチからさらにトピックブランチを作成した歴史

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

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

これは「client ブランチに移動して client ブランチと server ブランチの共通の先祖からのパッチを取得し、master 上でそれを適用しろ」という意味になります。ちょっと複雑ですが、その結果は図 3-32 に示すように非常にクールです。

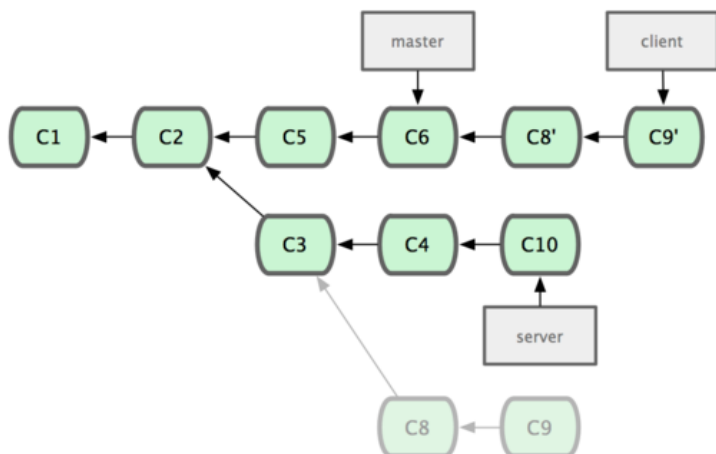


図 3-32. 別のトピックブランチから派生したトピックブランチのリベース

これで、master ブランチを fast-forward することができるようになりました (図 3-33 を参照ください)。

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

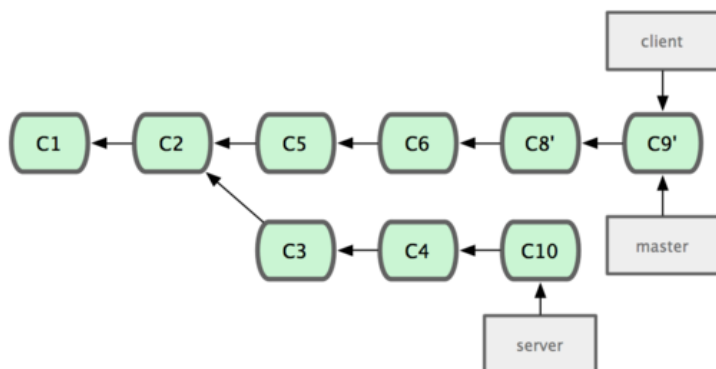


図 3-33. master ブランチを fast-forward し、client ブランチの変更を含める

さて、いよいよ server ブランチのほうも取り込む準備ができました。server ブランチの内容を master ブランチにリベースする際には、事前にチェックアウトする必要はなく `git rebase [basebranch] [topicbranch]` を実行するだけで大丈夫です。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、結果は図 3-34 のようになります。

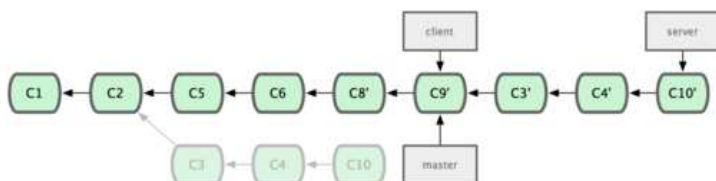


図 3-34. server ブランチを master ブランチ上にリベースする

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

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

ここで client ブランチと server ブランチを削除します。すべての作業が取り込まれたので、これらのブランチはもはや不要だからです。これらの処理を済ませた結果、最終的な歴史は図 3-35 のようになりました。

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

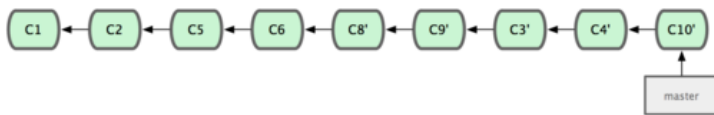


図 3-35. 最終的なコミット履歴

ほんとうは怖いリベース

ああ、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図 3-36 のようになっています。

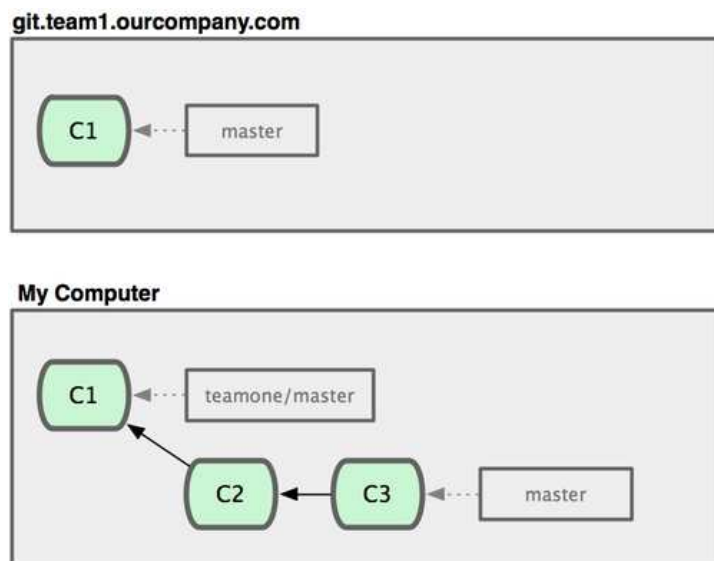


図 3-36. リポジトリをクローンし、なんらかの作業をすませた状態

さて、誰か他の人が、マージを含む作業をしてそれを中央サーバーにプッシュしました。それを取得し、リモートブランチの内容を作業環境にマージすると、図 3-37 のような状態になります。

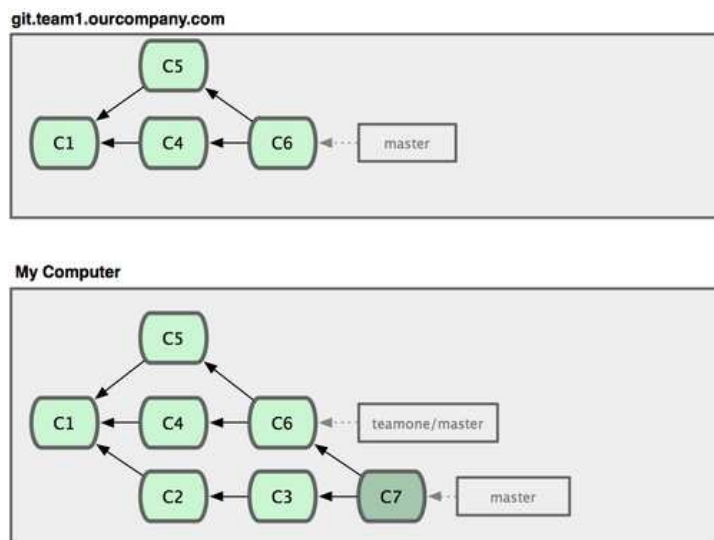
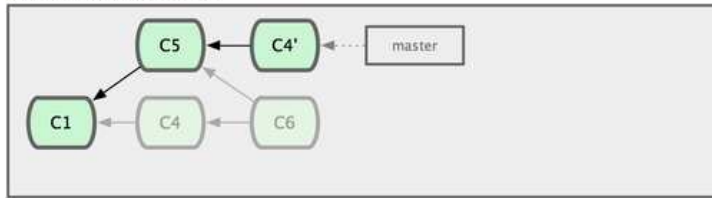


図 3-37. さらにコミットを取得し、作業環境にマージした状態

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。

git.team1.ourcompany.com



My Computer

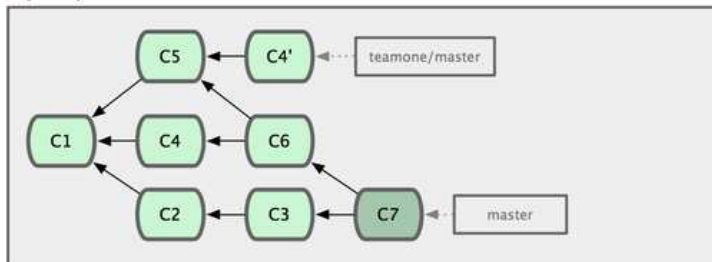
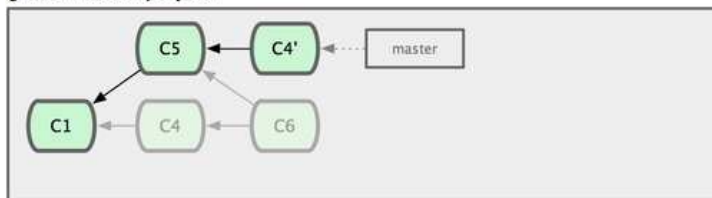


図 3-38. 誰かがリベースしたコミットをプッシュし、あなたの作業環境の元になっているコミットが破棄された

ここであなたは、新しく取得した内容をまたマージしなければなりません。すでにマージ済みのはずであるにもかかわらず。リベースを行うとコミットの SHA-1 ハッシュが変わってしまうので、Git はそれを新しいコミットと判断します。実際のところ C4 の作業は既に取り込み済みなのですが (図 3-39 を参照ください)。

git.team1.ourcompany.com



My Computer

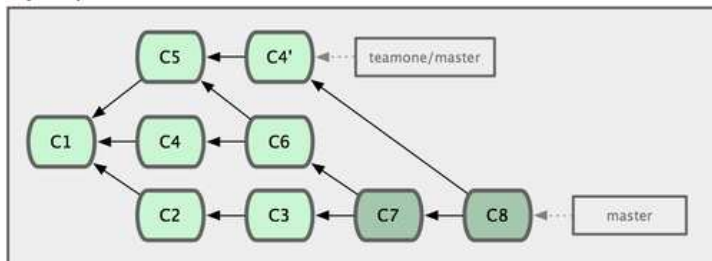


図 3-39. 同じ作業を再びマージして新たなマージコミットを作成する

今後の他の開発者の作業を追いかけていくために、今回のコミットもマージする必要があります。そうすると、あなたのコミット履歴には C4 と C4' の両方のコミットが含まれることになります。これらは SHA-1 ハッシュが異なるだけで、作業内容やコミットメッセージは同じものです。このような状態の歴史の上で `git log` を実行すると、同じ人による同じ日付で同じメッセージのコミットがふたつ登場することになり、混乱します。さらに、この歴史をサーバーにプッシュすると、リベースしたコミットを再び中央サーバーに戻すことになってしまい、混乱する人がさらに増えます。

リベースはあくまでもプッシュする前のコミットをきれいにするための方法であるにとらえ、リベースするのはまだ公開していないコミットのみに限定するようにしている限りはすべてがうまく進みます。もしいったんプッシュした後のコミットをリベースしてしまい、どこか他のところでそのコミットを元に作業を進めている人がいたとすると、やっかいなトラブルに巻き込まれることになるでしょう。

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。

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。本書ではこの手のリポジトリのことを "Git サーバー" と呼ぶことにします。しかし、一般的に Git リポジトリをホストするのに必要なリソースはほんの少しだけです。それ専用のサーバーをわざわざ用意する必要はまずありません。

Git サーバーを立ち上げるのは簡単です。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、データ転送用のネットワークプロトコルとして Local、Secure Shell (SSH)、Git そして HTTP 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。

注意すべき点として、HTTP 以外のすべてのプロトコルは、サーバー上に Git がインストールされている必要があります。

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。単にパスを指定した場合は、Git はハードリンクを行うか、必要に応じて直接ファイルをコピーします。`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、(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) 関係のない参照やオブジェクト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次のセクション「Git サーバーの取得」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よ。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、同じサーバーで稼働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。

SSH プロトコル

Git の転送プロトコルのうちもっとも一般的なのが SSH でしょう。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。また SSH は、ネットワークベースの Git 転送プロトコルの中で、容易に読み書き可能な唯一のもので、その他のネットワークプロトコル (HTTP および Git) は一般的に読み込み専用で用いるものです。不特定多数向けにこれらのプロトコルを開放したとしても、書き込みコマンドを実行するためには SSH が必要となります。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、プロトコルを省略することもできます。プロトコルを明示しなくても、Git はそれが SSH であると見なします。

```
$ 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、ネットワーク越しでのリポジトリへの書き込みアクセスで認証が必要となる場面では、基本的にこのプロトコルを使わなければなりません。次に、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、Git プロトコルや 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-export-daemon-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、もっとも高速な転送プロトコルです。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 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（この章の“Gitosis”のところで詳しく説明します）。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。

HTTP/S プロトコル

最後は HTTP プロトコルです。HTTP あるいは HTTPS のうれしいところは、準備するのが簡単だという点です。基本的に、必要な作業といえば Git リポジトリを HTTP のドキュメントルート以下に置いて `post-update` フックを用意することだけです（Git のフックについては第 7 章で詳しく説明します）。これで、ウェブサーバー上のその場所にアクセスできる人ならだれでもリポジトリをクローンできるようになります。リポジトリへの HTTP での読み込みアクセスを許可するには、こんなふうにします。

```
$ 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 http://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になっているかの詳細は第 9 章を参照ください）。

HTTP 越しの Git のプッシュを行うことも可能ですが、あまり使われていません。また、これには複雑な WebDAV の設定が必要です。めったに使われることがないので、本書では取り扱いません。HTTP でのプッシュに興味があるかたのために、それ用のリポジトリを準備する方法が <http://www.kernel.org/pub/software/scm/git/docs/howto/setup-git-server-over-http.txt> で公開されています。HTTP 越しでの Git のプッシュに関して、よいお知らせがひとつあります。どんな WebDAV サーバーでも使うことが可能で、特に Git ならではの機能は必要ありません。つまり、もしプロバイダが WebDAV によるウェブサイトの更新に対応しているのなら、それを使用することができます。

利点

HTTP を使用する利点は、簡単にセットアップできるということです。便利なコマンドで、Git リポジトリへの読み取りアクセスを全世界に公開できます。ものの数分で用意できることでしょう。また、HTTP はサーバー上のリソースを激しく使用することはありません。すべてのデータは HTTP サーバー上の静的なファイルとして扱われます。普通の Apache サーバーは毎秒数千ファイルぐらいは余裕でさばくので、小規模サーバーであったとしても（Git リポジトリへのへのアクセスで）サーバーが過負荷になることはないでしょう。

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。そこまでやるぐらいなら SSH の公開鍵を使うほうが簡単ではありますが、場合によっては署名入り SSL 証明書やその他の HTTP ベースの認証方式を使った HTTPS での読み込み専用アクセスを使うこともあるでしょう。

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 によるリポジトリの提供の問題点は、クライアント側から見て非効率的だということです。リポジトリのフェッチやクローンには非常に時間がかかります。また、他のネットワークプロトコルにくらべてネットワークのオーバーヘッドや転送量が非常に増加します。必要なデータだけをやりとりするといった賢い機能はない（サーバー側で転送時になんらかの作業をすることができない）ので、HTTP はよく *ダム (dumb)* プロトコルなどと呼ばれています。HTTP とその他のプロトコルの間の効率の違いに関する詳細な情報は、第 9 章を参照ください。

サーバー用の Git の取得

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
Initialized empty Git repository in /opt/projects/my_project.git/
```

このコマンドを実行したときの出力はちょっとわかりにくいかもしれませんが、`clone` は基本的に `git init` をしてから `git fetch` をするのと同じことなので、`git init` の部分の部分の出力も見ることになります。そのメッセージは「空のディレクトリを作成しました」というものです。実際にどんなオブジェクトの転送が行われたのかは何も表示されませんが、きちんと転送は行われています。これで、`my_project.git` ディレクトリに Git リポジトリのデータができました。

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。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 SSH 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、Gitosis の使い方などです。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多少難しくなります。

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る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「○○」「○○.pub」というファイル名の組み合わせを探します。「○○」の部分は、通常は `id_dsa` あるいは `id_rsa` となります。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 MSysGit パッケージに含まれています。

```
$ ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/Users/schacon/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /Users/schacon/.ssh/id_rsa.
Your public key has been saved in /Users/schacon/.ssh/id_rsa.pub.
The key fingerprint is:
43:c5:5b:5f:b1:f1:50:43:ad:20:a6:92:6a:1f:9a:3a schacon@agadorlaptop.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（訳注：メールなんかで送っているの？ とツッコミたいところだ……）。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 AAAAB3NzaC1yc2EAAAABIwAAAQEAklOUpKDhrfHY17SbrmTIpNLTKG9Tjom/BWDSU
GPL+nafzLHDTYW7hdI4yZ5ew18JH4JW9jbhUFrviQzM7xLELEVf4h9lFX5QVkbPppSwg0cda3
Pbv7kOdJ/MTyB1WXFCR+HAo3FXRitBqxiX1nKhXpHAZsMciLq8V6RjsNAQwdsdMFvSLVK/7XA
t3FaoJoAsncM1Q9x5+3V0Ww68/eIFmb1zuUFljQJKprX88XypNDvjYNby6vw/Pb0rwert/En
mZ+AW40ZPnTPi89ZPmVLUayrD2cE86Z/i18b+gw3r3+1nKatmIkjn2so1d01QraTlMqVSsbx
NrRFi9wrf+M7Q== schacon@agadorlaptop.local
```

各種 OS 上での SSH 鍵の作り方については、GitHub の <http://github.com/guides/providing-your-ssh-key> 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
```

次に、開発者たちの SSH 公開鍵をそのユーザーの `authorized_keys` に追加していきましょう。受け取った鍵が一時ファイルとして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 AAAAB3NzaC1yc2EAAAADAQABAAQCB007n/ww+ouN4gSLKssMxXnB0vf9LGt4L
ojG6rs6hPB09j9R/T17/x4lhJA0F3FR1rP6kYBRsWj2aThGw6HXLm9/5zytK6Ztg3RPPK+4k
Yjh6541NYsnEAZuXz0jTTyAUfrtU3Z5E003C4ox0j6H0rfIF1kKI9MAQLMdpGW1GYEIgS9Ez
Sdfd8AcCIicTDWbqLAcl4UpkaX8KyGllwsNuuGztobF8m72ALC/nLF6JLtPofwFBlgc+myiv
07TCUSBdLQlgMV0Fq1I2uPWQOkOWQAhuKE0mfjy2jctxSDBQ220ymjaNsHT4kgtZg2AYYgPq
dAv8JggJICUvax2T9va5 gsg-keypair
```

これを `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 --bare init
```

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。

```
# John のコンピュータで
$ 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
$ 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。

万一の場合に備えて `'git'` ユーザーができることを制限するのも簡単で、Git に関する作業しかできない制限付きシェル `git-shell` が Git に付属しています。これを `'git'` ユーザーのログインシェルにしておけば、`'git'` ユーザーはサーバーへの通常のシェルアクセスができなくなります。これを使用するには、ユーザーのログインシェルとして `bash` や `csh` ではなく `git-shell` を指定します。そのためには `/etc/passwd` ファイルを編集しなければなりません。

```
$ sudo vim /etc/passwd
```

いちばん最後に、このような行があるはずです。

```
git:x:1000:1000::/home/git:/bin/sh
```

ここで `/bin/sh` を `/usr/bin/git-shell` (which `git-shell` を実行してインストール先を探し、それを指定します) に変更します。変更後はこのようになるでしょう。

```
git:x:1000:1000::/home/git:/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、このように拒否されます。

```
$ ssh git@gitserver
fatal: What do you think I am? A shell?
Connection to gitserver closed.
```

一般公開

匿名での読み込み専用アクセス機能を追加するにはどうしたらいいでしょうか？ 身内に閉じたプロジェクトではなくオープンソースのプロジェクトを扱うときに、これを考えることになります。あるいは、自動ビルドや継続的インテグレーション用に大量のサーバーが用意されており、それがしばしば入れ替わるという場合など。単なる読み込み専用の匿名アクセスのためだけに毎回 SSH 鍵を作成しなければならないのは大変です。

おそらく一番シンプルな方法は、静的なウェブサーバーを立ち上げてそのドキュメントルートに Git リポジトリを置き、本章の最初のセクションで説明した `post-update` フックを有効にすることです。先ほどの例を続けてみましょう。リポジトリが `/opt/git` ディレクトリにあり、マシン上で Apache が稼働中であるものとします。念のために言いますが、別に Apache に限らずどんなウェブサーバーでも使えます。しかしここでは、Apache の設定を例にして何が 필요한かを説明していきます。

まずはフックを有効にします。

```
$ cd project.git
$ mv hooks/post-update.sample hooks/post-update
$ chmod a+x hooks/post-update
```

バージョン 1.6 より前の Git を使っている場合は `mv` コマンドは不要です。Git がフックのサンプルに `.sample` という拡張子をつけるようになったのは、つい最近のことです。

この `post-update` は、いったい何をやるのでしょうか？ その中身はこのようになります。

```
$ cat .git/hooks/post-update
#!/bin/sh
exec git-update-server-info
```

SSH 経由でサーバーへのプッシュが行われると、Git はこのコマンドを実行し、HTTP 経由での取得に必要なファイルを更新します。

次に、Apache の設定に `VirtualHost` エントリを追加して Git プロジェクトのディレクトリをドキュメントルートに設定します。ここでは、ワイルドカード DNS を設定して `*.gitserver` へのアクセスがすべてこのマシンに来るようになっているものとします。

```
<VirtualHost *:80>
    ServerName git.gitserver
    DocumentRoot /opt/git
    <Directory /opt/git/>
        Order allow, deny
        allow from all
    </Directory>
</VirtualHost>
```

また、`/opt/git` ディレクトリの所有グループを `www-data` にし、ウェブサーバーがこのリポジトリにアクセスできるようにしなければなりません。CGI スクリプトを実行する Apache インスタンスのユーザー権限で動作することになるからです。

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

Apache を再起動すれば、プロジェクトの URL を指定してリポジトリのクローンができるようになります。

```
$ git clone http://git.gitserver/project.git
```

この方法を使えば、多数のユーザーに HTTP での読み込みアクセス権を与える設定がたった数分でできあがります。多数のユーザーに認証なしでのアクセスを許可するためのもうひとつの方法として、Git デーモンを立ち上げることもできます。しかし、その場合はプロセスをデーモン化させなければなりません。その方法については次のセクションで説明します。

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。GitWeb の使用例は、たとえば `http://git.kernel.org` で確認できます (図 4-1 を参照ください)。

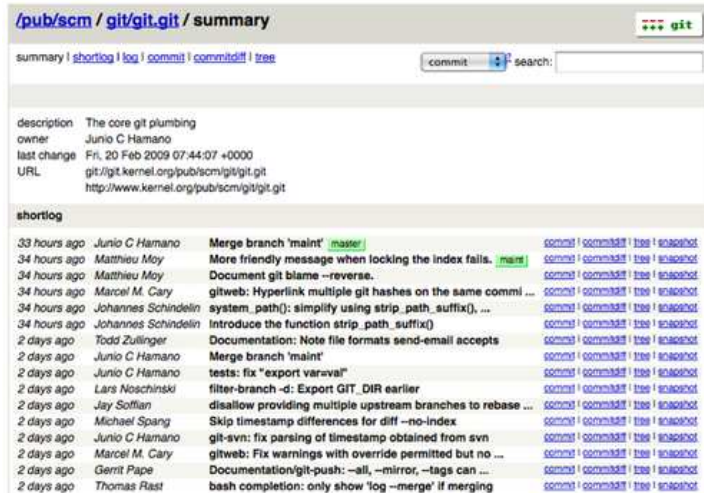


図 4-1. GitWeb のユーザーインターフェイス

自分のプロジェクトでために 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=webrick` 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/gitweb.cgi
$ 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 `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 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。また `http://git.gitserver` で、HTTP 越しのクローンやフェッチもできます。

Gitosis

ユーザーの公開鍵を `authorized_keys` にまとめてアクセス管理する方法は、しばらくの間はうまくいくでしょう。しかし、何百人ものユーザーを管理する段階になると、この方式はとても面倒になります。サーバーのシェルでの操作が毎回発生するわけですし、またアクセス制御が皆無な状態、つまり公開鍵を登録した人はすべてのプロジェクトのすべてのファイルを読み書きできる状態になってしまいます。

ここで、よく使われている Gitosis というソフトウェアについて紹介しましょう。Gitosis は、`authorized_keys` ファイルを管理したりちょっとしたアクセス制御を行ったりするためのスクリプト群です。ユーザーを追加したりアクセス権を定義したりするための UI に、ウェブではなく独自の Git リポジトリを採用しているというのが興味深い点です。プロジェクトに関する情報を準備してそれをプッシュすると、その情報に基づいて Gitosis がサーバーを設定するというクールな仕組みになっています。

Gitosis のインストールは簡単だとはいえませんが、それほど難しくありません。Linux サーバー上で運用するのがいちばん簡単でしょう。今回の例では、ごく平凡な Ubuntu 8.10 サーバーを使います。

Gitosis は Python のツールを使います。まずは Python の `setuptools` パッケージをインストールしなければなりません。Ubuntu なら `python-setuptools` というパッケージがあります。

```
$ apt-get install python-setuptools
```

次に、プロジェクトのメインサイトから Gitosis をクローンしてインストールします。

```

$ git clone git://eagain.net/gitosis.git
$ cd gitosis
$ sudo python setup.py install

```

これで、Gitosis が使う実行ファイル群がインストールされました。Gitosis は、リポジトリが `/home/git` にあることが前提となっています。しかしここではすでに `/opt/git` にリポジトリが存在するので、いろいろ設定しなおすのではなくシンボリックリンクを作ってしまうでしょう。

```
$ ln -s /opt/git /home/git/repositories
```

Gitosis は鍵の管理も行うので、まず現在の鍵ファイルを削除してあとでもう一度鍵を追加し、Gitosis に `authorized_keys` を自動管理させなければなりません。ここではまず `authorized_keys` を別の場所に移動します。

```
$ mv /home/git/.ssh/authorized_keys /home/git/.ssh/ak.bak
```

次は 'git' ユーザーのシェルをもし `git-shell` コマンドに変更していたのなら、元に戻さなければなりません。人にログインさせるのではなく、かわりに Gitosis に管理してもらうのです。 `/etc/passwd` ファイルにある

```
git:x:1000:1000::/home/git:/usr/bin/git-shell
```

の行を、次のように戻しましょう。

```
git:x:1000:1000::/home/git:/bin/sh
```

いよいよ Gitosis の初期設定です。自分の秘密鍵を使って `gitosis-init` コマンドを実行します。サーバー上に自分の公開鍵をおいていない場合は、まず公開鍵をコピーしましょう。

```
$ sudo -H -u git gitosis-init < /tmp/id_dsa.pub
Initialized empty Git repository in /opt/git/gitosis-admin.git/
Reinitialized existing Git repository in /opt/git/gitosis-admin.git/
```

これで、指定した鍵を持つユーザーが Gitosis 用の Git リポジトリを変更できるようになりました。次に、新しいリポジトリの post-update スクリプトに実行ビットを設定します。

```
$ sudo chmod 755 /opt/git/gitosis-admin.git/hooks/post-update
```

これで準備完了です。きちんと設定できていれば、Gitosis の初期設定時に登録した公開鍵を使って SSH でサーバーにログインできるはずです。結果はこのようなになります。

```
$ ssh git@gitserver
PTY allocation request failed on channel 0
fatal: unrecognized command 'gitosis-serve schacon@quaternion'
Connection to gitserver closed.
```

これは「何も Git のコマンドを実行していないので、接続を拒否した」というメッセージです。では、実際に何か Git のコマンドを実行してみましょう。Gitosis 管理リポジトリをクローンします。

```
# on your local computer
$ git clone git@gitserver:gitosis-admin.git
```

gitosis-admin というディレクトリができました。次のような内容になっています。

```
$ cd gitosis-admin
$ find .
./gitosis.conf
./keydir
./keydir/scott.pub
```

gitosis.conf が、ユーザーやリポジトリそしてパーミッションを指定するためのファイルです。keydir ディレクトリには、リポジトリへの何らかのアクセス権を持つ全ユーザーの公開鍵ファイルを格納します。ユーザーごとにひとつのファイルとなります。keydir ディレクトリ内のファイル名（この例では scott.pub）は人によって異なるでしょう。これは、gitosis-init スクリプトでインポートした公開鍵の最後にある説明をもとにして Gitosis がつけた名前です。

gitosis.conf ファイルを見ると、今のところは先ほどクローンした gitosis-admin プロジェクトについての情報しか書かれていません。

```
$ cat gitosis.conf
[gitosis]

[group gitosis-admin]
writable = gitosis-admin
members = scott
```

これは、'scott' ユーザー（Gitosis の初期化時に公開鍵を指定したユーザー）だけが gitosis-admin プロジェクトにアクセスできるという意味です。

では、新しいプロジェクトを追加してみましょう。mobile という新しいセクションを作成し、モバイルチームのメンバーとモバイルチームがアクセスするプロジェクトを書き入れます。今のところ存在するユーザーは 'scott' だけなので、とりあえずは彼をメンバーとして追加します。そして、新しいプロジェクト iphone_project を作ることにしましょう。

```
[group mobile]
writable = iphone_project
members = scott
```

gitosis-admin プロジェクトに手を入れたら、それをコミットしてサーバーにプッシュしないと変更が反映されません。

```
$ git commit -am 'add iphone_project and mobile group'
[master]: created 8962da8: "changed name"
```

```

1 files changed, 4 insertions(+), 0 deletions(-)
$ git push
Counting objects: 5, done.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 272 bytes, done.
Total 3 (delta 1), reused 0 (delta 0)
To git@gitserver:/opt/git/gitosis-admin.git
fb27aec..8962da8 master -> master

```

新しい `iphone_project` プロジェクトにプッシュするには、ローカル側のプロジェクトに、このサーバーをリモートとして追加します。サーバー側でわざわざベアリポジトリを作る必要はありません。先ほどプッシュした地点で、Gitosis が自動的にベアリポジトリの作成を済ませています。

```

$ git remote add origin git@gitserver:iphone_project.git
$ git push origin master
Initialized empty Git repository in /opt/git/iphone_project.git/
Counting objects: 3, done.
Writing objects: 100% (3/3), 230 bytes, done.
Total 3 (delta 0), reused 0 (delta 0)
To git@gitserver:iphone_project.git
* [new branch]      master -> master

```

パスを指定する必要がないことに注目しましょう (実際、パスを指定しても動作しません)。コロンの後にプロジェクト名を指定するだけで、Gitosis がプロジェクトを見つけてくれます。

このプロジェクトに新たなメンバーを迎え入れることになりました、公開鍵を追加しなければなりません。しかし、今までのようにサーバー上の `~/.ssh/authorized_keys` に追記する必要はありません。ユーザー単位の鍵ファイルを `keydir` ディレクトリ内に置くだけです。鍵ファイルにつけた名前が、`gitosis.conf` でその人を指定するときの名前となります。では、John と Josie そして Jessica の公開鍵を追加しましょう。

```

$ cp /tmp/id_rsa.john.pub keydir/john.pub
$ cp /tmp/id_rsa.josie.pub keydir/josie.pub
$ cp /tmp/id_rsa.jessica.pub keydir/jessica.pub

```

そして彼らを 'mobile' チームに追加し、`iphone_project` を読み書きできるようにします。

```

[group mobile]
writable = iphone_project
members = scott john josie jessica

```

この変更をコミットしてプッシュすると、この四人のユーザーがプロジェクトへの読み書きをできるようになります。

Gitosis にはシンプルなアクセス制御機能もあります。John には読み込み専用のアクセス権を設定したいという場合は、このようにします。

```

[group mobile]
writable = iphone_project
members = scott josie jessica

[group mobile_ro]
readonly = iphone_project
members = john

```

John はプロジェクトをクローンして変更内容を受け取れます。しかし、手元での変更をプッシュしようとするとき Gitosis に拒否されます。このようにして好きなだけのグループを作成し、それぞれに個別のユーザーとプロジェクトを含めることができます。また、グループのメンバーとして別のグループを指定し (その場合は先頭に `@` をつけます)、メンバーを自動的に継承することもできます。

```

[group mobile_committers]
members = scott josie jessica

[group mobile]
writable = iphone_project
members = @mobile_committers

```

```
[group mobile_2]
writable = another_iphone_project
members = @mobile_committers john
```

何か問題が発生した場合には、[gitosis] セクションの下に `loglevel=DEBUG` を書いておくと便利です。設定ミスでプッシュ権限を奪われてしまった場合は、サーバー上の `/home/git/.gitosis.conf` を直接編集して元に戻します。Gitosis は、このファイルから情報を読み取っています。gitosis.conf ファイルへの変更がプッシュされてきたときに、その内容をこのファイルに書き出します。このファイルを手動で変更しても、次に gitosis-admin プロジェクトへのプッシュが成功した時点でその内容が書き換えられることになります。

Gitolite

Git は法人の環境でもよく使われるようになってきました。それにつれて、アクセス制御に関する要望もいくつか出てきました。そのような要望にこたえるために作られたのが Gitolite です。

Gitolite は、単なるリポジトリ単位の権限付与 (Gitosis と同じもの) だけではなくリポジトリ内のブランチやタグ単位で権限を付与することができます。つまり、特定の人 (あるいはグループ) にだけ特定の "refs" (ブランチあるいはタグ) に対するプッシュ権限を与えて他の人には許可しないといったことができるのです。

インストール

Gitolite のインストールは非常に簡単で、豊富な付属ドキュメントを読まなくてもインストールできます。必要なものは、何らかの Unix 系サーバのアカウントです。各種の Linux や Solaris 10 でテストされています。root アクセス権は不要です。git や perl、そして openssh 互換の ssh サーバは既にインストールされているものとします。以下の例では、gitserver というホストにあるアカウント gitolite を使います。

不思議なことに、Gitolite のインストールは *ワークステーション上から* スクリプトを実行します。したがって、ワークステーション上に bash シェルが使えなければなりません。msysgit に付属している bash でもかまいません。

まず最初に公開鍵を使ってサーバにアクセスできるようにし、パスワードプロンプトなしにサーバにログインできるようにします。Linux では次の手順が使えますが、他の OS では手動で同じことをしなければならないかもしれません。鍵ペアは、すでに ssh-keygen で生成済みであるものとします。

```
$ ssh-copy-id -i ~/.ssh/id_rsa gitolite@gitserver
```

これを実行すると gitolite アカウントのパスワードを聞かれ、それから公開鍵によるアクセスを設定します。インストールスクリプトを実行するにはこれが **必須** なので、パスワードプロンプトなしにコマンドが実行できることを確認しておきましょう。

```
$ ssh gitolite@gitserver pwd
/home/gitolite
```

次に、プロジェクトのメインサイトから Gitolite をクローンし、"easy install" スクリプトを実行します (三番目の引数は、gitolite-admin リポジトリ上で使われるあなたの名前です)。

```
$ git clone git://github.com/sitaramc/gitolite
$ cd gitolite/src
$ ./gl-easy-install -q gitolite gitserver sitaram
```

これで完了です! Gitolite はサーバにインストールされ、gitolite-admin という新しいリポジトリがあなたのワークステーションのホームディレクトリにできあがりました。gitolite の設定を管理するには、このリポジトリに変更を加えてプッシュします (Gitosis と同じです)。

[ところで、gitolite の *アップグレード* も同じ方法でできます。また、もし興味があれば、スクリプトを引数なしで実行すると利用方法の説明が表示されます。]

最後のコマンドは、かなりの量の出力があります。読んでみるとおもしろいでしょう。また、最初にこれを実行したときには新しい鍵ペアが作られます。このとき、パスフレーズを指定しなければなりません。パスフレーズをなしにするときは単に enter キーを押します。なぜまた別の鍵ペアが必要なのか、そしてそれをどのように使うのかについては、Gitolite の付属ドキュメント "ssh troubleshooting" に説明があります。

インストールのカスタマイズ

デフォルトのインストールは素早く済ませられ、たいいていの人にとってはこれで十分でしょう。しかし、必要に応じてインストール方法をカスタマイズすることもできます。引数 `-q` を省略すると、冗長モードのインストールとなり、各段階で何がインストールされるのかについて詳しい情報が表示されるようになります。冗長モードでインストールすると、サーバ側のパラメータを変更することもできます。たとえば実際のリポジトリの場所などです。パラメータを変更するには、サーバが使用する "rc" ファイルを編集します。この "rc" ファイル

には大量にコメントが書かれているので、必要に応じて簡単に書き換えることができるでしょう。このファイルには、gitolite の高度な機能の有効/無効を切り替えるような設定項目も多く含まれています。

設定ファイルおよびアクセス制御ルール

インストールが終わったら、gitolite-admin リポジトリ (HOME ディレクトリにあります) に移動して中をのぞいてみましょう。

```
$ cd ~/gitolite-admin/
$ ls
conf/  keydir/
$ find conf keydir -type f
conf/gitolite.conf
keydir/sitaram.pub
$ cat conf/gitolite.conf
#gitolite conf
# please see conf/example.conf for details on syntax and features

repo gitolite-admin
  RW+                = sitaram

repo testing
  RW+                = @all
```

"sitaram" (先ほどの gl-easy-install コマンドで、最後の引数に指定したユーザです) が、gitolite-admin リポジトリおよび同名の公開鍵ファイルへの読み書き権限を持っていることに注目しましょう。

Gitolite の設定ファイルの構文は、Gitis のものとは まったく 違います。何度も言いますが conf/example.conf には詳しいコメントが書かれているので、ここでは大事なところに絞って説明します。

ユーザやリポジトリをグループにまとめることもできます。グループ名は単なるマクロのようなものです。グループを定義する際には、それがユーザであるかプロジェクトであるかは無関係です。実際にその「マクロ」を 使う 段階になって初めてそれらを区別することになります。

```
@oss_repos      = linux perl rakudo git gitolite
@secret_repos   = fenestra pear

@admins         = scott      # Adams, not Chacon, sorry :)
@interns        = ashok     # get the spelling right, Scott!
@engineers      = sitaram dilbert wally alice
@staff          = @admins @engineers @interns
```

パーミッションは、"ref" レベルで設定することができます。次の例では、インターン (@interns) は "int" ブランチにしかプッシュできないように設定しています。エンジニア (@engineers) は名前が "eng-" で始まるすべてのブランチにプッシュでき、また "rc" のあとに一桁の数字が続く名前のタグにもプッシュできます。また、管理者 (@admins) はすべての ref に対してあらゆることができます。

```
repo @oss_repos
  RW int$          = @interns
  RW eng-          = @engineers
  RW refs/tags/rc[0-9] = @engineers
  RW+              = @admins
```

RW や RW+ の後に書かれている式は正規表現 (regex) で、これにマッチする refname (ref) が対象となります。なので、これに "refex" と名付けました! もちろん、refex はこの例で示したよりずっと強力なものです。perl の正規表現になじめない人は、あまりやりすぎないようにしましょう。

また、すでにお気づきかもしれませんが、refs/ で始まらない refex を指定すると、Gitolite はその先頭に refs/heads/ がついているものとみなします。これは構文上の利便性を意識したものです。

設定ファイルの構文の中でも重要なのは、ひとつのリポジトリに対するルールをすべてひとまとめにしなくてもよいということです。共通の設定をひとまとめにして上の例のように oss_repos に対してルールを設定し、その後で個々の場合について個別のルールを追加していくこともできます。

```
repo gitolite
  RW+                = sitaram
```

このルールは、gitolite リポジトリのルール群に追加されます。

で、実際のアクセス制御ルールはどのように書けばいいの? と思われたことでしょう。簡単に説明します。

gitolite のアクセス制御には二段階のレベルがあります。まず最初はリポジトリレベルのアクセス制御です。あるリポジトリへの読み込み (書き込み) アクセス権を持っているということは、そのリポジトリの *すべての* ref に対する読み込み (書き込み) 権限を持っていることを意味します。Gitolite で設定できたのはこのレベルのアクセス制御だけです。

もうひとつのレベルは "書き込み" アクセス権だけを制御するものですが、リポジトリ内のブランチやタグ単位で設定できます。ユーザ名、試みられるアクセス (W あるいは +)、そして更新される refname が既知となります。アクセスルールのチェックは、設定ファイルに書かれている順に行われ、この組み合わせにマッチ (単なる文字列マッチではなく正規表現によるマッチであることに注意しましょう) するものを探していきます。マッチするものが見つかったら、プッシュが成功します。マッチしなかった場合は、アクセスが拒否されます。

"拒否" ルールによる高度なアクセス制御

これまでに見てきた権限は R、RW あるいは RW+ だけでした。しかし、gitolite にはそれ以外の権限もあります。それが - で、"禁止" をあらわすものです。これを使えばより強力なアクセス制御ができるようになりますが、少し設定は複雑になります。マッチしなければアクセスを拒否するというだけでなく、ルールを書く順番もからんでくることになるからです。

上の例で、エンジニアは master と integ *以外* のすべてのブランチを巻き戻せるように設定しようとなると、次のようになります。

```
RW master integ      = @engineers
-   master integ      = @engineers
RW+                               = @engineers
```

この場合も上から順にルールを適用し、最初にマッチしたアクセス権をあてはめます。何もマッチしなければアクセスは拒否されます。master や integ に対する巻き戻し以外のプッシュは、最初のルールにマッチするので許可されます。これらのブランチに対する巻き戻しのプッシュは最初のルールにマッチしません。そこで二番目のルールに移動し、この時点で拒否されます。master と integ 以外への (巻き戻しを含む) 任意のプッシュは最初の二つのルールのいずれにもマッチしないので、三番目のルールが適用されます。

ファイル単位でのプッシュの制限

変更をプッシュすることのできるブランチを制限するだけでなく、変更できるファイルを制限することも可能です。たとえば、Makefile (あるいはその他のプログラム) などは誰もが変更できるというものではないでしょう。このファイルはさまざまなものに依存しており、変更によっては壊れてしまうことがあるかもしれないからです。そんな場合は次のように設定します。

```
repo foo
RW                               = @junior_devs @senior_devs

RW NAME/                        = @senior_devs
-   NAME/Makefile              = @junior_devs
RW NAME/                        = @junior_devs
```

この強力な機能については conf/example.conf に説明があります。

個人ブランチ

Gitolite には "個人ブランチ" ("個人的なブランチ用の名前空間" と言ったほうがいいでしょうか) という機能があります。これは、法人の環境では非常に便利なものです。

git の世界では、いわゆる「ブルリクエスト」によるコードのやりとりが頻繁に発生します。しかし法人の環境では、権限のない人によるアクセスは厳禁です。開発者のワークステーションにはそんな権限はありません。そこで、まず一度中央サーバにプッシュして、そこからプルしてもらうよう誰かにお願いすることになります。

これを中央管理型の VCS でやろうとすると、同じ名前のブランチが乱造されることになってしまいます。また、これらのアクセス権限を設定するのは管理者にとって面倒な作業です。

Gitolite では、各開発者に対して "personal" あるいは "scratch" といった名前空間プレフィックス (たとえば refs/personal /<devname>/*) を定義します。そして、その開発者には完全なアクセス権を、それ以外のメンバーには読み込みのみのアクセス権を設定します。この機能を使うには、冗長モードでインストールして "rc" ファイルの変数 \$PERSONAL を refs/personal に設定するだけです。プロジェクトのチーム構成が頻繁に変わる環境であっても、管理者が注意していればきちんと追いかけるでしょう。

"ワイルドカード" リポジトリ

Gitolite では、ワイルドカード (実際のところは perl の正規表現です) を使ってリポジトリを指定することができます。たとえば assignments/s[0-9][0-9]/a[0-9][0-9] のようにします。これは非常に強力な機能で、使うには rc ファイルに \$GL_WILDCARD = 1; と設定しなければなりません。この機能を使うと、新たな権限モード ("C") が用意されます。これは、ワイルドカードにマッチするリポジトリの作成を許可するモードです。新たに作成したリポジトリの所有者は自動的にそのユーザに設定され、他のユーザに R あるいは RW の

権限を付与できるようになります。この機能の説明は doc/4-wildcard-repositories.mkd にあります。

その他の機能

最後にその他の機能をまとめて紹介しましょう。これらについての詳しい説明は、ドキュメントの "faqs, tips, etc" にあります。

ログ記録: Gitolite は、成功したアクセスをすべてログに記録します。巻き戻し権限 (RW+) を与えているときに、誰かが "master" を吹っ飛ばしてしまったとしましょう。そんなときにはログファイルが救世主となります。ログを見れば、問題を起こした SHA をすぐに発見できるでしょう。

通常の PATH 以外にインストールした git: gitolite の非常に便利な機能のひとつは、通常の \$PATH 以外にインストールされた git もサポートしているという点です (これは思っているほど珍しいことではありません。法人の環境やホスティング環境ではシステム全体への影響を及ぼすインストールが禁じられているところもあり、そんな場合は自分のディレクトリの配下にインストールすることになるからです)。通常なら、標準以外の場所にインストールされた git バイナリを何らかの方法でクライアント側の git に認識させなければなりません。gitolite があれば単に冗長モードのインストールで "rc" ファイルに \$GIT_PATH を設定するだけです。クライアント側の変更は不要です :-)

アクセス権の報告: もうひとつの便利な機能は、サーバに ssh で接続したときに起こります。古いバージョンの gitolite では、環境変数 SSH_ORIGINAL_COMMAND (興味のあるかたは ssh のドキュメントを参照ください) が空であることが不満でした。現在の gitolite では、このようになります。

```
hello sitaram, the gitolite version here is v0.90-9-g91e1e9f
you have the following permissions:
R      anu-wsd
R      entrans
R W    git-notes
R W    gitolite
R W    gitolite-admin
R      indic_web_input
R      shreelipi_converter
```

委譲: 大規模な環境では、特定のリポジトリのグループに対する責任を委譲して個別に管理させることもできます。こうすれば管理者の負荷が軽減され、管理者がボトルネックとなることも少なくなります。この機能については、doc/ ディレクトリの中に個別のファイルで説明されています。

Gitweb のサポート: Gitolite は gitweb を何通りかの方法でサポートしています。まず、どのリポジトリを gitweb 上で見せるかを設定することができます。また、gitweb 用の "owner" と "description" を gitolite の設定ファイルに書くことができます。Gitweb には HTTP 認証に基づいたアクセス制御の機能もありますが、gitolite が生成する "コンパイル済み" の設定ファイルを使えば gitweb と gitolite で共通の (読み込み) アクセス制御ルールを適用することができます。

Git デーモン

認証の不要な読み取り専用アクセスを一般に公開する場合は、HTTP を捨てて Git プロトコルを使うことを考えることになるでしょう。主な理由は速度です。Git プロトコルのほうが HTTP に比べてずっと効率的で高速です。Git プロトコルを使えば、ユーザーの時間を節約することになります。

Git プロトコルは、認証なしで読み取り専用アクセスを行うためのものです。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ります。

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、フルパスをしてい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/event.d/local-git-daemon
```

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 ¥
    --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 Gitosis 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 Gitosis に指定します。各リポジトリ用のセクションを追加すれば、Git デーモンからの読み込みアクセスを許可するように指定することができます。Git プロトコルでのアクセスを iphone プロジェクトに許可したい場合は、gitosis.conf の最後に次のように追加します。

```
[repo iphone_project]
daemon = yes
```

この変更をコミットしてプッシュすると、デーモンがこのプロジェクトへのアクセスを受け付けるようになります。

Gitosis を使わずに Git デーモンを設定したい場合は、Git デーモンで公開したいプロジェクトに対してこのコマンドを実行しなければなりません。

```
$ 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。

Gitosis を使うと、どのプロジェクトを GitWeb で見せるのかを指定することもできます。まずは次のような行を /etc/gitweb.conf に追加しましょう。

```
$projects_list = "/home/git/gitosis/projects.list";
$projectroot = "/home/git/repositories";
$export_ok = "git-daemon-export-ok";
@git_base_url_list = ('git://gitserver');
```

GitWeb でどのプロジェクトを見せるのかを設定するには、Gitosis の設定ファイルで gitweb を指定します。たとえば、iphone プロジェクトを GitWeb で見せたい場合は、repo の設定は次のようになります。

```
[repo iphone_project]
daemon = yes
gitweb = yes
```

これをコミットしてプッシュすると、GitWeb で iphone プロジェクトが自動的に表示されるようになります。

Git のホスティング

Git サーバーを立ち上げる作業が面倒なら、外部の専用ホスティングサイトに Git プロジェクトを置くという選択肢があります。この方法には多くの利点があります。ホスティングサイトでプロジェクトを立ち上げるのは簡単ですし、サーバーのメンテナンスや日々の監視も不要です。自前のサーバーを持っているとしても、オープンソースのコードなどはホスティングサイトで公開したいこともあるかもしれません。そのほうがオープンソースコミュニティのメンバーに見つけてもらいやすく、そして支援を受けやすくなります。

今ではホスティングの選択肢が数多くあり、それぞれ利点もあれば欠点もあります。最新の情報を知るには、Git wiki で GitHosting のページを調べましょう。

<http://git.or.cz/gitwiki/GitHosting>

これらすべてについて網羅することは不可能ですし、たまたま私自身がこの中のひとつで働いていることもあるので、ここでは GitHub を使ってアカウントの作成からプロジェクトの立ち上げまでの手順を説明します。どのような流れになるのかを見ていきましょう。

GitHub は最大のオープンソース Git ホスティングサイトで、公開リポジトリだけでなく非公開のリポジトリもホスティングできる数少ないサイトのひとつです。つまり、オープンソースのコードと非公開のコードを同じ場所で管理できるのです。実際、本書に関する非公開の共同作業についても GitHub を使っています。

GitHub

GitHub がその他多くのコードホスティングサイトと異なるのは、プロジェクトの位置づけです。プロジェクトを主体に考えるのではなく、GitHub ではユーザー主体の構成になっています。私が GitHub に `grit` プロジェクトを公開したとして、それは `github.com/grit` ではなく `github.com/schacon/grit` となります。どのプロジェクトにも「正式な」バージョンというものはありません。たとえ最初の作者がプロジェクトを放棄したとしても、それをユーザーからユーザーへと自由に移動することができます。

GitHub は営利企業なので、非公開のリポジトリについては料金をとって管理しています。しかし、フリーのアカウントを取得すればオープンソースのプロジェクトを好きなだけ公開することができます。その方法についてこれから説明します。

ユーザーアカウントの作成

まずはフリー版のユーザーアカウントを作成しましょう。Pricing and Signup のページ <http://github.com/plans> で、フリーアカウントの "Sign Up" ボタンを押すと (図 4-2 を参照ください)、新規登録ページに移動します。



図 4-2. GitHub のプラン説明ページ

ユーザー名を選び、メールアドレスを入力します。アカウントとパスワードがこのメールアドレスに関連づけられます (図 4-3 を参照ください)。

図 4-3. GitHub のユーザー登録フォーム

それが終われば、次に SSH の公開鍵を追加しましょう。新しい鍵を作成する方法については、さきほど「ちょっとしたセットアップ」のところで説明しました。公開鍵の内容をコピーし、SSH Public Key のテキストボックスに貼り付けます。"explain ssh keys" のリンクをクリックすると、主要 OS 上での公開鍵の作成手順を詳しく説明してくれます。"I agree, sign me up" ボタンをクリックすると、あなたのダッシュボードに移動します (図 4-4 を参照ください)。



図 4-4. GitHub のユーザーダッシュボード

では次に、新しいリポジトリの作成に進みましょう。

新しいリポジトリの作成

ダッシュボードで、Your Repositories の横にあるリンク "create a new one" をクリックしましょう。新規リポジトリの作成フォームに進みます (図 4-5 を参照ください)。

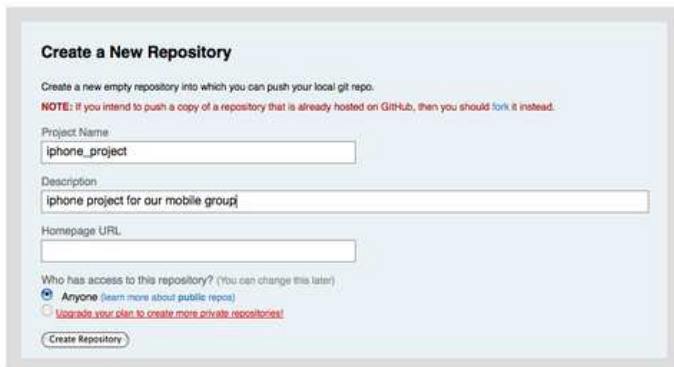


図 4-5. GitHub での新しいリポジトリの作成

ここで必要なのはプロジェクト名を決めることです。ただ、それ以外に説明文を追加することもできます。ここで "Create Repository" ボタンを押せば、GitHub 上での新しいリポジトリのできあがりです (図 4-6 を参照ください)。



図 4-6. GitHub でのプロジェクトのヘッダ情報

まだ何もコードが追加されていないので、ここでは「新しいプロジェクトを作る方法」「既存の Git プロジェクトをプッシュする方法」「Subversion の公開リポジトリからインポートする方法」が説明されています (図 4-7 を参照ください)。



図 4-7. 新しいリポジトリに関する説明

この説明は、本書でこれまでに説明してきたものとほぼ同じです。まだ Git プロジェクトでないプロジェクトを初期化するには、次のようになります。

```
$ git init
```

```
$ git add .
$ git commit -m 'initial commit'
```

ローカルにある Git リポジトリを使用する場合は、GitHub をリモートに登録して master ブランチをプッシュします。

```
$ git remote add origin git@github.com:testinguser/iphone_project.git
$ git push origin master
```

これで GitHub 上でリポジトリが公開され、だれもがプロジェクトにアクセスできるような URL ができあがりしました。この例の場合は `http://github.com/testinguser/iphone_project` です。各プロジェクトのページのヘッダには、ふたつの Git URL が表示されています (図 4-8 を参照ください)。



図 4-8. 公開 URL とプライベート URL が表示されたヘッダ

Public Clone URL は、読み込み専用の公開 URL で、これを使えば誰でもプロジェクトをクローンできます。この URL は、あなたのウェブサイトをはじめとしたお好みの場所で紹介することができます。

Your Clone URL は、読み書き可能な SSH の URL で、先ほどアップロードした公開鍵に対応する SSH 秘密鍵を使った場合にのみアクセスできます。他のユーザーがこのプロジェクトのページを見てもこの URL は表示されず、公開 URL のみが見えるようになっています。

Subversion からのインポート

GitHub では、Subversion で公開しているプロジェクトを Git にインポートすることもできます。先ほどの説明ページの最後のリンクをクリックすると、Subversion からのインポート用ページに進みます。このページにはインポート処理についての情報が表示されており、公開 Subversion リポジトリの URL を入力するテキストボックスが用意されています。

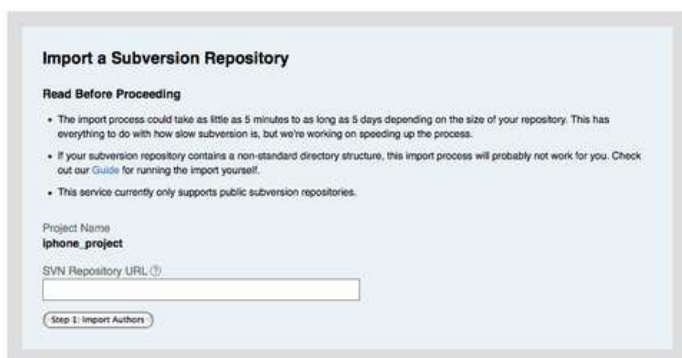


図 4-9. Subversion からのインポート

もしそのプロジェクトが非常に大規模なものであったり標準とは異なるものであったり、あるいは公開されていないものであったりした場合は、この手順ではうまくいかないでしょう。第 7 章で、手動でのプロジェクトのインポート手順について詳しく説明します。

共同作業者の追加

では、チームの他のメンバーを追加しましょう。John、Josie そして Jessica は全員すでに GitHub のアカウントを持っており、彼らもこのリポジトリにプッシュできるようにしたければ、プロジェクトの共同作業者として登録します。そうすれば、彼らの公開鍵をつかったプッシュも可能となります。

プロジェクトのヘッダにある "edit" ボタンをクリックするかプロジェクトの上の Admin タブを選択すると、GitHub プロジェクトの管理者用ページに移動します (図 4-10 を参照ください)。

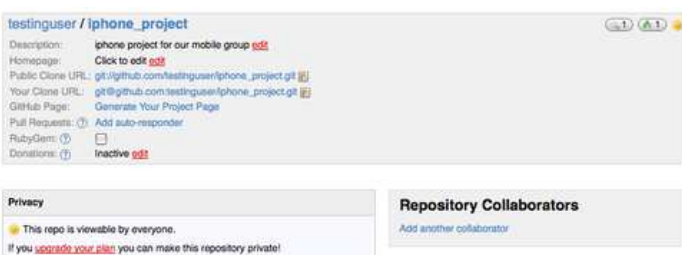


図 4-10. GitHub の管理者用ページ

別のユーザーにプロジェクトへの書き込み権限を付与するには、“Add another collaborator”リンクをクリックします。新しいテキストボックスがあらわれるので、そこにユーザー名を記入します。何か入力すると、マッチするユーザー名の候補がポップアップ表示されます。ユーザーが見つければ、Add ボタンをクリックすればそのユーザーを共同作業者に追加できます (図 4-11 を参照ください)。



図 4-11. プロジェクトへの共同作業者の追加

対象者を全員追加し終えたら、Repository Collaborators のところにその一覧が見えるはずで (図 4-12 を参照ください)。



図 4-12. プロジェクトの共同作業者一覧

誰かのアクセス権を剥奪したい場合は、“revoke” リンクをクリックすればそのユーザーはプッシュできなくなります。また、今後新たにプロジェクトを作ったときに、この共同作業者一覧をコピーして使うこともできます。

あなたのプロジェクト

プロジェクトをプッシュするか、あるいは Subversion からのインポートを済ませると、プロジェクトのメインページは図 Figure 4-13 のようになります。

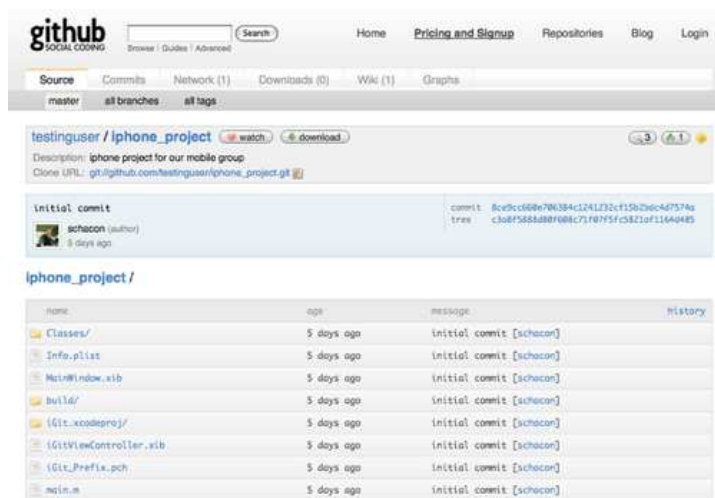


図 4-13. GitHub プロジェクトのメインページ

他の人がこのプロジェクトにアクセスしたときに見えるのがこのページとなります。このページには、さまざまな情報を見るためのタブが用意されています。Commits タブに表示されるのはコミットの一覧で、git log コマンドの出力と同様にコミット時刻が新しい順に表示されます。Network タブには、このプロジェクトをフォークして何か貢献してくれた人の一覧が表示されます。Downloads タブには、プロジェクト内でタグが打たれている任意の点について tar や zip でまとめたものをアップロードすることができます。Wiki タブには、プロジェクトに関するドキュメントやその他の情報を書き込むための wiki が用意されています。Graphs タブは、プロジェクトに対する貢献やその他の統計情報を視覚化して表示します。そして、Source タブにはプロジェクトのメインディレクトリの一覧が表示され、もし README ファイルがあればその内容が下に表示されます。このタブでは、最新のコミットについての情報も表示されます。

プロジェクトのフォーク

プッシュアクセス権のない別のプロジェクトに協力したくなったときは、GitHub ではプロジェクトをフォークすることを推奨しています。興味を持ったとあるプロジェクトのページに行って、それをちょっとばかりハックしたくなったときは、プロジェクトのヘッダにある “fork” ボタンをクリックしましょう。GitHub が自分のところにそのプロジェクトをコピーしてくれるので、そこへのプッシュができるようになります。

この方式なら、プッシュアクセス権を与えるために共同作業者としてユーザーを追加することを気にせずに済みます。プロジェクトをフォークした各ユーザーが自分のところにプッシュし、主メンテナーは必要に応じてこれらの作業をマージすればいいのです。

プロジェクトをフォークするには、そのプロジェクトのページ (この場合はmojombo/chronic) に移動してヘッダの “fork” ボタンをク

クリックします (図 4-14 を参照ください)。



図 4-14. 任意のプロジェクトの書き込み可能なコピーを取得する "fork" ボタン

数秒後に新しいプロジェクトのページに移動します。そこには、このプロジェクトがどのプロジェクトのフォークであるかが表示されています (図 4-15 を参照ください)。



図 4-15. フォークしたプロジェクト

GitHub のまとめ

これで GitHub についての説明を終えますが、特筆すべき点はこれらの作業を本当に手早く済ませられることです。アカウントを作ってプロジェクトを追加してそこにプッシュする、ここまでがほんの数分でできてしまいます。オープンソースのプロジェクトを公開したのなら、数多くの開発者のコミュニティがあなたのプロジェクトにアクセスできるようになりました。きっと中にはあなたに協力してくれる人もあられることでしょう。少なくとも、Git を動かして試してみる土台としては使えるはずです。

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 (図 5-1 を参照ください)。

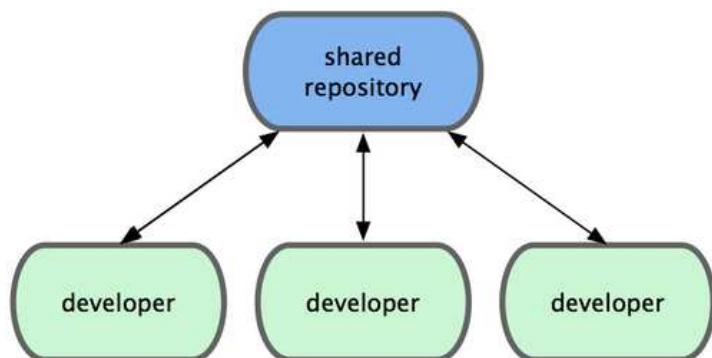


図 5-1. 中央集権型のワークフロー

二人の開発者がハブからのクローンを作成して個々に変更をした場合、最初の実験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、まず最初の実験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、最初の実験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) と同様に生かれます。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。誰かがクローンして手元で変更を加えた内容をプッシュしようとしたときに、もし既に他の誰かの変更がプッシュされていれば、サーバー側でそのプッシュは拒否されます。そして、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しなさいと教えてくれます。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。

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まれます。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 (図 5-2 を参照ください)。

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

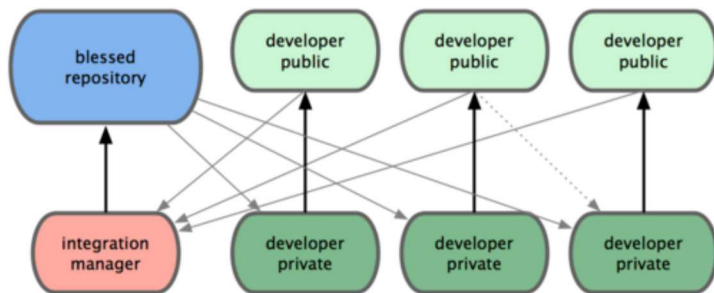


図 5-2. 統合マネージャー型のワークフロー

これは GitHub のようなサイト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け、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。

独裁者と若頭型のワークフロー

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。彼らは若頭 (lieutenant) と呼ばれます。そしてすべての若頭をまとめる統合マネージャーが「慈悲深い独裁者 (benevolent 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 (図 5-3 を参照ください)。

1. 一般の開発者はトピックブランチ上で作業を進め、master の先頭にリベースする。独裁者の master ブランチがマスターとなる
2. 若頭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. 独裁者が各若頭の 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

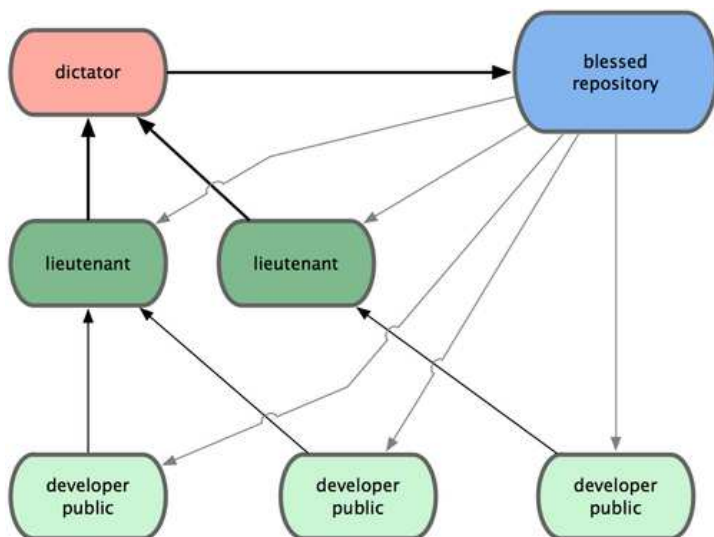


図 5-3. 慈悲深い独裁者型のワークフロー

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

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

さまざまなワークフローの概要について説明しました。また、すでにみなさんは Git の基本的な使い方を身につけています。このセクションでは、何らかのプロジェクトに貢献する際のよくあるパターンについて学びましょう。

これ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。大企業や大規模なプロジェクトでは、開発者の数が数千になることも

あります。数十から下手したら百を超えるようなパッチ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、正しいパッチ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？ 若頭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 をご覧ください。

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。例を示しましょう。端末上では赤で表示される箇所を X で置き換えています。

```
$ git diff --check
lib/simplegit.rb:5: trailing whitespace.
+ @git_dir = File.expand_path(git_dir)XX
lib/simplegit.rb:7: trailing whitespace.
+ XXXXXXXXXXXX
lib/simplegit.rb:26: trailing whitespace.
+ def command(git_cmd)XXXX
```

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（詳しくは第 6 章で説明します）。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。第 6 章では、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。第 6 章で説明する方法を使えば、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 "私は〇〇のテストを追加しました (I added tests for)" とか "〇〇のテストを追加します (Adding tests for,)" ではなく "〇〇のテストを追加 (Add tests for,)" 形式にすることです。Tim Pope が tpope.net で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。本書での私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せんが、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を開発を始めるときにどうなるかを見ていきましょう。最初の実験者 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@github: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@github: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@github:simplegit.git
1edee6b..fbff5bc master -> master
```

John も同様にプッシュしようとした。

```
# John のマシン
$ git push origin master
To john@github:simplegit.git
! [rejected] master -> master (non-fast forward)
error: failed to push some refs to 'john@github: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。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@github:simplegit
+ 049d078...fbff5bc master    -> origin/master
```

この時点で、John のローカルリポジトリは図 5-4 のようになっています。

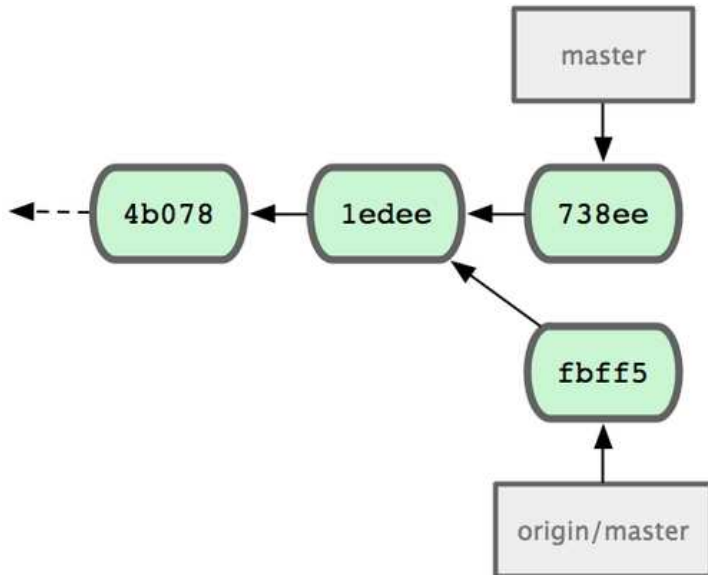


図 5-4. John のリポジトリ

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 のコミット履歴は図 5-5 のようになります。

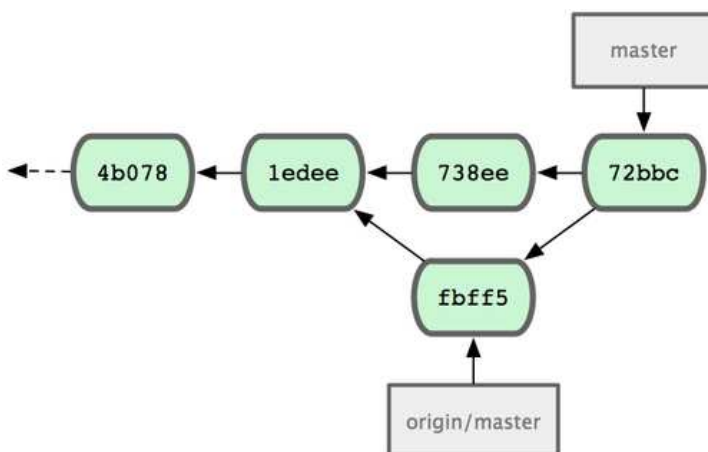


図 5-5. origin/master をマージした後の John のリポジトリ

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@github:simplegit.git
fbff5bc..72bbc59 master -> master
```


最終的に、John のコミット履歴は図 5-6 のようになりました。

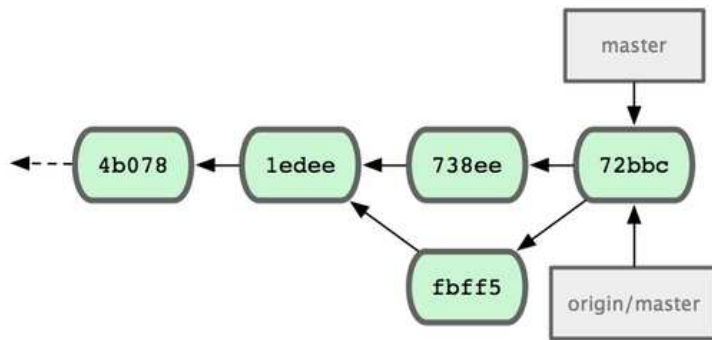


図 5-6. origin サーバーにプッシュした後の John の履歴

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図 5-7 のような状態です。

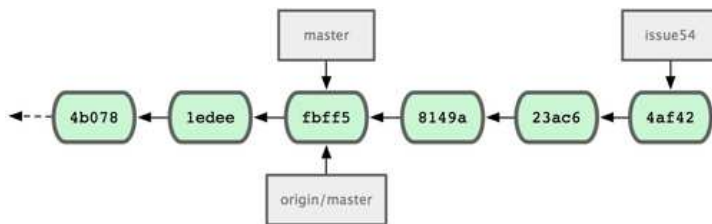


図 5-7. Jessica のコミット履歴

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@github: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 の履歴は図 5-8 のようになります。

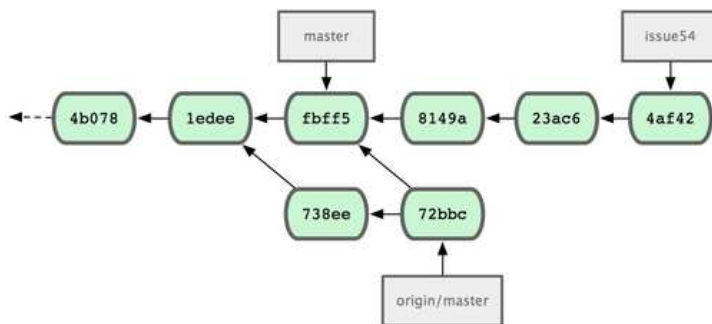


図 5-8. John の変更を取り込んだ後の Jessica の履歴

Jessica のトピックブランチ上での作業が完了しました。プッシュする前にどんな作業をマージしなければならないのかを知るため、彼女は `git log` コマンドを実行しました。

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

removed invalid default value

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なりました。

```
$ 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 の履歴は図 5-9 のようになります。

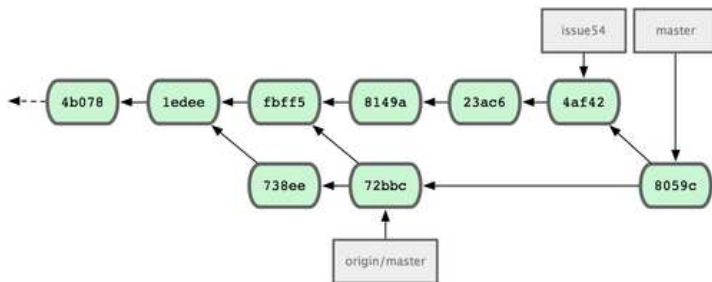


図 5-9. John の変更をマージした後の Jessica の履歴

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@github: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。図 5-10 をごらんください。

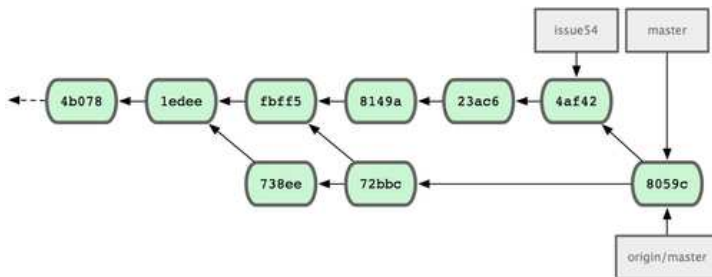


図 5-10. すべての変更をサーバーに書き戻した後の Jessica の履歴

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。全体的な流れは図 5-11 のようになります。



図 5-11. 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 origin featureA
...
To jessica@github: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 のリポジトリは図 5-12 のようになっています。

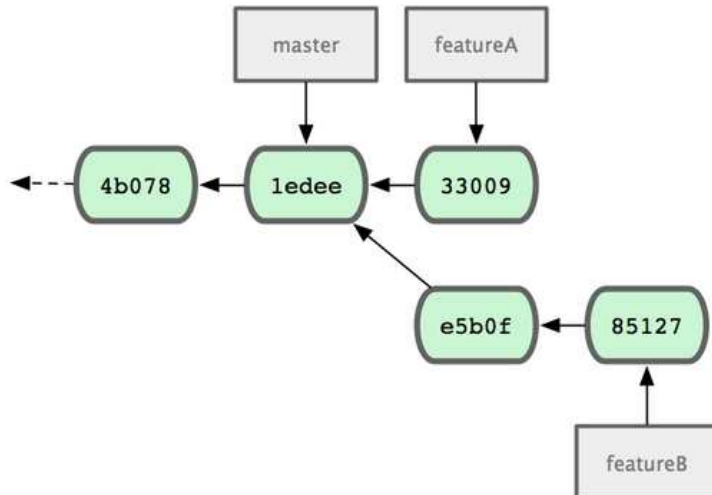


図 5-12. Jessica のコミット履歴

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@github: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 origin featureB:featureBee
...
To jessica@github:simplegit.git
 fba9af8..cd685d1 featureB -> featureBee
```

これは *refspec* と呼ばれます。第 9 章で、Git の refspec の詳細とそれで何ができるのかを説明します。

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@github: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 origin/featureA ^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 ed774b3] small tweak
 1 files changed, 1 insertions(+), 1 deletions(-)
$ git push origin featureA
...
To jessica@github:simplegit.git
3300904..ed774b3 featureA -> featureA
```

Jessica のコミット履歴は、この時点で図 Figure 5-13 のようになります。

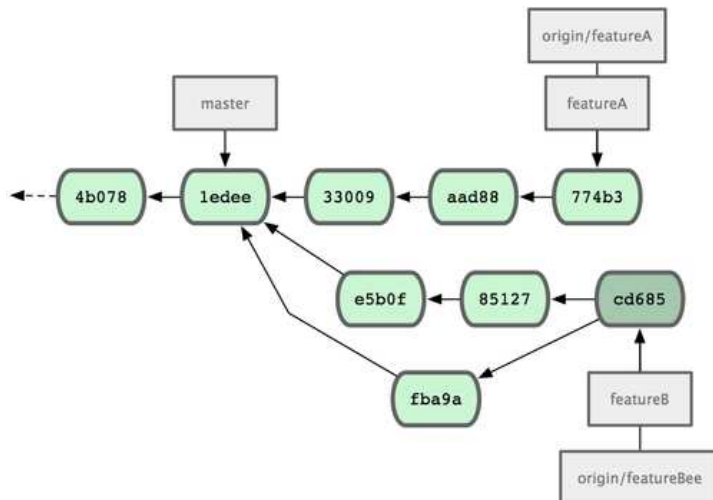


図 5-13. Jessica がブランチにコミットした後のコミット履歴

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。これらのブランチが本流に統合された後で本流を取得すると、マージコミットが新たに追加されて図 5-14 のような状態になります。

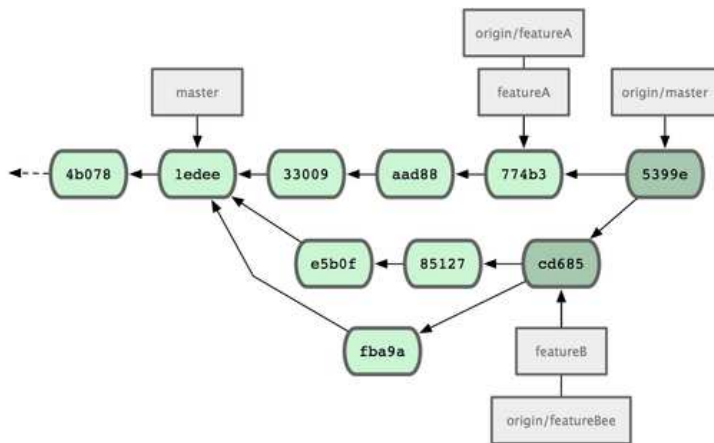


図 5-14. Jessica が両方のトピックブランチをマージしたあとのコミット履歴

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、図 5-15 のようになります。

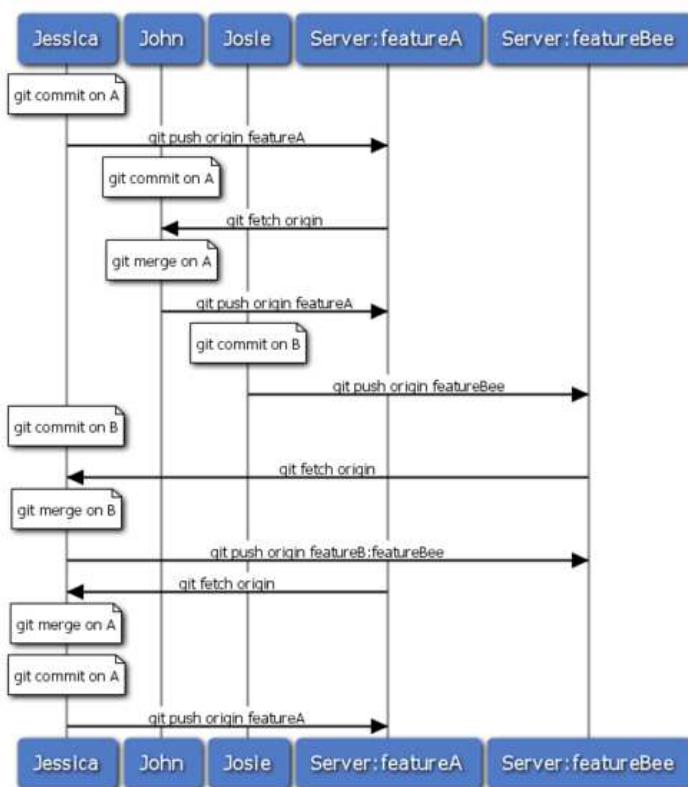


図 5-15. 管理されたチームでのワークフローの基本的な流れ

小規模な公開プロジェクト

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。最初の例では、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。repo.or.cz と GitHub はどちらもフォークに対応しており、多くのメンテナ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
$ (作業)
$ git commit
$ (作業)
$ git commit
```

`rebase -i` を使ってすべての作業をひとつのコミットにまとめたり、メンテナがレビューしやすいようにコミット内容を整理したりといったことも行うかもしれません。対話的なリベースの方法については第 6 章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 をリモートとして追加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)
```

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リモート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 myfork featureA
```

自分用のフォークに作業内容をプッシュし終わったら、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 (GitHub には "pull request" ボタンがあり、メンテナに自動的にメッセージを送ってくれます) 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://github.com/simblegit.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。図 5-16 を参照ください。

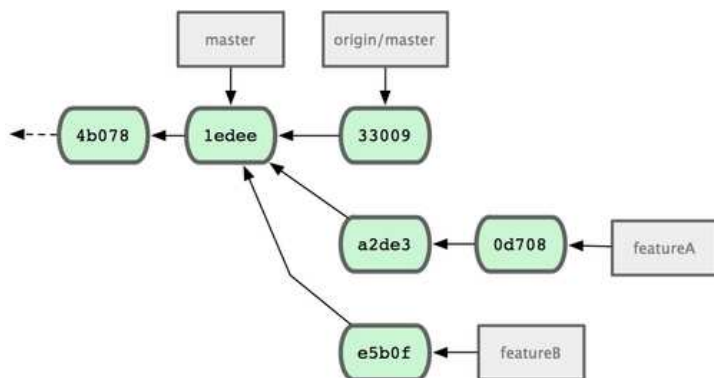


図 5-16. featureB に関する作業のコミット履歴

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、あなたの歴史は図 5-17 のように書き換えられました。

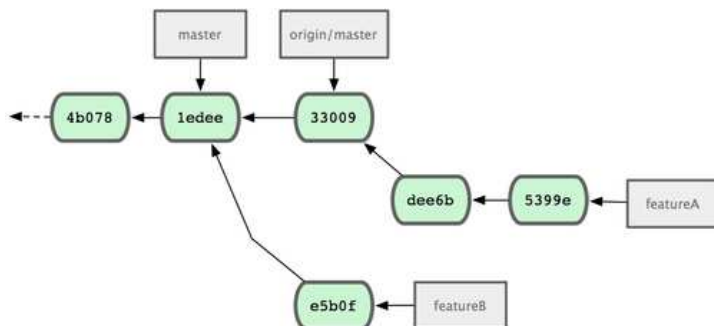


図 5-17. featureA の作業を終えた後のコミット履歴

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 (図 5-18 を参照ください)。

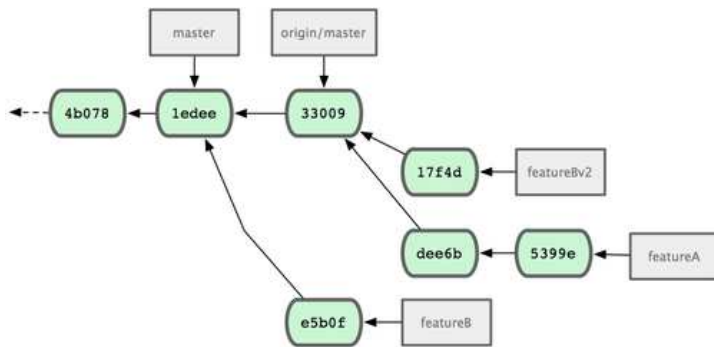


図 5-18. featureBv2 の作業を終えた後のコミット履歴

大規模な公開プロジェクト

多くの大規模プロジェクトでは、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。しかし、大規模なプロジェクトの多くは開発者用メーリングリストへのパッチの投稿を受け付けています。そこで、ここではそれを例にとりて話を進めます。

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ことです。次のセクションで実際にパッチを適用するところになれば、よりはっきりと実感するでしょう。

```
$ 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')
--
1.6.2.rc1.20.g8c5b.dirty
```

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 (lib/simplegit.rb 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、たまたま私が使ってるメールソフトが 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。

```
$ 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
```

あとは、Drafts フォルダに移動して To フィールドをメーリングリストのアドレスに変更し (おそらく CC には担当メンテナのアドレスを入れ)、送信できるようになりました。

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` 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 でのパッチの適用

`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、全体的に `patch` よりもこだわりを持った処理を行うのです。`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いくなります。

`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同じようなものです。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は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あ」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 git://github.com/onetimeguy/project.git
From git://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。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れるのかを完全な 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か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れば、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図 5-19 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、歴史は図 5-20 のようになります。

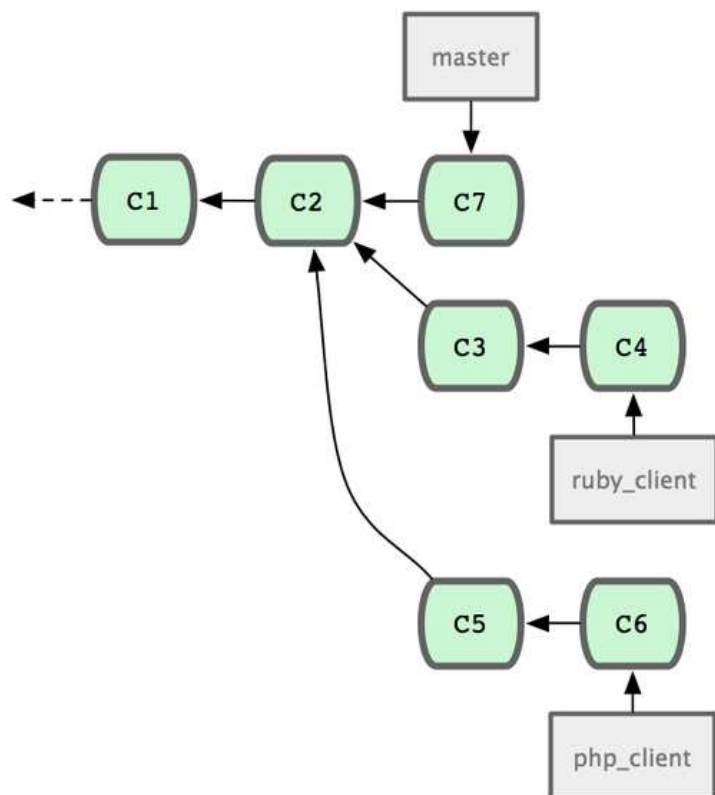


図 5-19. いくつかのトピックブランチを含む履歴

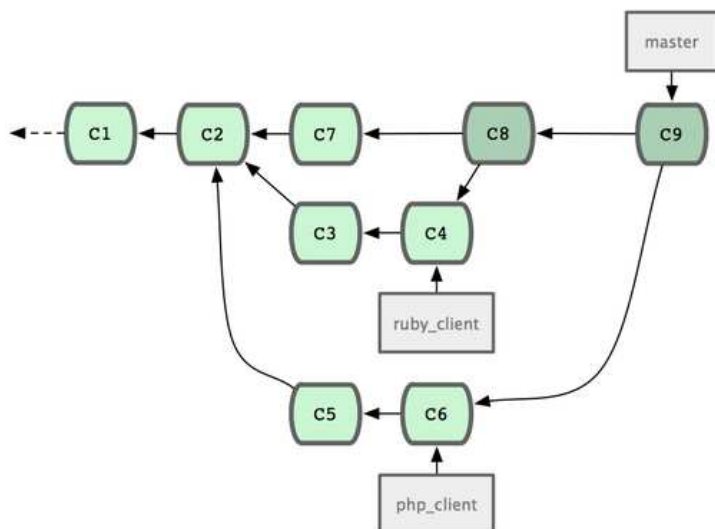


図 5-20. トピックブランチをマージした後の状態

これがおそらく一番シンプルなワークフローでしょうが、大規模なリポジトリやプロジェクトで作業をしていると問題が発生することもあります。

多人数で開発していたり大規模なプロジェクトに参加していたりする場合は、二段階以上のマージサイクルを使うこともあるでしょう。ここでは、長期間運用するブランチが master と develop のふたつあるものとします。master が更新されるのは安定版がリリースされるときだけで、新しいコードはすべて develop ブランチに統合されるという流れです。これらのブランチは、両方とも定期的に公開リポジトリにプッシュすることになります。新しいトピックブランチをマージする準備ができたなら (図 5-21)、それを develop にマージします (図 5-22)。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 (図 5-23)。

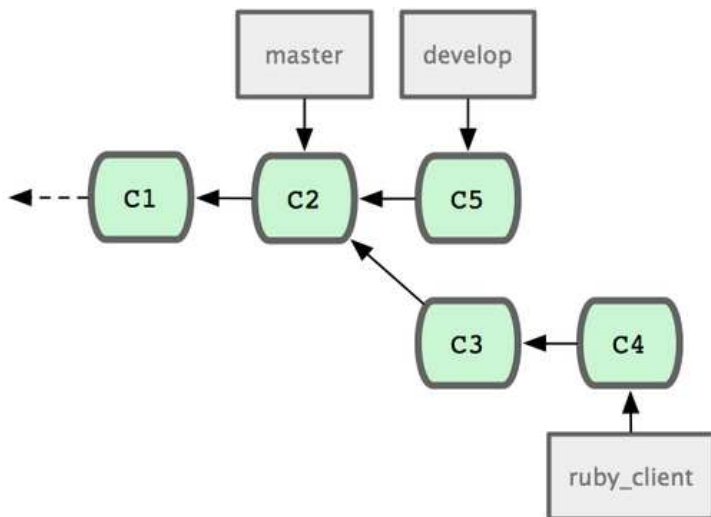


図 5-21. トピックブランチのマージ前

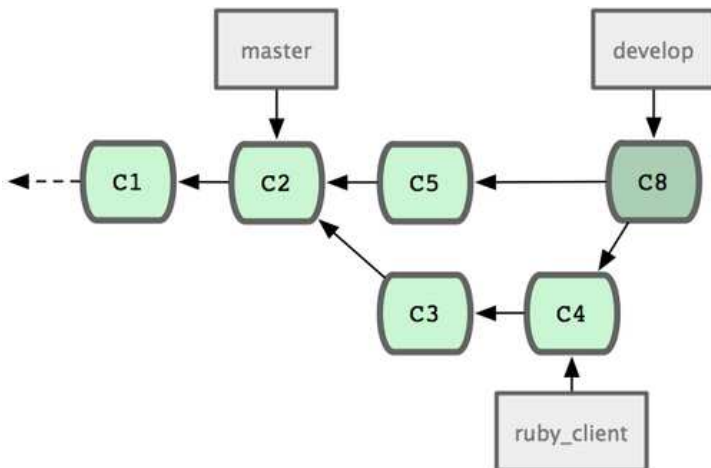


図 5-22. トピックブランチのマージ後

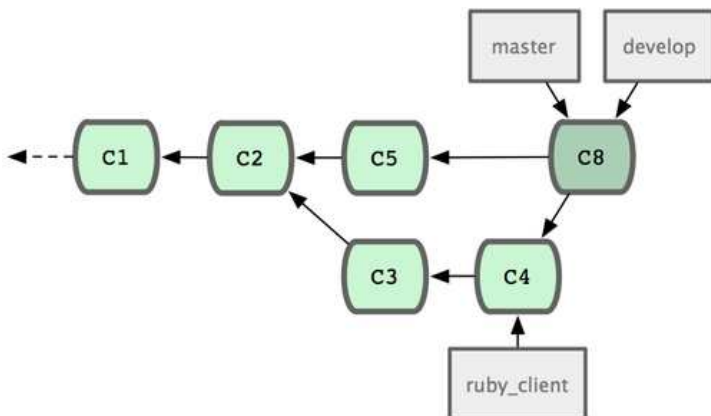


図 5-23. トピックブランチのリリース後

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なります。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 (図 5-24 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。だいたいよ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。

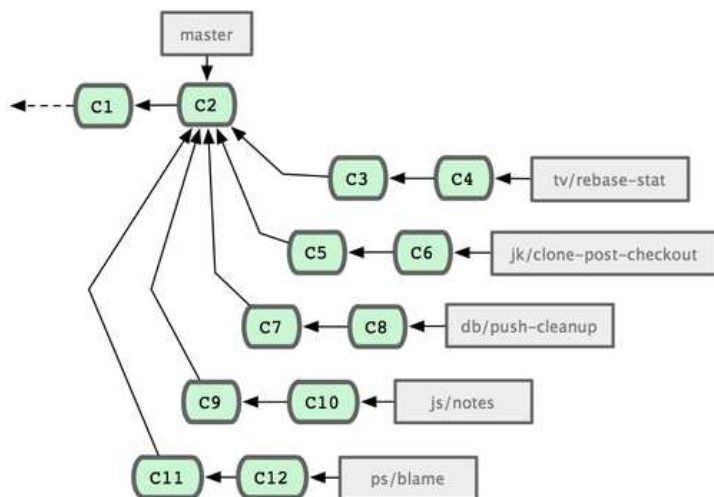


図 5-24. 複数のトピックブランチの並行管理

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 (図 5-25 を参照ください)。

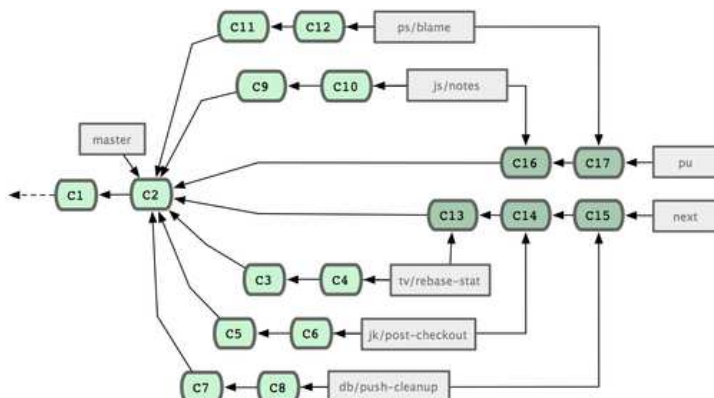


図 5-25. 常時移動する統合用ブランチへのトピックブランチのマージ

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てみたいか」「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す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。図 5-26 のようなプロジェクトを例にとりて考えましょう。

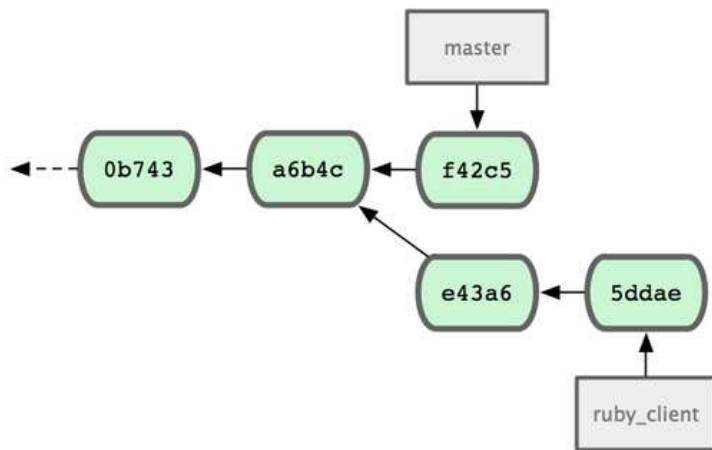


図 5-26. チェリーピック前の歴史

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 e43a6fd3e94888d76779ad79fb568ed180e5fcdf
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、歴史は図 5-27 のように変わりました。

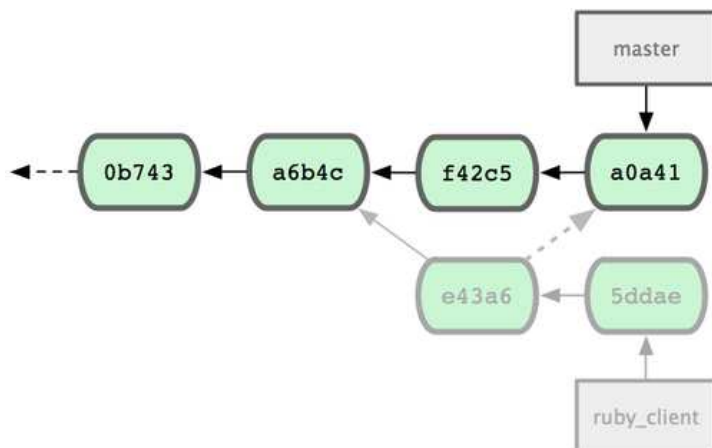


図 5-27. トピックブランチのコミットをチェリーピックした後の歴史

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。

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第 2 章で説明しました。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、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-gpg-pub 659ef797d181633c87ec71ac3f9ba29fe5775b92
```

`git push --tags` を実行すると、maintainer-gpg-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ブルで取得し、それを PGP にインポートすればいいのです。

```
$ git show maintainer-gpg-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、複雑な状況に対応するためのより強力なツールやヒントを学びます。これであなたは真の Git マスターとなることでしょう。

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。最も大規模な Git プロジェクトのひとつである Linux カーネルの場合は、40 文字のうち先頭の 12 文字を指定しないと一意性を確保できません。

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か?

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、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)。

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 1.2×10^{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 (100 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。これを五年間続けたとして、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第 9 章で説明します。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、今実際に何が起き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ができます。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 d921970^2
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を持っている場合など「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?」といった疑問に答え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。図 6-1 のようなコミット履歴を例に考えましょう。

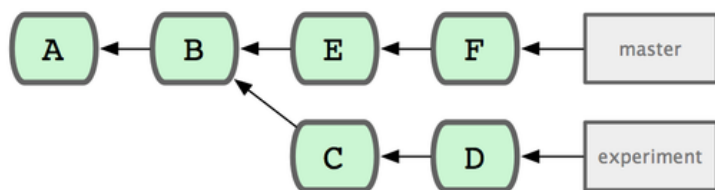


図 6-1. 範囲指定選択用の歴史の例

`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 からのはたどれるけれど、master からの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ことです。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。図 6-1 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。

ファイルのステージとその取り消し

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` を使用すること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、そのうち的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なになります。

```

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。

最後に、この対話的追加モードを使わずに部分的なステージを行いたい場合は、コマンドラインから git add -p あるいは git add --patch を実行すれば同じことができます。

作業を隠す

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
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   index.html
#
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#

```

```
#      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` 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
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#
#       modified:   index.html
#       modified:   lib/simplegit.rb
#
```

Git がファイルを変更して、未コミットのファイルが先ほどスタックに隠したときと同じ状態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
#
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#
#       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。

隠した内容の適用の取り消し

隠した変更を適用して何らかの作業をした後に、先ほどの適用を取り消してしまいたくなることもあるでしょう。そんなときに使えるような `stash unapply` コマンドは `git` にはありませんが、同じような操作をすることはできます。適用した変更を表すパッチを取得して、それを逆に適用すればいいのです。

```
$ git stash show -p stash@{0} | git apply -R
```

名前を指定しなければ、`Git` は直近に隠した変更を使うものとみなします。

```
$ git stash show -p | git apply -R
```

次の例のようにエイリアスを作れば、`git` に `stash-unapply` コマンドを追加したのと事実上同じことになります。

```
$ git config --global alias.stash-unapply '!git stash show -p | git apply -R'
$ git stash
$ #... 何か作業をして ...
$ git stash-unapply
```

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
Switched to a new branch "testchanges"
# On branch testchanges
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   index.html
#
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#
#       modified:   lib/simplegit.rb
#
Dropped refs/stash@{0} (f0dfc4d5dc332d1cee34a634182e168c4efc3359)
```

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。

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いことが多々あります。`Git` のすばらしい点のひとつは、何をどうするかを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
# e, edit = use commit, but stop for amending
# s, squash = use commit, but meld into previous commit
#
# If you remove a line here THAT COMMIT WILL BE LOST.
# However, if you remove everything, the rebase will be aborted.
#
```

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 7482e0d... updated the gemspec to hopefully work better
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
# e, edit = use commit, but stop for amending
# s, squash = use commit, but meld into previous commit
#
# If you remove a line here THAT COMMIT WILL BE LOST.
# However, if you remove everything, the rebase will be aborted.
#
```

「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
```

そして、スクリプトからコマンドラインに戻ってきたらそのコミットをリセットし、リセットされた変更を複数のコミットに分割します。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。ここで、必要なファイルをステージしてコミットしていきます。すべての処理が終われば、`git rebase --continue` を実行します。

```
$ 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、間違え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 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。

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が、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。最近私は `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ました。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 setObject:
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、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、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
f24d3c6ebcf639b1a3814550e62d60b8e68a8e4 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 を返してどこがおかしいときに 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て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い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ますようになります。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

Rack ライブラリ (Ruby のウェブサーバーゲートウェイインターフェイス) を自分のプロジェクトに取り込むことになったとしましょう。手元で変更を加えるかもしれませんが、本家で更新があった場合にはそれを取り込み続けるつもりです。まず最初にしなければならないことは、外部のリポジトリをサブディレクトリにクローンすることです。外部のプロジェクトをサブモジュールとして追加するには `git submodule add` コマンドを使用します。

```
$ git submodule add git://github.com/chneukirchen/rack.git rack
Initialized empty Git repository in /opt/subtest/rack/.git/
remote: Counting objects: 3181, done.
remote: Compressing objects: 100% (1534/1534), done.
remote: Total 3181 (delta 1951), reused 2623 (delta 1603)
Receiving objects: 100% (3181/3181), 675.42 KiB | 422 KiB/s, done.
Resolving deltas: 100% (1951/1951), done.
```

これで、プロジェクト内の rack サブディレクトリに Rack プロジェクトが取り込まれました。このサブディレクトリに入って変更を加えたり、書き込み権限のあるリモートリポジトリを追加してそこに変更をプッシュしたり、本家のリポジトリの内容を取得してマージしたり、さまざまなことができるようになります。サブモジュールを追加した直後に `git status` を実行すると、二つのものが見られます。

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

まず気づくのが `.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。

```
$ cat .gitmodules
[submodule "rack"]
    path = rack
    url = git://github.com/chneukirchen/rack.git
```

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。

`git status` の出力に、もうひとつ rack というエントリが含まれています。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 rack
diff --git a/rack b/rack
new file mode 160000
```

```
index 0000000..08d709f
--- /dev/null
+++ b/rack
@@ -0,0 +1 @@
+Subproject commit 08d709f78b8c5b0fbeb7821e37fa53e69afcf433
```

rack 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。このサブディレクトリ内に変更を加えてコミットすると、親プロジェクト側で HEAD が変わったことを検知し、実際の作業内容をコミットとして記録します。そうすることで、他の人がこのプロジェクトをクローンしたときに正しく環境を作れるようになります。

ここがサブモジュールのポイントです。サブモジュールは、それが場所の実際のコミットとして記録され、master やその他の参照として記録することはできません。

コミットすると、このようになります。

```
$ git commit -m 'first commit with submodule rack'
[master 0550271] first commit with submodule rack
 2 files changed, 4 insertions(+), 0 deletions(-)
 create mode 100644 .gitmodules
 create mode 160000 rack
```

rack 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。

rack ディレクトリを独立したプロジェクトとして扱い、ときどき親プロジェクトをアップデートして親プロジェクトの最新コミットにポインタを移動させることができます。すべての Git コマンドが、これらふたつのディレクトリで独立して使用可能です。

```
$ git log -1
commit 0550271328a0038865aad6331e620cd7238601bb
Author: Scott Chacon <schacon@gmail.com>
Date: Thu Apr 9 09:03:56 2009 -0700

    first commit with submodule rack

$ cd rack/
$ git log -1
commit 08d709f78b8c5b0fbeb7821e37fa53e69afcf433
Author: Christian Neukirchen <chneukirchen@gmail.com>
Date: Wed Mar 25 14:49:04 2009 +0100
```

Document version change

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。すると、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 git://github.com/schacon/myproject.git
Initialized empty Git repository in /opt/myproject/.git/
remote: Counting objects: 6, done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 6 (delta 0), reused 0 (delta 0)
Receiving objects: 100% (6/6), done.
$ cd myproject
$ ls -l
total 8
-rw-r--r-- 1 schacon admin  3 Apr  9 09:11 README
drwxr-xr-x 2 schacon admin 68 Apr  9 09:11 rack
$ ls rack/
$
```

rack ディレクトリは存在しますが、中身がからっぽです。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 'rack' (git://github.com/chneukirchen/rack.git) registered for path 'rack'
$ git submodule update
Initialized empty Git repository in /opt/myproject/rack/.git/
remote: Counting objects: 3181, done.
remote: Compressing objects: 100% (1534/1534), done.
remote: Total 3181 (delta 1951), reused 2623 (delta 1603)
Receiving objects: 100% (3181/3181), 675.42 KiB | 173 KiB/s, done.
Resolving deltas: 100% (1951/1951), done.
Submodule path 'rack': checked out '08d709f78b8c5b0fbeb7821e37fa53e69afcf433'
```

これで、サブディレクトリ rack 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。別の開発者が rack のコードを変更してコミットしたときにそれを取り込んでマージするには、もう少し付け加えます。

```
$ git merge origin/master
Updating 0550271..85a3eee
Fast forward
 rack |    2 +-
 1 files changed, 1 insertions(+), 1 deletions(-)
[master*]$ git status
# On branch master
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   rack
#
```

このマージで、サブモジュールが指すポインタの位置が変わりました。しかしサブモジュールディレクトリ内のコードは更新されていません。つまり、作業ディレクトリ内でダーティな状態になっています。

```
$ git diff
diff --git a/rack b/rack
index 6c5e70b..08d709f 160000
--- a/rack
+++ b/rack
@@ -1,1 @@
-Subproject commit 6c5e70b984a60b3cecd395edd5b48a7575bf58e0
+Subproject commit 08d709f78b8c5b0fbeb7821e37fa53e69afcf433
```

これは、サブモジュールのポインタが指す位置と実際のサブモジュールディレクトリの中身が異なるからです。これを修正するには、ふたたび `git submodule update` を実行します。

```
$ git submodule update
remote: Counting objects: 5, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 1), reused 2 (delta 0)
Unpacking objects: 100% (3/3), done.
From git@github.com:schacon/rack
 08d709f..6c5e70b master    -> origin/master
Submodule path 'rack': checked out '6c5e70b984a60b3cecd395edd5b48a7575bf58e0'
```

サブモジュールの変更をプロジェクトに取り込んだときには、毎回これをしなければなりません。ちょっと奇妙ですが、これでうまく動作します。

よくある問題が、開発者がサブモジュール内でローカルに変更を加えたけれどそれを公開サーバーにプッシュしていないときに起こります。ポインタの指す先を非公開の状態にしたまま、それを親プロジェクトにプッシュしてしまうと、他の開発者が `git submodule update` をしたときにサブモジュールが参照するコミットを見つけられなくなります。そのコミットは最初の開発者の環境にしか存在しないからです。この状態になると、次のようなエラーとなります。

```
$ git submodule update
fatal: reference isn't a tree: 6c5e70b984a60b3cecd395edd5b48a7575bf58e0
Unable to checkout '6c5e70b984a60b3cecd395edd5b48a7575bf58e0' in submodule path 'rack'
```

サブモジュールを最後に更新したのがいったい誰なのかを突き止めなければなりません。

```
$ git log -1 rack
commit 85a3eee996800fcfa91e2119372dd4172bf76678
Author: Scott Chacon <schacon@gmail.com>
Date: Thu Apr 9 09:19:14 2009 -0700
```

```
    added a submodule reference I will never make public. hahahahaha!
```

犯人がわかったら、メールで彼に怒鳴りつけてやりましょう。

親プロジェクト

時には、大規模なプロジェクトのサブディレクトリから今自分がいるチームに応じた組み合わせを取得したくなることもあるでしょう。これは、CVS や Subversion から移行した場合によくあることでしょう。モジュールを定義したりサブディレクトリのコレクションを定義していたりといったかつてのワークフローをそのまま維持したいというような状況です。

Git でこれと同じことをするためのよい方法は、それぞれのサブフォルダを別々の Git リポジトリにして、それらのサブモジュールとして含む親プロジェクトとなる Git リポジトリを作ることです。この方式の利点は、親プロジェクトのタグやブランチを活用してプロジェクト間の関係をより細やかに定義できることです。

サブモジュールでの問題

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。まず、サブモジュールのディレクトリで作業をするときはいつも以上に注意深くしなければなりません。git submodule update を実行すると、プロジェクトの特定のバージョンをチェックアウトしますが、それはブランチの中にあるものではありません。これを、切り離されたヘッド (detached head) と呼びます。つまり、HEAD が何らかの参照ではなく直接特定のコミットを指している状態です。通常は、ヘッドが切り離された状態で作業をしようとは思わないでしょう。手元の変更が簡単に失われてしまうからです。最初に submodule update し、作業用のブランチを作らずにサブモジュールディレクトリ内にコミットし、git submodule update を再び実行すると、親プロジェクトでコミットが何もなくても Git は手元の変更を断りなく上書きしてしまいます。技術的な意味では手元の作業は失われたわけではないのですが、それを指すブランチが存在しない以上、先ほどの作業を取り戻すのは困難です。

この問題を回避するには、サブモジュールのディレクトリで作業をするときに git checkout -b work などとしてブランチを作っておきます。次にサブモジュールを更新するときにあなたの作業は消えてしまいますが、少なくとも元に戻すためのポインタは残っています。

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 rack
Switched to a new branch "rack"
$ git submodule add git@github.com:schacon/rack.git rack
Initialized empty Git repository in /opt/myproj/rack/.git/
...
Receiving objects: 100% (3184/3184), 677.42 KiB | 34 KiB/s, done.
Resolving deltas: 100% (1952/1952), done.
$ git commit -am 'added rack submodule'
[rack cc49a69] added rack submodule
 2 files changed, 4 insertions(+), 0 deletions(-)
 create mode 100644 .gitmodules
 create mode 160000 rack
$ git checkout master
Switched to branch "master"
$ git status
# On branch master
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       rack/
```

これをどこか別の場所に移すか、削除しなければなりません。いずれにせよ、先ほどのブランチに戻ったときには改めてクローンしなおさなければならず、ローカルでの変更やプッシュしていないブランチは失われてしまうことになります。

最後にもうひとつ、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。rack の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な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 rack/
$ git submodule add git@github.com:schacon/rack.git rack
'rack' already exists in the index
```

まず最初に rack 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 rack
$ git submodule add git@github.com:schacon/rack.git rack
Initialized empty Git repository in /opt/testsub/rack/.git/
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 | 88 KiB/s, done.
Resolving deltas: 100% (1952/1952), 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: Untracked working tree file 'rack/AUTHORS' would be overwritten by merge.
```

いったん rack サブモジュールのディレクトリを別の場所に追い出してからでないと、サブモジュールを持たないブランチに切り替えることはできません。

```
$ mv rack /tmp/
$ git checkout master
Switched to branch "master"
$ ls
README rack
```

さて、戻ってきたら、空っぽの rack ディレクトリが得られました。ここで git submodule update を実行して再クローンするか、あるいは /tmp/rack ディレクトリを書き戻します。

サブツリーマージ

サブモジュールの仕組みに関する問題を見てきました。今度は同じ問題を解決するための別の方法を見ていきましょう。Git でマージを行うときには、何をマージしなければならないのかを Git がまず調べてそれに応じた適切なマージ手法を選択します。ふたつのブランチをマージするときに Git が使うのは、*再帰 (recursive)* 戦略です。三つ以上のブランチをマージするときには、Git は *たこ足 (octopus)* 戦略を選択します。どちらの戦略を使うかは、Git が自動的に選択します。再帰戦略は複雑な三方向のマージ (共通の先祖が複数あるなど) もこなせますが、ふたつのブランチしか処理できないからです。たこ足マージは三つ以上のブランチを扱うことができますが、難しいコンフリクトを避けるためにより慎重になります。そこで、三つ以上のブランチをマージするときのデフォルトの戦略として選ばれています。しかし、それ以外にも選べる戦略があります。そのひとつが *サブツリー (subtree)* マージで、これを使えば先ほどのサブプロジェクト問題に対応することができます。先ほどのセクションと同じような rack の取り込みを、サブツリーマージを用いて行う方法を紹介しましょう。

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たりその逆を行ったりするというものです。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理解して適切にマージを行います。驚くべきことです。

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 git@github.com:schacon/rack.git
$ git fetch rack_remote
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 git@github.com:schacon/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
```

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第 9 章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 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、戦略を指定するオプション -s subtree のほかに --squash オプションと --no-commit オプションを使います。

```
$ git checkout master
$ git merge --squash -s subtree --no-commit 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。

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
```

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いくつか紹介しました。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、Git をよりカスタマイズするための操作方法を扱います。重要な設定項目やフックシステムについても説明します。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

第 1 章で手短かにごらんいただいたように、`git config` コマンドで Git の設定をすることができます。まず最初にとこと言えば、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。

Git の設定については最初の章でちらっと説明しましたが、ここでもう一度振り返っておきます。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。まず最初に 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` 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。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。これらのファイルを手動で編集して正しい構文で値を追加することもできますが、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。大量のオプションがありますが、ここでは一般的に使われているものやワークフローに大きな影響を及ぼすものに絞っていくつかを紹介します。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。

```
$ git config --help
```

また、`git config` のマニュアルページには、利用できるすべてのオプションについて詳しい説明があります。

core.editor

コミットやタグのメッセージを編集するときに使うエディタは、ユーザーがデフォルトエディタとして設定したものとなります。デフォルトエディタが設定されていない場合は Vi エディタを使います。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を設定していない場合に Git が起動するエディタが Emacs に変わりました。

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 `$HOME/.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 $HOME/.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することができます。

```
$ 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（第 2 章で取り上げました）を作る場合は、GPG 署名用の鍵を登録しておくとう便利です。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 がそのファイルを見捨ててステージしないようになります。これについては第 2 章で説明しました。しかし、これらの内容をプロジェクトの外部で管理したい場合は、そのファイルがどこにあるのかを core.excludesfile で設定します。ここに設定する内容はファイルのパスです。ファイルの中身は .gitignore と同じ形式になります。

help.autocorrect

このオプションが使えるのは Git 1.6.1 以降だけです。Git 1.6 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 com
git: 'com' is not a git-command. See 'git --help'.

Did you mean this?
    commit
```

`help.autocorrect` を 1 にしておく、同じような場面でもし候補がひとつしかなければ自動的にそれを実行します。

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

あらかじめ指定しておけば、Git は自動的に大半の出力に色づけをします。何にどのような色をつけるかをこと細かに指定することもできますが、すべてをターミナルのデフォルト色設定にまかせるなら `color.ui` を `true` にします。

```
$ git config --global color.ui true
```

これを設定すると、出力がターミナルに送られる場合に Git がその出力を色づけします。ほかに `false` という値を指定することもでき、これは出力に決して色をつけません。また `always` を指定すると、すべての場合に色をつけます。すべての場合とは、Git コマンドをファイルにリダイレクトしたり他のコマンドにパイプでつないだりする場合も含みます。この設定項目は Git バージョン 1.5.5 で追加されました。それより前のバージョンを使っている場合は、すべての色設定を個別に指定しなければなりません。

`color.ui = 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い場合は Git コマンドに `--color` フラグを渡してカラーコードの使用を強制します。ふだんは `color.ui = true` 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、あるいはバージョンが古くて先ほどの設定が使えない場合は、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` を指定することができます。

```
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る、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` の出力でのメタ情報を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` あるいは `white` のいずれかです。先ほどの例の `bold` のように属性を指定することもできます。`bold`、`dim`、`ul`、`blink` および `reverse` のいずれかを指定できます。

`git config` のマニュアルページに、すべてのサブ設定がまとめられていますので参照ください。

外部のマージツールおよび Diff ツール

Git には `diff` の実装が組み込まれておりそれを使うことができますが、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、しかもフリーだからです。

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 をここからダウンロードします。

```
http://www.perforce.com/perforce/downloads/component.html
```

最初に、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。次のようなマージ用ラッパースクリプト `extMerge` を用意しました。これは、すべての引数を受け取ってバイナリをコールします。

```
$ 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.*.cmd でそのコマンドを実行する方法を指定し、mergetool.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.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
```

すべて設定し終わったら、

```
$ git diff 32d1776b1^ 32d1776b1
```

このような diff コマンドを実行すると、結果をコマンドラインに出力するかわりに P4Merge を立ち上げ、図 7-1 のようになります。

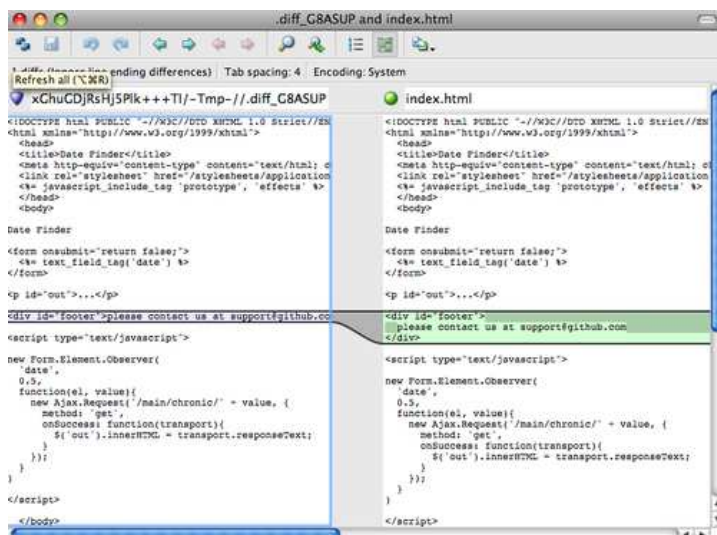


図 7-1. P4Merge

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、あとで `diff` ツールやマージツールを変更したくなったときにも簡単に変更することができます。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することができます。事前に設定が準備されているツールは `kdiff3`、`opendiff`、`tkdiff`、`meld`、`xxdiff`、`emerge`、`vimdiff` そして `gvimdiff` です。`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 で開発していたり、チームの中に Windows で開発している人がいたりといった場合に、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。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。

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することができます。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。またリポジトリにコミットする際には LF への変換を行います。

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四つの問題に対応するもので、そのうち二つはデフォルトで有効になっています。残りの二つはデフォルトでは有効になっていませんが、有効化することができます。

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 `trailing-space` と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設定は、行頭にある八文字以上の空白文字を見つける `indent-with-non-tab` と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、リベースのオプションにも適用されます。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、`rebase` に `--whitespace=fix` オプションをつけて実行すれば、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一貫性をチェックしません。各オブジェクトの 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 Git にチェックさせることもできますが、デフォルトでは毎回のプッシュ時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ができます。

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。その対策として、新しいバージョン (バージョン 1.6.1 以降) の Git では `receive.denyDeletes` を `true` に設定することができます。

```
$ 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

本章の最後で扱います。

Git の属性

設定項目の中には、パスにも指定できるものがあります。Git はその設定を、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するのです。これらのパス固有の設定は 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での diff の取得方法を指示したり、あるいはチェックインやチェックアウトの前に Git にフィルタリングさせたりすることができます。このセクションでは、Git プロジェクトでパスに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）指定して 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もあります。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 がビルド設定などをディスクに書き出したものです。すべて ASCII で構成されるので、理論上はこれはテキストファイルです。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をマージすることはできませんし、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 -crlf -diff
```

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。Git 1.6 系では、次のようなマクロを使うこともできます。これは `-crlf -diff` と同じ意味です。

```
*.pbxproj binary
```

バイナリファイルの差分

Git 1.6系では、バイナリファイルの差分を効果的に扱うためにGitの属性機能を使うことができます。通常のdiff機能を使って比較を行うことができるように、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。

これは素晴らしい機能ですがほとんど知られていないので、少し例をあげてみたいと思います。あなたはまず最初に人類にとっても最も厄介な問題のひとつを解決するためにこのテクニックを使いたいと思うでしょう。そう、Wordで作成した文書のバージョン管理です。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るにも係わらず、全ての人が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commit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.doc b/chapter1.doc
index 88839c4..4afcb7c 100644
Binary files a/chapter1.doc and b/chapter1.doc differ
```

これでは2つのバージョンをcheckoutしてそれらを自分で見比べてみない限り、比較することは出来ませんよね？ Gitの属性を使えば、うまく解決できます。`.gitattributes`に次の行を追加して下さい。

```
*.doc diff=word
```

これは、指定したパターン(.doc)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べきであるとGitに教えているのです。"word"フィルタとは何でしょうか？ それはあなたが用意しなければなりません。Word文書をテキストファイルに変換するプログラムとして `strings` を使うように次のようにGitを設定してみましょう。

```
$ git config diff.word.textconv strings
```

これで、.docという拡張子をもったファイルはそれぞれのファイルにstringsというプログラムとして定義された"word"フィルタを通してからdiffを取るべきだということをGitは知っていることになります。こうすることで、Wordファイルに対して直接差分を取るのではなく、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Gitリポジトリに登録した後、ある段落にいくつかの文章を追加して保存し、それから、変更箇所を確認するためにgit diffを実行しました。

```
$ git diff
diff --git a/chapter1.doc b/chapter1.doc
index c1c8a0a..b93c9e4 100644
--- a/chapter1.doc
+++ b/chapter1.doc
@@ -8,7 +8,8 @@ re going to cover Version Control Systems (VCS) and Git basics
 re going to cover how to get it and set it up for the first time if you don
 t already have it on your system.
 In Chapter Two we will go over basic Git usage - how to use Git for the 80%
 -s going on, modify stuff and contribute changes. If the book spontaneously
 +s going on, modify stuff and contribute changes. If the book spontaneously
 +Let's see if this works.
```

Gitは正しく、追加した"Let's see if this works"という文字列を首尾よく、かつ、簡潔に知らせてくれました。予想外の差分が表示されているので、完璧といえませんが、しかし、正しく動作しているとはいええます。あなたがWord文書をテキストファイルに変換するもっと良いプログラムを見付けられれば、よりよい結果を得られるでしょう。とはいえ、stringsはほとんどのMacとLinuxで動作するので、様々なバイナリフォーマットに試してみるのに、最初の選択肢としては良いと思います。

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。JPEGファイルに対するひとつの方法としては、EXIF情報(多くのファイルでメタデータとして使われています)を抽出するフィルタを使う方法です。exiftoolをダウンロードしインストールすれば、画像データをメタデータの形でテキストデータとして扱うことができます。従って、次のように設定すれば、画像データの差分をメタデータの差分という形で表示することができます。

```
$ echo '*.png diff=exif' >> .gitattributes
$ git config diff.exif.textconv exiftool
```

上記の設定をしてからプロジェクトで画像データを置き換えてgit diffと実行すれば、次のように表示されることになるでしょう。

```
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るでしょう。

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。これについてGitにおける主な問題は、Gitはまずファイルのチェックサムを生成するためにcommitした後にファイルに関する情報を変更できないという点です。しかし、commitするためにaddする前にファイルをcheckoutしremoveするという手順を踏めば、その時にファイルにテキストを追加することが可能です。Gitの属性はそうするための方法を2つ提供します。

ひとつめの方法として、ファイルの\$Id\$フィールドを自動的にblobのSHA-1 checksumを挿入するようにできます。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checkoutする時、Gitはこの置き換えを行うようになるでしょう。ただし、挿入されるチェックサムはcommitに対するものではなく、対象となるblobものであるという点に注意して下さい。

```
$ echo '*.txt ident' >> .gitattributes
$ echo '$Id$' > test.txt
```

次にtest.txtをcheckoutする時、Gitは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、このやりかたには制限があります。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、commit/checkout時にキーワード展開を行うためのフィルタを書いてやることで対応できます。このために"clean"と"smudge"フィルタがあります。特定のファイルに対して使用するフィルタを設定し、checkoutされる前("smudge" 図7-2参照)もしくはcommitされる前("clean" 図7-3参照)に指定したスクリプトが実行させるよう、.gitattributesファイルで設定できます。これらのフィルタはあらゆる種類の面白い内容を実行するように設定できます。

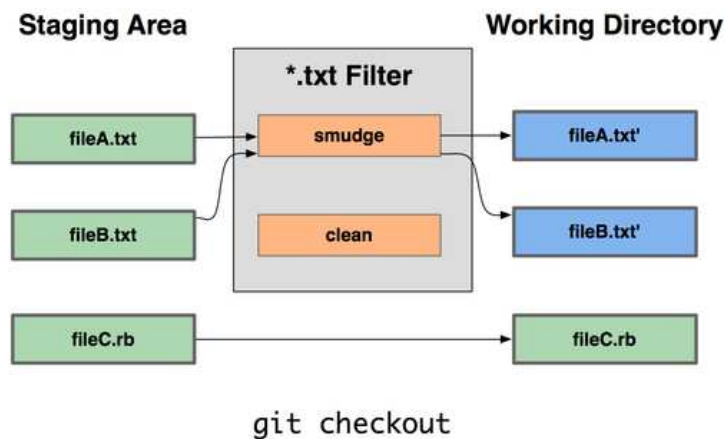


図7-2. checkoutする時に"smudge"フィルタを実行する

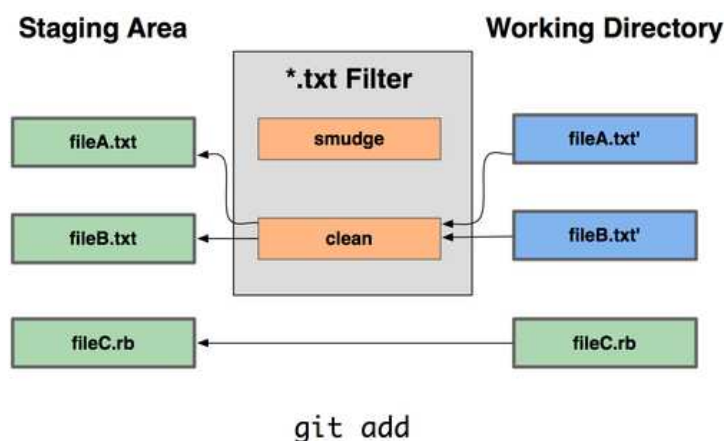


図7-3. ステージする時に"clean"フィルタを実行する。

この機能に対してオリジナルのcommitメッセージは簡単な例を与えてくれています。それはcommit前にあなたの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"indent"フィルタを実行するように、.gitattributesファイルにfilter属性を設定することができます。

```
*.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commitした時、Gitはcommit前にindentプログラムにファイルを通し、checkout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。この組み合わせでCのソースコードに対してcommit前にindentを通すことが効果的に行えます。

RCSスタイルの\$Date\$キーワード展開もまた別の興味深い例です。満足のいく形でこれを行うには、ファイル名を受け取って、プロジェクトの最新のcommitの日付を見付けだし、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。そのようなRubyスクリプトが以下です。

```
#!/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commitの日付を取得し、標準入力からの入力中のすべての\$Date\$文字列にその日付を追加し、結果を表示します。あなたのお気に入りのどのような言語でスクリプトを書くにしても、簡潔にすべきです。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が使うフィルタ(daterと呼びましょうか)を設定し、checkout時にexpand_dateが実行されるようにGitに教えてあげましょう。

```
$ git config filter.dater.smudge expand_date
$ git config filter.dater.clean 'perl -pe "s/¥¥¥$Date[^\¥¥¥$]*¥¥¥$/¥¥¥$Date¥¥¥$/'"
```

このPerlのスクリプトは、開始点に戻るために\$Date\$文字列内の他の文字列を削除します。さあ、フィルタの準備ができました。ファイルに\$Date\$キーワードを追加して新しいフィルタに仕事をさせるためにGitの属性を設定して、テストしてみましょう。

```
$ echo '# $Date$' > date_test.txt
$ echo 'date*.txt filter=dater' >> .gitattributes
```

これらの変更をcommitして再度ファイルをcheckoutすれば、キーワード展開が正しく行わ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
# $Date: Tue Apr 21 07:26:52 2009 -0700$
```

アプリケーションをカスタマイズするための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、注意が必要です。.gitattributesファイルはcommitされ、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)はそうはいきません。そう、すべての環境で動くとは限らないのです。あなたがこうしたフィルタをデザインする時、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を行うことができます。

export-ignore

アーカイブを生成するとき、ある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することができます。プロジェクトにはcheckin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を設定してやることができます。

例えば、test/ディレクトリ以下にいくつかのテストファイルがあって、それらをプロジェクトのtarballには含めたくないとしましょう。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

アーカイブ作成時にできる別のこととして、いくつかの簡単なキーワード展開があります。第2章で紹介した--pretty=formatで指定でき

フォーマット指定子とともに\$Format:\$文字列をファイルに追加することができます。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を実行した時にそれを最新のcommitの日付に変換したい場合、次のように設定します。

```
$ echo 'Last commit date: $Format:%cd$' > LAST_COMMIT
$ echo "LAST_COMMIT export-subst" >> .gitattributes
$ git add LAST_COMMIT .gitattributes
$ git commit -am 'adding LAST_COMMIT file for archives'
```

git archiveを実行したあと、アーカイブを展開すると、LAST_COMMITは以下のような内容になっているでしょう。

```
$ cat LAST_COMMIT
Last commit date: $Format:Tue Apr 21 08:38:48 2009 -0700$
```

マージの戦略

Git属性を使えば、プロジェクトにある指定したファイルに対して異なるマージ戦略を使うようにすることができます。とても有効なオプションのひとつは、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です。

これはブランチを分岐させ特別な作業をしている時、そのブランチでの変更をマージさせたいが、いくつか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
```

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スクリプトを叩く方法があります。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に、サーバーサイドフックはプッシュされたコミットを受け取るといったサーバーでの操作に利用されます。これらのフックをさまざまな理由に用いることができます。ここではそのうちのいくつかをご紹介します。

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サブディレクトリに格納されています。一般的なプロジェクトでは、.git/hooksがそれにあたります。Gitはデフォルトでこのディレクトリに例となるスクリプトを生成します。それらの多くはそのままでも十分有用ですし、引数も記載されています。全ての例は基本的にシェルスクリプトで書かれています。いくつかPerlを含むものもありますが、適切に命名されたそれらの実行可能スクリプトはうまく動きます。RubyやPython等で自作していただいてもかまいません。バージョン1.6以降のGitの場合、それらのフックファイルの末尾は.sampleとなっていますので適時リネームしてください。バージョン1.6以前のGitの場合ファイル名は適切ですが実行可能にはなっていません。

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適切な名前の実行可能なファイルを配置する必要があります。これによってファイルが呼び出されることになります。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。

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。これはいまだコミットされるであろうスナップショットを検査したり、何かし忘れた事を確認したり、事前にテストを実行したり、何かしらコードを検査する目的で使用されます。git commit --no-verifyで回避することもできますが、このフックから0でない値が返るとコミットが中断されます。コーディングスタイルの検査（lintを実行する等）や、空白文字の追跡（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を検査したり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。コミットの作者がそれを目にする前にデフォルトメッセージを編集することができます。このフックはオプションを必要とします: 現在ま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SHA-1が必要です。このフックは普段のコミットにおいてあまり有用ではありませんが、テンプレートの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で挿入されるコミットにおいて効果を発揮します。テンプレートのコミットメッセージと組み合わせて、動的な情報をプログラムで挿入することができます。

`commit-msg`フックも、現在の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持つ必要があります。このスクリプトが0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ので、プロジェクトの状態や許可待ちになっているコミットメッセージを有効にすることができます。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、`post-commit`フックが実行されます。パラメータは必要無く、`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。

コミットワークフロークライアントサイドスクリプトはあらゆるワークフローに使用することができます。clone中にスクリプトが転送される事はありませんが、これらはしばしばサーバー側で決められたポリシーを強制する目的で使用されます。これらのスクリプトは開発者を支援するために存在するのですから、いつでもオーバーライドされたり変更されたりすることがありえるとしても開発者らによってセッアップされ、メンテナンスされてしかるべきです。

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することができます。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、ふだんの作業でこのコマンドを使っていない場合は次のセクションを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場合は、ここで説明する内容が有用になるかもしれません。

まず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だけ受け取ります。コミットメッセージを含む一時ファイル名です。このスクリプトがゼロ以外の値で終了した場合、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することができます。

`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。これは引数を受け取らず、パッチが適用された後に実行されます。このフックを使うと、パッチ適用後の状態をコミットする前に調べることができます。つまり、このスクリプトでテストを実行したり、その他の調査をしたりといったことができますということです。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。これを使うと、グループのメンバーやその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、ゼロ以外を返すとその処理を中断させることができます。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することができます。Gitに含まれているサンプルの `pre-rebase` フックがちょうどこの働きをします。ただしこのサンプルは、公開ブランチの名前が `next` であることを想定したものです。実際に使っている安定版公開ブランチの名前に変更する必要があるでしょう。

`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することができます。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や自動生成ドキュメントなどを作業ディレクトリに取り込むといった処理です。

最後に説明する `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することができます。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。

サーバーサイドフック

クライアントサイドフックの他に、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。これは、システム管理者がプロジェクトのポリシーを強制させるために使うものです。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、プッシュに関するポリシーをここで設定することができますということです。

pre-receive および post-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、プッシュしてきたユーザーがそれらのファイルに対する適切なアクセス権を持っているかを調べたりといったことができます。

`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。`pre-receive` フックと同様、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、メーリングリストへの投稿や継続的インテグレーション

ンサーバーへの通知、チケット追跡システムの更新などの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ができます。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することができません。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。

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。

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、プッシュは fast-forward 限定にし、そしてプロジェクト内の各サブディレクトリに対して特定のユーザーだけが変更を加えられるようにするというものです。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、そして実際にそのポリシーを実施するためのサーバースクリプトを作成します。

スクリプトは Ruby を使って書きます。その理由のひとつは私が Ruby を好きなこと、そしてもうひとつの理由はその他のスクリプト言語の疑似コードとしてもそれっぽく見えるであろうということです。Ruby 使いじゃなくても、きっとコードの大まかな流れは追えるはずです。しかし、Ruby 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 で書かれているので、それらの言語のサンプルも大量に見ることができます。

サーバーサイドフック

サーバーサイドの作業は、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るもので、プッシュされる参照と操作前のブランチのリビジョン、そしてプッシュされる新しいリビジョンを受け取ります。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させている場合は、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。スクリプトは、まずこれらの情報を取得するところから始まります。

```
#!/usr/bin/env ruby

$refname = ARGV[0]
$soldrev = ARGV[1]
$newrev = ARGV[2]
$user = ENV['USER']

puts "Enforcing Policies... ¥n(#{ $refname }) (#{ $soldrev[0,6] }) (#{ $newrev[0,6] })"
```

ああ、グローバル変数を使ってるとかいうツツコミは勘弁してください。このほうが説明が楽なので。

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その文字列があるかどうかを調べ、もしなければゼロ以外の値で終了して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 \$soldrev の内容を git rev-list という低レベル Git 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 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 を取得し、個々のメッセージを取り出し、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

低レベルコマンド `git cat-file` を使います。低レベルコマンドについては第 9 章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 cfd3bf379e4f8dba8717dee55aab78aef7f4daf
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
```

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、ゼロ以外の値で終了させます。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を変更できるのかを指定できるようにしてみましょう。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だけにしか変更をプッシュできないユーザーもいる、といった仕組みです。これを実施するには、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、プッシュされたコミットにどの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がそれらのファイルを変更する権限があるのかどうかを判断します。

まずは ACL を作る場所から始めましょう。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すもので、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、全体にアクセスする管理者と `doc` ディレクトリにアクセスするドキュメント担当者、そして `lib` と `tests` サブ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。ユーザー名が配列のキー、そのユーザーが書き込み権を持つパスの配列が対応する値となります。

```
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することができるのかどうかを判断できます。

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 (第 2 章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、そのユーザーが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.index(access_path) == 0) # 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

```

それほど難しい処理ではありません。まず最初に `git rev-list` でコミットの一覧を取得し、それぞれに対してどのファイルが変更されるのかを調べ、ユーザーがそのファイルを変更する権限があることを確かめています。Ruby を知らない人にはわかりにくいところがあるとすれば `path.index(access_path) == 0` でしょうか。これは、パスが `access_path` で始まるときに真となります。つまり、`access_path` がパスの一部に含まれるのではなく、パスがそれで始まっていることを確認しています。

これで、まずい形式のコミットメッセージや権利のない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。

Fast-Forward なプッシュへの限定

最後は、fast-forward なプッシュに限るという仕組みです。Git バージョン 1.6 以降には `receive.denyDeletes` および `receive.denyNonFastForwards` という設定項目がありますが、これをフックで記述しておけば、古いバージョンの Git でも動作します。また、特定のユーザーにだけこの制約を加えたいなどといった変更にも対応できます。

これを調べるには、旧リビジョンからたどれるすべてのコミットについて、新リビジョンから到達できないものがないかどうかを探します。もしひとつもなければ、それは fast-forward なプッシュです。ひとつでも見つければ、却下することになります。

```
# enforces fast-forward only pushes
def check_fast_forward
  missed_refs = `git rev-list #{$newrev}..#{$oldrev}`
  missed_ref_count = missed_refs.split("\n").size
  if missed_ref_count > 0
    puts "[POLICY] Cannot push a non fast-forward reference"
    exit 1
  end
end

check_fast_forward
```

これですべてがととのいました。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、fast-forward ではない参照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] Cannot push a non-fast-forward reference
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、いくつか興味深い点があります。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「スクリプトから標準出力に送った内容は、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] Cannot push a non fast-forward reference
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。

さらに、もしコミットメッセージに適切な ref が含まれていなければ、それを示す次のようなエラーメッセージが表示されるでしょう。

```
[POLICY] Your message is not formatted correctly
```

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
```

以上です。この update スクリプトが動いてさえいれば、もう二度とリポジトリが汚されることはありません。コミットメッセージは決まりどおりのきちんとしたものになるし、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。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することができます。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、別の何らかの方法で各ユーザーにスクリプトを配布しなければなりません。各ユーザーはそれを .git/hooks にコピーし、実行可能にします。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ないのです。

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。そして、サーバーに却下されないようにコミットメッセージの書式を調べるのです。そのためには 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 files 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することができます。

```
#!/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
    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。それ以外はまったく同じように動作します。ただ、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が、これは多少特殊です。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 fast-forward ではないプッシュをできないようにしているので、それ以外にあり得るのは、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
end

```

このスクリプトでは、第 6 章の「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コミットの親のいずれかからアクセスできないコミットを探します。

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う限り、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

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 を使っていることも時々あるでしょうし、多くの場合 Subversion が使われているのではないかと思います。この章の前半では、まず Subversion と Git を繋ぐ双方向ゲートウェイである `git svn` について説明します。

どこかの時点で、プロジェクトで Git を使うようにしたくなることもあるでしょう。この章の後半では、プロジェクトの VCS を Git へ移行する方法について説明します。Subversion と Perforce からの移行について説明したあと、特殊なケースにおいてスクリプトを使ったインポートの方法を説明します。

Git と Subversion

現在のところ、オープンソースや企業のプロジェクトの大多数が、ソースコードの管理に Subversion を利用しています。Subversion は最も人気のあるオープンソースの VCS で、10 年近く前から使われています。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 の素晴ら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징・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 の素晴ら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。すべてはここから始めることができます。この後に続くコマンドはかなりのたくさんあるので、いくつかのワークフローを通して一般的なものから身につけていきましょう。

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 ほど洗練されてはいません。ローカルでのブランチ作成やマージは簡単にできますが、作業内容をリベースするなどして歴史をできるだけ直線に保つようにし、Git リモ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。

準備

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、`svnsync` というツールを使います。最近のバージョンの Subversion、少なくとも 1.4 以降に付属しているツールです。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。

```
$ svn sync file:///tmp/test-svn
Committed revision 1.
Copied properties for revision 1.
Committed revision 2.
Copied properties for revision 2.
Committed revision 3.
...
```

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、この処理には約一時間かかります。総コミット数はたかだか 100 にも満たないにもかかわらず。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。

はじめましょう

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 /Users/schacon/projects/testsvnsync/svn/.git/
r1 = b4e387bc68740b5af56c2a5faf4003ae42bd135c (trunk)
  A   m4/acx_pthread.m4
  A   m4/stl_hash.m4
...
r75 = d1957f3b307922124eec6314e15bcda59e3d9610 (trunk)
Found possible branch point: file:///tmp/test-svn/trunk => ¥
  file:///tmp/test-svn /branches/my-calc-branch, 75
Found branch parent: (my-calc-branch) d1957f3b307922124eec6314e15bcda59e3d9610
Following parent with do_switch
Successfully followed parent
r76 = 8624824ecc0badd73f40ea2f01fce51894189b01 (my-calc-branch)
Checked out HEAD:
  file:///tmp/test-svn/branches/my-calc-branch r76
```

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ないので、数分しかかかりません。しかし、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
  my-calc-branch
  tags/2.0.2
  tags/release-2.0.1
  tags/release-2.0.2
  tags/release-2.0.2rc1
  trunk
```

このツールがリモート参照を取り込むときの名前空間が通常と異なることに注意しましょう。Git リポジトリのクローンを作成した場合は、リモートサーバー上のすべてのブランチが `origin/[branch]` のような形式で取り込まれます。つまりリモートの名前で名前空間が作られます。しかし、`git svn` はリモートが複数あることを想定しておらず、すべてのリモートサーバーを名前空間なしに保存します。Git のコマンド `show-ref` を使うと、すべての参照名を完全な形式で見ることができます。

```
$ git show-ref
1cbd4904d9982f386d87f88fce1c24ad7c0f0471 refs/heads/master
aee1ecc26318164f355a883f5d99cff0c852d3c4 refs/remotes/my-calc-branch
03d09b0e2aad427e34a6d50ff147128e76c0e0f5 refs/remotes/tags/2.0.2
50d02cc0adc9da4319eeba0900430ba219b9c376 refs/remotes/tags/release-2.0.1
4caaa711a50c77879a91b8b90380060f672745cb refs/remotes/tags/release-2.0.2
1c4cb508144c513ff1214c3488abe66dcb92916f refs/remotes/tags/release-2.0.2rc1
1cbd4904d9982f386d87f88fce1c24ad7c0f0471 refs/remotes/trunk
```

通常の Git リポジトリは、このようになります。

```
$ git show-ref
83e38c7a0af325a9722f2fdc56b10188806d83a1 refs/heads/master
3e15e38c198baac84223acfc6224bb8b99ff2281 refs/remotes/gitserver/master
0a30dd3b0c795b80212ae723640d4e5d48cabdff refs/remotes/origin/master
25812380387fdd55f916652be4881c6f11600d6f refs/remotes/origin/testing
```

2 つのリモートサーバーがあり、一方の gitserver には master ブランチが、そしてもう一方の origin には master と testing の 2 つのブランチがあります。

サンプルのリモート参照が git svn でどのように取り込まれたかに注目しましょう。タグはリモートブランチとして取り込まれており、Git のタグにはなっていません。Subversion から取り込んだ内容は、まるで tags という名前のリモートからブランチを取り込んだように見えます。

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 97031e5] Adding git-svn instructions to the README
1 files changed, 1 insertions(+), 1 deletions(-)
```

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目しましょう。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 r79
M      README.txt
r79 = 938b1a547c2cc92033b74d32030e86468294a5c8 (trunk)
No changes between current HEAD and refs/remotes/trunk
Resetting to the latest refs/remotes/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 サーバーを動じ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 938b1a547c2cc92033b74d32030e86468294a5c8
Author: schacon <schacon@4c93b258-373f-11de-be05-5f7a86268029>
Date: Sat May 2 22:06:44 2009 +0000
```

```
Adding git-svn instructions to the README
```

```
git-svn-id: file:///tmp/test-svn/trunk@79 4c93b258-373f-11de-be05-5f7a86268029
```

元のコミットの SHA チェックサムが 97031e5 で始まっていたのに対して今は 938b1a5 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 ...
Merge conflict during commit: Your file or directory 'README.txt' is probably ¥
out-of-date: resource out of date; try updating at /Users/schacon/libexec/git-¥
core/git-svn line 482
```

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
M      README.txt
r80 = ff829ab914e8775c7c025d741beb3d523ee30bc4 (trunk)
First, rewinding head to replay your work on top of it...
Applying: first user change
```

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 r81
M      README.txt
r81 = 456cbe6337abe49154db70106d1836bc1332deed (trunk)
No changes between current HEAD and refs/remotes/trunk
Resetting to the latest refs/remotes/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 r84
M      autogen.sh
r83 = 8aa54a74d452f82eee10076ab2584c1fc424853b (trunk)
M      configure.ac
r84 = cdbac939211ccb18aa744e581e46563af5d962d0 (trunk)
W: d2f23b80f67aaaa1f6f5aaef48fce3263ac71a92 and refs/remotes/trunk differ, ¥
using rebase:
:100755 100755 efa5a59965fbbb5b2b0a12890f1b351bb5493c18 ¥
015e4c98c482f0fa71e4d5434338014530b37fa6 M  autogen.sh
First, rewinding head to replay your work on top of it...
Nothing to do.
```

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      generate_descriptor_proto.sh
```

```
r82 = bd16df9173e424c6f52c337ab6efa7f7643282f1 (trunk)
First, rewinding head to replay your work on top of it...
Fast-forwarded master to refs/remotes/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うです。

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、出力はこのようなになります。

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...
    M      CHANGES.txt
Committed r85
    M      CHANGES.txt
r85 = 4bfebeec434d156c36f2bcd18f4e3d97dc3269a2 (trunk)
No changes between current HEAD and refs/remotes/trunk
Resetting to the latest refs/remotes/trunk
COPYING.txt: locally modified
INSTALL.txt: locally modified
    M      COPYING.txt
    M      INSTALL.txt
Committed r86
    M      INSTALL.txt
    M      COPYING.txt
r86 = 2647f6b86ccfcaad4ec58c520e369ec81f7c283c (trunk)
No changes between current HEAD and refs/remotes/trunk
Resetting to the latest refs/remotes/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、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はできません。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 [branchname] を実行します。

```
$ git svn branch opera
Copying file:///tmp/test-svn/trunk at r87 to file:///tmp/test-svn/branches/opera...
Found possible branch point: file:///tmp/test-svn/trunk => ¥
    file:///tmp/test-svn/branches/opera, 87
Found branch parent: (opera) 1f6bfe471083cbca06ac8d4176f7ad4de0d62e5f
Following parent with do_switch
Successfully followed parent
r89 = 9b6fe0b90c5c9adf9165f700897518dbc54a7cbf (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が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/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 に書き戻すことにはなりますが 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だけとなります。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 | 2009-05-02 16:07:37 -0700 (Sat, 02 May 2009) | 2 lines

autogen change

-----
r86 | schacon | 2009-05-02 16:00:21 -0700 (Sat, 02 May 2009) | 2 lines

Merge branch 'experiment'

-----
r85 | schacon | 2009-05-02 16:00:09 -0700 (Sat, 02 May 2009) | 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则认为ればよいでしょう。

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。

Migrating to Git

If you have an existing codebase in another VCS but you've decided to start using Git, you must migrate your project one way or another. This section goes over some importers that are included with Git for common systems and then demonstrates how to develop your own custom importer.

Importing

You'll learn how to import data from two of the bigger professionally used SCM systems — Subversion and Perforce — both because they make up the majority of users I hear of who are currently switching, and because high-quality tools for both systems are distributed with Git.

Subversion

If you read the previous section about using `git svn`, you can easily use those instructions to `git svn clone` a repository; then, stop using the Subversion server, push to a new Git server, and start using that. If you want the history, you can accomplish that as quickly as you can pull the data out of the Subversion server (which may take a while).

However, the import isn't perfect; and because it will take so long, you may as well do it right. The first problem is the author information. In Subversion, each person committing has a user on the system who is recorded in the commit information. The examples in the previous section show `schacon` in some places, such as the `blame` output and the `git svn log`. If you want to map this to better Git author data, you need a mapping from the Subversion users to the Git authors. Create a file called `users.txt` that has this mapping in a format like this:

```
schacon = Scott Chacon <schacon@geemail.com>
selse = Someo Nelse <selse@geemail.com>
```

To get a list of the author names that SVN uses, you can run this:

```
$ svn log --xml | grep author | sort -u | perl -pe 's/.(?<./$1 = /'
```

That gives you the log output in XML format — you can look for the authors, create a unique list, and then strip out the XML. (Obviously this only works on a machine with `grep`, `sort`, and `perl` installed.) Then, redirect that output into your `users.txt` file so you can add the equivalent Git user data next to each entry.

You can provide this file to `git svn` to help it map the author data more accurately. You can also tell `git svn` not to include the metadata that Subversion normally imports, by passing `--no-metadata` to the `clone` or `init` command. This makes your `import` command look like this:

```
$ git-svn clone http://my-project.googlecode.com/svn/ ¥
--authors-file=users.txt --no-metadata -s my_project
```

Now you should have a nicer Subversion import in your `my_project` directory. Instead of commits that look like this

```
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
```

they look like this:

```
commit 03a8785f44c8ea5cdb0e8834b7c8e6c469be2ff2
Author: Scott Chacon <schacon@geemail.com>
Date:   Sun May 3 00:12:22 2009 +0000

    fixed install - go to trunk
```

Not only does the Author field look a lot better, but the `git-svn-id` is no longer there, either.

You need to do a bit of post-import cleanup. For one thing, you should clean up the weird references that `git svn` set up. First you'll move the tags so they're actual tags rather than strange remote branches, and then you'll move the rest of the branches so they're local.

To move the tags to be proper Git tags, run

```
$ cp -Rf .git/refs/remotes/tags/* .git/refs/tags/
$ rm -Rf .git/refs/remotes/tags
```

This takes the references that were remote branches that started with tag/ and makes them real (lightweight) tags.

Next, move the rest of the references under refs/remotes to be local branches:

```
$ cp -Rf .git/refs/remotes/* .git/refs/heads/
$ rm -Rf .git/refs/remotes
```

Now all the old branches are real Git branches and all the old tags are real Git tags. The last thing to do is add your new Git server as a remote and push to it. Because you want all your branches and tags to go up, you can run this:

```
$ git push origin --all
```

All your branches and tags should be on your new Git server in a nice, clean import.

Perforce

The next system you'll look at importing from is Perforce. A Perforce importer is also distributed with Git, but only in the contrib section of the source code — it isn't available by default like git svn. To run it, you must get the Git source code, which you can download from git.kernel.org:

```
$ git clone git://git.kernel.org/pub/scm/git/git.git
$ cd git/contrib/fast-import
```

In this fast-import directory, you should find an executable Python script named git-p4. You must have Python and the p4 tool installed on your machine for this import to work. For example, you'll import the Jam project from the Perforce Public Depot. To set up your client, you must export the P4PORT environment variable to point to the Perforce depot:

```
$ export P4PORT=public.perforce.com:1666
```

Run the git-p4 clone command to import the Jam project from the Perforce server, supplying the depot and project path and the path into which you want to import the project:

```
$ git-p4 clone //public/jam/src@all /opt/p4import
Importing from //public/jam/src@all into /opt/p4import
Reinitialized existing Git repository in /opt/p4import/.git/
Import destination: refs/remotes/p4/master
Importing revision 4409 (100%)
```

If you go to the /opt/p4import directory and run git log, you can see your imported work:

```
$ git log -2
commit 1fd4ec126171790efd2db83548b85b1bbbc07dc2
Author: Perforce staff <support@perforce.com>
Date: Thu Aug 19 10:18:45 2004 -0800

    Drop 'rc3' moniker of jam-2.5. Folded rc2 and rc3 RELNOTES into
    the main part of the document. Built new tar/zip balls.

    Only 16 months later.

[git-p4: depot-paths = "//public/jam/src/": change = 4409]

commit ca8870db541a23ed867f38847eda65bf4363371d
Author: Richard Geiger <rmg@perforce.com>
Date: Tue Apr 22 20:51:34 2003 -0800

    Update derived jamgram.c

[git-p4: depot-paths = "//public/jam/src/": change = 3108]
```

You can see the git-p4 identifier in each commit. It's fine to keep that identifier there, in case you need to reference the Perforce change number later. However, if you'd like to remove the identifier, now is the time to do so — before you start doing work on the new repository. You can use git filter-branch to remove the identifier strings en masse:

```
$ git filter-branch --msg-filter '
    sed -e "/^¥[git-p4:/d"
,
Rewrite 1fd4ec126171790efd2db83548b85b1bbbc07dc2 (123/123)
Ref 'refs/heads/master' was rewritten
```

If you run `git log`, you can see that all the SHA-1 checksums for the commits have changed, but the `git-p4` strings are no longer in the commit messages:

```
$ git log -2
commit 10a16d60cffca14d454a15c6164378f4082bc5b0
Author: Perforce staff <support@perforce.com>
Date: Thu Aug 19 10:18:45 2004 -0800

    Drop 'rc3' moniker of jam-2.5. Folded rc2 and rc3 RELNOTES into
    the main part of the document. Built new tar/zip balls.

    Only 16 months later.

commit 2b6c6db311dd76c34c66ec1c40a49405e6b527b2
Author: Richard Geiger <rmg@perforce.com>
Date: Tue Apr 22 20:51:34 2003 -0800

    Update derived jamgram.c
```

Your import is ready to push up to your new Git server.

A Custom Importer

If your system isn't Subversion or Perforce, you should look for an importer online — quality importers are available for CVS, Clear Case, Visual Source Safe, even a directory of archives. If none of these tools works for you, you have a rarer tool, or you otherwise need a more custom importing process, you should use `git fast-import`. This command reads simple instructions from stdin to write specific Git data. It's much easier to create Git objects this way than to run the raw Git commands or try to write the raw objects (see Chapter 9 for more information). This way, you can write an import script that reads the necessary information out of the system you're importing from and prints straightforward instructions to stdout. You can then run this program and pipe its output through `git fast-import`.

To quickly demonstrate, you'll write a simple importer. Suppose you work in `current`, you back up your project by occasionally copying the directory into a time-stamped `back_YYYY_MM_DD` backup directory, and you want to import this into Git. Your directory structure looks like this:

```
$ ls /opt/import_from
back_2009_01_02
back_2009_01_04
back_2009_01_14
back_2009_02_03
current
```

In order to import a Git directory, you need to review how Git stores its data. As you may remember, Git is fundamentally a linked list of commit objects that point to a snapshot of content. All you have to do is tell `fast-import` what the content snapshots are, what commit data points to them, and the order they go in. Your strategy will be to go through the snapshots one at a time and create commits with the contents of each directory, linking each commit back to the previous one.

As you did in the "An Example Git Enforced Policy" section of Chapter 7, we'll write this in Ruby, because it's what I generally work with and it tends to be easy to read. You can write this example pretty easily in anything you're familiar with — it just needs to print the appropriate information to stdout. And, if you are running on Windows, this means you'll need to take special care to not introduce carriage returns at the end your lines — `git fast-import` is very particular about just wanting line feeds (LF) not the carriage return line feeds (CRLF) that Windows uses.

To begin, you'll change into the target directory and identify every subdirectory, each of which is a snapshot that you want to import as a commit. You'll change into each subdirectory and print the commands necessary to export it. Your basic main loop looks like this:

```
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
```

You run `print_export` inside each directory, which takes the manifest and mark of the previous snapshot and returns the manifest and mark of this one; that way, you can link them properly. "Mark" is the fast-import term for an identifier you give to a commit; as you create commits, you give each one a mark that you can use to link to it from other commits. So, the first thing to do in your `print_export` method is generate a mark from the directory name:

```
mark = convert_dir_to_mark(dir)
```

You'll do this by creating an array of directories and using the index value as the mark, because a mark must be an integer. Your method looks like this:

```
$marks = []
def convert_dir_to_mark(dir)
  if !$marks.include?(dir)
    $marks << dir
  end
  ($marks.index(dir) + 1).to_s
end
```

Now that you have an integer representation of your commit, you need a date for the commit metadata. Because the date is expressed in the name of the directory, you'll parse it out. The next line in your `print_export` file is

```
date = convert_dir_to_date(dir)
```

where `convert_dir_to_date` is defined as

```
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
```

That returns an integer value for the date of each directory. The last piece of meta-information you need for each commit is the committer data, which you hardcode in a global variable:

```
$author = 'Scott Chacon <schacon@example.com>'
```

Now you're ready to begin printing out the commit data for your importer. The initial information states that you're defining a commit object and what branch it's on, followed by the mark you've generated, the committer information and commit message, and then the previous commit, if any. The code looks like this:

```
# 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
```

You hardcode the time zone (-0700) because doing so is easy. If you're importing from another system, you must specify the time zone as an offset. The commit message must be expressed in a special format:

```
data (size)%n(contents)
```

The format consists of the word `data`, the size of the data to be read, a newline, and finally the data. Because you need to use the same format to specify the file contents later, you create a helper method, `export_data`:

```
def export_data(string)
  print "data #{string.size}%n#{string}"
end
```

All that's left is to specify the file contents for each snapshot. This is easy, because you have each one in a directory — you can print out the `deleteall` command followed by the contents of each file in the directory. Git will then record each snapshot appropriately:

```
puts 'deleteall'
Dir.glob("**/*").each do |file|
  next if !File.file?(file)
  inline_data(file)
end
```

Note: Because many systems think of their revisions as changes from one commit to another, `fast-import` can also take commands with each commit to specify which files have been added, removed, or modified and what the new contents are. You could calculate the differences between snapshots and provide only this data, but doing so is more complex — you may as well give Git all the data and let it figure it out. If this is better suited to your data, check the `fast-import` man page for details about how to provide your data in this manner.

The format for listing the new file contents or specifying a modified file with the new contents is as follows:

```
M 644 inline path/to/file
data (size)
(file contents)
```

Here, 644 is the mode (if you have executable files, you need to detect and specify 755 instead), and `inline` says you'll list the contents immediately after this line. Your `inline_data` method looks like this:

```
def inline_data(file, code = 'M', mode = '644')
  content = File.read(file)
  puts "#{code} #{mode} inline #{file}"
  export_data(content)
end
```

You reuse the `export_data` method you defined earlier, because it's the same as the way you specified your commit message data.

The last thing you need to do is to return the current mark so it can be passed to the next iteration:

```
return mark
```

NOTE: If you are running on Windows you'll need to make sure that you add one extra step. As mentioned before, Windows uses CRLF for new line characters while `git fast-import` expects only LF. To get around this problem and make `git fast-import` happy, you need to tell ruby to use LF instead of CRLF:

```
$stdout.binmode
```

That's it. If you run this script, you'll get content that looks something like this:

```
$ ruby import.rb /opt/import_from
```

```

commit refs/heads/master
mark :1
committer Scott Chacon <schacon@geemail.com> 1230883200 -0700
data 29
imported from back_2009_01_02deleteall
M 644 inline file.rb
data 12
version two
commit refs/heads/master
mark :2
committer Scott Chacon <schacon@geemail.com> 1231056000 -0700
data 29
imported from back_2009_01_04from :1
deleteall
M 644 inline file.rb
data 14
version three
M 644 inline new.rb
data 16
new version one
(...)

```

To run the importer, pipe this output through `git fast-import` while in the Git directory you want to import into. You can create a new directory and then run `git init` in it for a starting point, and then run your script:

```

$ 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:        18 (      1 duplicates      )
  blobs :              7 (      1 duplicates      0 deltas)
  trees :              6 (      0 duplicates      1 deltas)
  commits:             5 (      0 duplicates      0 deltas)
  tags :               0 (      0 duplicates      0 deltas)
Total branches:       1 (      1 loads      )
  marks:             1024 (      5 unique      )
  atoms:               3
Memory total:         2255 KiB
  pools:             2098 KiB
  objects:            156 KiB
-----
pack_report: getpagesize() =      4096
pack_report: core.packedGitWindowSize = 33554432
pack_report: core.packedGitLimit = 268435456
pack_report: pack_used_ctr =          9
pack_report: pack_mmap_calls =         5
pack_report: pack_open_windows =        1 /        1
pack_report: pack_mapped =      1356 /      1356
-----

```

As you can see, when it completes successfully, it gives you a bunch of statistics about what it accomplished. In this case, you imported 18 objects total for 5 commits into 1 branch. Now, you can run `git log` to see your new history:

```

$ git log -2
commit 10bfe7d22ce15ee25b60a824c8982157ca593d41
Author: Scott Chacon <schacon@example.com>
Date:   Sun May 3 12:57:39 2009 -0700

    imported from current

commit 7e519590de754d079dd73b44d695a42c9d2df452
Author: Scott Chacon <schacon@example.com>
Date:   Tue Feb 3 01:00:00 2009 -0700

```



```
imported from back_2009_02_03
```

There you go — a nice, clean Git repository. It's important to note that nothing is checked out — you don't have any files in your working directory at first. To get them, you must reset your branch to where `master` is now:

```
$ ls
$ git reset --hard master
HEAD is now at 10bfe7d imported from current
$ ls
file.rb lib
```

You can do a lot more with the `fast-import` tool — handle different modes, binary data, multiple branches and merging, tags, progress indicators, and more. A number of examples of more complex scenarios are available in the `contrib/fast-import` directory of the Git source code; one of the better ones is the `git-p4` script I just covered.

Summary

You should feel comfortable using Git with Subversion or importing nearly any existing repository into a new Git one without losing data. The next chapter will cover the raw internals of Git so you can craft every single byte, if need be.

Gitの内側

以前の章を飛ばしてこの章に来るか、この本の他の部分を読んだ後にここに到達したかでしょう。どちらの場合であっても、この章は、Gitの内部動作と実装を辿るところになります。この情報を学習することは、Gitがどうして便利で効果的なのかを理解するのに根本的には重要ですが、他の人々は初心者には混乱を招き無駄に複雑だと主張してきました。このため、遅かれ早かれ学習の仕方にあわせて読めるように、この議論をこの本の最後の章におきました。いつ読むかは、読者の判断にお任せします。

今やあなたはこの章を読んでいるので、早速、この章の議論を始めましょう。まず、Gitは基本的に連想記憶ファイル・システム（content-addressable filesystem）であり、その上に書かれたVCSユーザー・インターフェイスを備えています。これが意味することを、もうちょっと学習していきましょう。

初期のGit（主として1.5以前）では、磨き上げられたVCSというより、むしろファイル・システムを（Gitの特徴として）強調したため、そのユーザー・インターフェイスは今よりも複雑なものでした。最近の数年間で、ユーザー・インターフェイスは、簡潔でそこら中のあらゆるシステムで簡単に扱えるまで改良されましたが、Gitに対する固定観念はたいいてい、複雑で学習するのが難しい、初期のGitのユーザー・インターフェイスのあたりから変わっていません。

連想記憶ファイル・システム層は、驚くほど素晴らしいので、この章の最初でそれをカバーすることにします。そして、転送メカニズムと、結局は取り扱う事になるリポジトリの保守作業について学習することにします。

Plumbing and Porcelain

This book covers how to use Git with 30 or so verbs such as checkout, branch, remote, and so on. But because Git was initially a toolkit for a VCS rather than a full user-friendly VCS, it has a bunch of verbs that do low-level work and were designed to be chained together UNIX style or called from scripts. These commands are generally referred to as "plumbing" commands, and the more user-friendly commands are called "porcelain" commands.

The book's first eight chapters deal almost exclusively with porcelain commands. But in this chapter, you'll be dealing mostly with the lower-level plumbing commands, because they give you access to the inner workings of Git and help demonstrate how and why Git does what it does. These commands aren't meant to be used manually on the command line, but rather to be used as building blocks for new tools and custom scripts.

When you run `git init` in a new or existing directory, Git creates the `.git` directory, which is where almost everything that Git stores and manipulates is located. If you want to back up or clone your repository, copying this single directory elsewhere gives you nearly everything you need. This entire chapter basically deals with the stuff in this directory. Here's what it looks like:

```
$ ls
HEAD
branches/
config
description
hooks/
index
info/
objects/
refs/
```

You may see some other files in there, but this is a fresh `git init` repository — it's what you see by default. The `branches` directory isn't used by newer Git versions, and the `description` file is only used by the GitWeb program, so don't worry about those. The `config` file contains your project-specific configuration options, and the `info` directory keeps a global exclude file for ignored patterns that you don't want to track in a `.gitignore` file. The `hooks` directory contains your client- or server-side hook scripts, which are discussed in detail in Chapter 6.

This leaves four important entries: the `HEAD` and `index` files and the `objects` and `refs` directories. These are the core parts of Git. The `objects` directory stores all the content for your database, the `refs` directory stores pointers into commit objects in that data (branches), the `HEAD` file points to the branch you currently have checked out, and the `index` file is where Git stores your staging area information. You'll now look at each of these sections in detail to see how Git operates.

Git Objects

Git is a content-addressable filesystem. Great. What does that mean? It means that at the core of Git is a simple key-value data store. You can insert any kind of content into it, and it will give you back a key that you can use to retrieve the content again at any time. To demonstrate, you can use the plumbing command `hash-object`, which takes some data, stores it in your `.git` directory, and gives you back the key the data is stored as. First, you initialize a new Git repository and verify that there is nothing in the `objects` directory:

```
$ mkdir test
```

```
$ cd test
$ git init
Initialized empty Git repository in /tmp/test/.git/
$ find .git/objects
.git/objects
.git/objects/info
.git/objects/pack
$ find .git/objects -type f
$
```

Git has initialized the objects directory and created pack and info subdirectories in it, but there are no regular files. Now, store some text in your Git database:

```
$ echo 'test content' | git hash-object -w --stdin
d670460b4b4aece5915caf5c68d12f560a9fe3e4
```

The `-w` tells `hash-object` to store the object; otherwise, the command simply tells you what the key would be. `--stdin` tells the command to read the content from `stdin`; if you don't specify this, `hash-object` expects the path to a file. The output from the command is a 40-character checksum hash. This is the SHA-1 hash — a checksum of the content you're storing plus a header, which you'll learn about in a bit. Now you can see how Git has stored your data:

```
$ find .git/objects -type f
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
```

You can see a file in the objects directory. This is how Git stores the content initially — as a single file per piece of content, named with the SHA-1 checksum of the content and its header. The subdirectory is named with the first 2 characters of the SHA, and the filename is the remaining 38 characters.

You can pull the content back out of Git with the `cat-file` command. This command is sort of a Swiss army knife for inspecting Git objects. Passing `-p` to it instructs the `cat-file` command to figure out the type of content and display it nicely for you:

```
$ git cat-file -p d670460b4b4aece5915caf5c68d12f560a9fe3e4
test content
```

Now, you can add content to Git and pull it back out again. You can also do this with content in files. For example, you can do some simple version control on a file. First, create a new file and save its contents in your database:

```
$ echo 'version 1' > test.txt
$ git hash-object -w test.txt
83baae61804e65cc73a7201a7252750c76066a30
```

Then, write some new content to the file, and save it again:

```
$ echo 'version 2' > test.txt
$ git hash-object -w test.txt
1f7a7a472abf3dd9643fd615f6da379c4acb3e3a
```

Your database contains the two new versions of the file as well as the first content you stored there:

```
$ find .git/objects -type f
.git/objects/1f/7a7a472abf3dd9643fd615f6da379c4acb3e3a
.git/objects/83/baae61804e65cc73a7201a7252750c76066a30
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
```

Now you can revert the file back to the first version

```
$ git cat-file -p 83baae61804e65cc73a7201a7252750c76066a30 > test.txt
$ cat test.txt
version 1
```

or the second version:

```
$ git cat-file -p 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a > test.txt
$ cat test.txt
version 2
```

But remembering the SHA-1 key for each version of your file isn't practical; plus, you aren't storing the filename in your system — just the content. This object type is called a blob. You can have Git tell you the object type of any object in Git, given its SHA-1 key, with `cat-file -t`:

```
$ git cat-file -t 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a
blob
```

Tree Objects

The next type you'll look at is the tree object, which solves the problem of storing the filename and also allows you to store a group of files together. Git stores content in a manner similar to a UNIX filesystem, but a bit simplified. All the content is stored as tree and blob objects, with trees corresponding to UNIX directory entries and blobs corresponding more or less to inodes or file contents. A single tree object contains one or more tree entries, each of which contains a SHA-1 pointer to a blob or subtree with its associated mode, type, and filename. For example, the most recent tree in the `simplegit` project may look something like this:

```
$ git cat-file -p master^{tree}
100644 blob a906cb2a4a904a152e80877d4088654daad0c859    README
100644 blob 8f94139338f9404f26296bfa88755fc2598c289    Rakefile
040000 tree 99f1a6d12cb4b6f19c8655fca46c3ecf317074e0    lib
```

The `master^{tree}` syntax specifies the tree object that is pointed to by the last commit on your `master` branch. Notice that the `lib` subdirectory isn't a blob but a pointer to another tree:

```
$ git cat-file -p 99f1a6d12cb4b6f19c8655fca46c3ecf317074e0
100644 blob 47c6340d6459e05787f644c2447d2595f5d3a54b    simplegit.rb
```

Conceptually, the data that Git is storing is something like Figure 9-1.

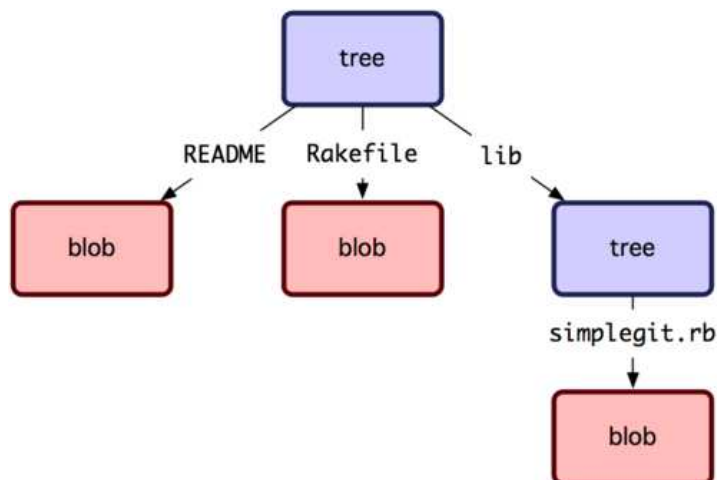


Figure 9-1. Simple version of the Git data model.

You can create your own tree. Git normally creates a tree by taking the state of your staging area or index and writing a tree object from it. So, to create a tree object, you first have to set up an index by staging some files. To create an index with a single entry — the first version of your `test.txt` file — you can use the plumbing command `update-index`. You use this command to artificially add the earlier version of the `test.txt` file to a new staging area. You must pass it the `--add` option because the file doesn't yet exist in your staging area (you don't even have a staging area set up yet) and `--cacheinfo` because the file you're adding isn't in your directory but is in your database. Then, you specify the mode, SHA-1, and filename:

```
$ git update-index --add --cacheinfo 100644 ¥
83baae61804e65cc73a7201a7252750c76066a30 test.txt
```

In this case, you're specifying a mode of `100644`, which means it's a normal file. Other options are `100755`, which means it's an executable file; and `120000`, which specifies a symbolic link. The mode is taken from normal UNIX modes but is much less flexible — these three modes are the only ones that are valid for files (blobs) in Git (although other modes are used for directories and submodules).

Now, you can use the `write-tree` command to write the staging area out to a tree object. No `-w` option is needed — calling `write-tree` automatically creates a tree object from the state of the index if that tree doesn't yet exist:

```
$ git write-tree
d8329fc1cc938780ffdd9f94e0d364e0ea74f579
$ git cat-file -p d8329fc1cc938780ffdd9f94e0d364e0ea74f579
100644 blob 83baae61804e65cc73a7201a7252750c76066a30    test.txt
```

You can also verify that this is a tree object:

```
$ git cat-file -t d8329fc1cc938780ffdd9f94e0d364e0ea74f579
tree
```

You'll now create a new tree with the second version of `test.txt` and a new file as well:

```
$ echo 'new file' > new.txt
$ git update-index test.txt
$ git update-index --add new.txt
```

Your staging area now has the new version of `test.txt` as well as the new file `new.txt`. Write out that tree (recording the state of the staging area or index to a tree object) and see what it looks like:

```
$ git write-tree
0155eb4229851634a0f03eb265b69f5a2d56f341
$ git cat-file -p 0155eb4229851634a0f03eb265b69f5a2d56f341
100644 blob fa49b077972391ad58037050f2a75f74e3671e92    new.txt
100644 blob 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a    test.txt
```

Notice that this tree has both file entries and also that the `test.txt` SHA is the "version 2" SHA from earlier (`1f7a7a`). Just for fun, you'll add the first tree as a subdirectory into this one. You can read trees into your staging area by calling `read-tree`. In this case, you can read an existing tree into your staging area as a subtree by using the `--prefix` option to `read-tree`:

```
$ 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
```

If you created a working directory from the new tree you just wrote, you would get the two files in the top level of the working directory and a subdirectory named `bak` that contained the first version of the `test.txt` file. You can think of the data that Git contains for these structures as being like Figure 9-2.

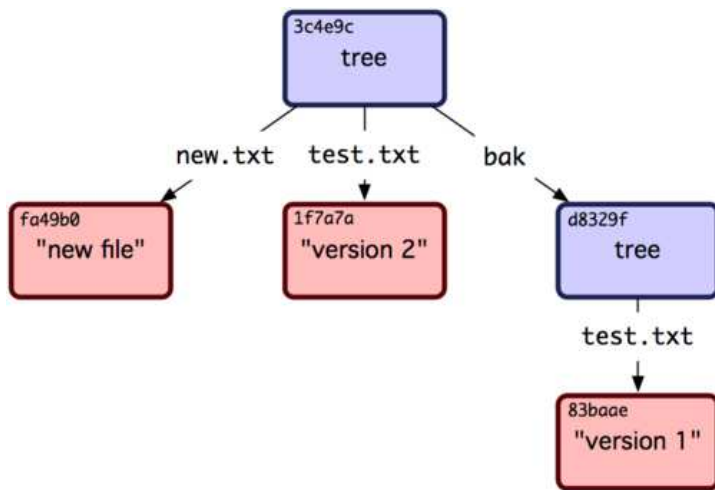


Figure 9-2. The content structure of your current Git data.

Commit Objects

You have three trees that specify the different snapshots of your project that you want to track, but the earlier problem remains: you must remember all three SHA-1 values in order to recall the snapshots. You also don't have any information about who saved the snapshots, when they were saved, or why they were saved. This is the basic information that the commit object stores for you.

To create a commit object, you call `commit-tree` and specify a single tree SHA-1 and which commit objects, if any, directly preceded it. Start with the first tree you wrote:

```
$ echo 'first commit' | git commit-tree d8329f
fdf4fc3344e67ab068f836878b6c4951e3b15f3d
```

Now you can look at your new commit object with `cat-file`:

```
$ 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
```

The format for a commit object is simple: it specifies the top-level tree for the snapshot of the project at that point; the author/committer information pulled from your `user.name` and `user.email` configuration settings, with the current timestamp; a blank line, and then the commit message.

Next, you'll write the other two commit objects, each referencing the commit that came directly before it:

```
$ echo 'second commit' | git commit-tree 0155eb -p fdf4fc3
cac0cab538b970a37ea1e769cbbde608743bc96d
$ echo 'third commit' | git commit-tree 3c4e9c -p cac0cab
1a410efbd13591db07496601ebc7a059dd55cfe9
```

Each of the three commit objects points to one of the three snapshot trees you created. Oddly enough, you have a real Git history now that you can view with the `git log` command, if you run it on the last commit SHA-1:

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

```
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 deletions(-)
```

```
commit fdf4fc3344e67ab068f836878b6c4951e3b15f3d
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri May 22 18:09:34 2009 -0700
```

first commit

```
test.txt |    1 +
1 files changed, 1 insertions(+), 0 deletions(-)
```

Amazing. You've just done the low-level operations to build up a Git history without using any of the front ends. This is essentially what Git does when you run the `git add` and `git commit` commands — it stores blobs for the files that have changed, updates the index, writes out trees, and writes commit objects that reference the top-level trees and the commits that came immediately before them. These three main Git objects — the blob, the tree, and the commit — are initially stored as separate files in your `.git/objects` directory. Here are all the objects in the example directory now, commented with what they store:

```
$ 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
```

If you follow all the internal pointers, you get an object graph something like Figure 9-3.

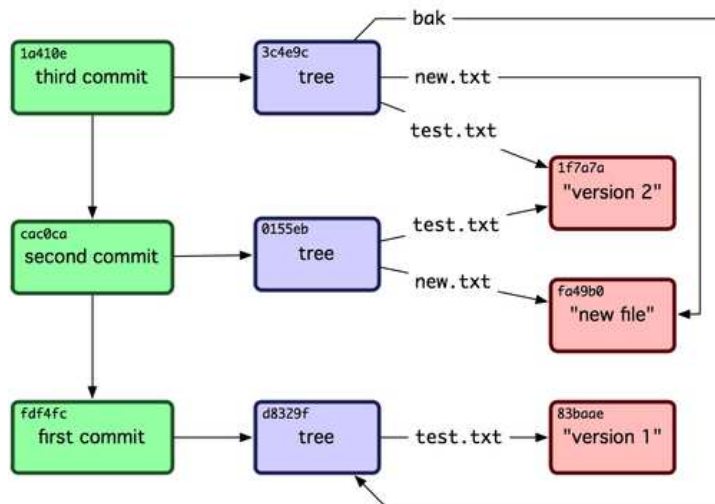


Figure 9-3. All the objects in your Git directory.

Object Storage

I mentioned earlier that a header is stored with the content. Let's take a minute to look at how Git stores its objects. You'll see how to store a blob object — in this case, the string "what is up, doc?" — interactively in the Ruby scripting language. You can start up interactive Ruby mode with the `irb` command:

```
$ irb
```

```
>> content = "what is up, doc?"
=> "what is up, doc?"
```

Git constructs a header that starts with the type of the object, in this case a blob. Then, it adds a space followed by the size of the content and finally a null byte:

```
>> header = "blob #{content.length}\0"
=> "blob 16\000"
```

Git concatenates the header and the original content and then calculates the SHA-1 checksum of that new content. You can calculate the SHA-1 value of a string in Ruby by including the SHA1 digest library with the `require` command and then calling `Digest::SHA1.hexdigest()` with the string:

```
>> store = header + content
=> "blob 16\000what is up, doc?"
>> require 'digest/sha1'
=> true
>> sha1 = Digest::SHA1.hexdigest(store)
=> "bd9dbf5aae1a3862dd1526723246b20206e5fc37"
```

Git compresses the new content with `zlib`, which you can do in Ruby with the `zlib` library. First, you need to require the library and then run `Zlib::Deflate.deflate()` on the content:

```
>> require 'zlib'
=> true
>> zlib_content = Zlib::Deflate.deflate(store)
=> "x\234K\312\3110R04c(\317H,Q\310,V(-\320QH\3110\266\000_\034\0235"
```

Finally, you'll write your `zlib`-deflated content to an object on disk. You'll determine the path of the object you want to write out (the first two characters of the SHA-1 value being the subdirectory name, and the last 38 characters being the filename within that directory). In Ruby, you can use the `FileUtils.mkdir_p()` function to create the subdirectory if it doesn't exist. Then, open the file with `File.open()` and write out the previously `zlib`-compressed content to the file with a `write()` call on the resulting file handle:

```
>> 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
```

That's it — you've created a valid Git blob object. All Git objects are stored the same way, just with different types — instead of the string blob, the header will begin with `commit` or `tree`. Also, although the blob content can be nearly anything, the `commit` and `tree` content are very specifically formatted.

Git References

You can run something like `git log 1a410e` to look through your whole history, but you still have to remember that `1a410e` is the last commit in order to walk that history to find all those objects. You need a file in which you can store the SHA-1 value under a simple name so you can use that pointer rather than the raw SHA-1 value.

In Git, these are called "references" or "refs"; you can find the files that contain the SHA-1 values in the `.git/refs` directory. In the current project, this directory contains no files, but it does contain a simple structure:

```
$ find .git/refs
.git/refs
.git/refs/heads
.git/refs/tags
$ find .git/refs -type f
$
```


To create a new reference that will help you remember where your latest commit is, you can technically do something as simple as this:

```
$ echo "1a410efbd13591db07496601ebc7a059dd55cfe9" > .git/refs/heads/master
```

Now, you can use the head reference you just created instead of the SHA-1 value in your Git commands:

```
$ git log --pretty=oneline master
1a410efbd13591db07496601ebc7a059dd55cfe9 third commit
cac0cab538b970a37ea1e769cbbde608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

You aren't encouraged to directly edit the reference files. Git provides a safer command to do this if you want to update a reference called `update-ref`:

```
$ git update-ref refs/heads/master 1a410efbd13591db07496601ebc7a059dd55cfe9
```

That's basically what a branch in Git is: a simple pointer or reference to the head of a line of work. To create a branch back at the second commit, you can do this:

```
$ git update-ref refs/heads/test cac0ca
```

Your branch will contain only work from that commit down:

```
$ git log --pretty=oneline test
cac0cab538b970a37ea1e769cbbde608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

Now, your Git database conceptually looks something like Figure 9-4.

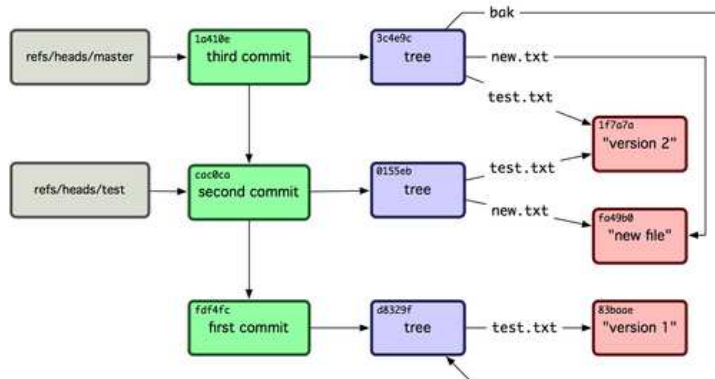


Figure 9-4. Git directory objects with branch head references included.

When you run commands like `git branch (branchname)`, Git basically runs that `update-ref` command to add the SHA-1 of the last commit of the branch you're on into whatever new reference you want to create.

The HEAD

The question now is, when you run `git branch (branchname)`, how does Git know the SHA-1 of the last commit? The answer is the HEAD file. The HEAD file is a symbolic reference to the branch you're currently on. By symbolic reference, I mean that unlike a normal reference, it doesn't generally contain a SHA-1 value but rather a pointer to another reference. If you look at the file, you'll normally see something like this:

```
$ cat .git/HEAD
ref: refs/heads/master
```

If you run `git checkout test`, Git updates the file to look like this:

```
$ cat .git/HEAD
```

```
ref: refs/heads/test
```

When you run `git commit`, it creates the commit object, specifying the parent of that commit object to be whatever SHA-1 value the reference in `HEAD` points to.

You can also manually edit this file, but again a safer command exists to do so: `symbolic-ref`. You can read the value of your `HEAD` via this command:

```
$ git symbolic-ref HEAD
refs/heads/master
```

You can also set the value of `HEAD`:

```
$ git symbolic-ref HEAD refs/heads/test
$ cat .git/HEAD
ref: refs/heads/test
```

You can't set a symbolic reference outside of the `refs` style:

```
$ git symbolic-ref HEAD test
fatal: Refusing to point HEAD outside of refs/
```

Tags

You've just gone over Git's three main object types, but there is a fourth. The tag object is very much like a commit object — it contains a tagger, a date, a message, and a pointer. The main difference is that a tag object points to a commit rather than a tree. It's like a branch reference, but it never moves — it always points to the same commit but gives it a friendlier name.

As discussed in Chapter 2, there are two types of tags: annotated and lightweight. You can make a lightweight tag by running something like this:

```
$ git update-ref refs/tags/v1.0 cac0cab538b970a37ea1e769cbbde608743bc96d
```

That is all a lightweight tag is — a branch that never moves. An annotated tag is more complex, however. If you create an annotated tag, Git creates a tag object and then writes a reference to point to it rather than directly to the commit. You can see this by creating an annotated tag (`-a` specifies that it's an annotated tag):

```
$ git tag -a v1.1 1a410efbd13591db07496601ebc7a059dd55cfe9 -m 'test tag'
```

Here's the object SHA-1 value it created:

```
$ cat .git/refs/tags/v1.1
9585191f37f7b0fb9444f35a9bf50de191beadc2
```

Now, run the `cat-file` command on that SHA-1 value:

```
$ 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
```

Notice that the object entry points to the commit SHA-1 value that you tagged. Also notice that it doesn't need to point to a commit; you can tag any Git object. In the Git source code, for example, the maintainer has added their GPG public key as a blob object and then tagged it. You can view the public key by running

```
$ git cat-file blob junio-gpg-pub
```

in the Git source code repository. The Linux kernel repository also has a non-commit-pointing tag object — the first tag created points to the initial tree of the import of the source code.

Remotes

The third type of reference that you'll see is a remote reference. If you add a remote and push to it, Git stores the value you last pushed to that remote for each branch in the `refs/remotes` directory. For instance, you can add a remote called `origin` and push your `master` branch to it:

```
$ 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
```

Then, you can see what the `master` branch on the `origin` remote was the last time you communicated with the server, by checking the `refs/remotes/origin/master` file:

```
$ cat .git/refs/remotes/origin/master
ca82a6dff817ec66f44342007202690a93763949
```

Remote references differ from branches (`refs/heads` references) mainly in that they can't be checked out. Git moves them around as bookmarks to the last known state of where those branches were on those servers.

Packfiles

Let's go back to the objects database for your test Git repository. At this point, you have 11 objects — 4 blobs, 3 trees, 3 commits, and 1 tag:

```
$ 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 compresses the contents of these files with `zlib`, and you're not storing much, so all these files collectively take up only 925 bytes. You'll add some larger content to the repository to demonstrate an interesting feature of Git. Add the `repo.rb` file from the Grit library you worked with earlier — this is about a 12K source code file:

```
$ curl http://github.com/mojombo/grit/raw/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, 459 insertions(+), 2 deletions(-)
delete mode 100644 bak/test.txt
create mode 100644 repo.rb
rewrite test.txt (100%)
```

If you look at the resulting tree, you can see the SHA-1 value your `repo.rb` file got for the blob object:

```
$ git cat-file -p master^{tree}
100644 blob fa49b077972391ad58037050f2a75f74e3671e92    new.txt
100644 blob 9bc1dc421dcd51b4ac296e3e5b6e2a99cf44391e    repo.rb
```

```
100644 blob e3f094f522629ae358806b17daf78246c27c007b    test.txt
```

You can then use `git cat-file` to see how big that object is:

```
$ git cat-file -s 9bc1dc421dcd51b4ac296e3e5b6e2a99cf44391e
12898
```

Now, modify that file a little, and see what happens:

```
$ echo '# testing' >> repo.rb
$ git commit -am 'modified repo a bit'
[master ab1afef] modified repo a bit
1 files changed, 1 insertions(+), 0 deletions(-)
```

Check the tree created by that commit, and you see something interesting:

```
$ git cat-file -p master^{tree}
100644 blob fa49b077972391ad58037050f2a75f74e3671e92    new.txt
100644 blob 05408d195263d853f09dca71d55116663690c27c    repo.rb
100644 blob e3f094f522629ae358806b17daf78246c27c007b    test.txt
```

The blob is now a different blob, which means that although you added only a single line to the end of a 400-line file, Git stored that new content as a completely new object:

```
$ git cat-file -s 05408d195263d853f09dca71d55116663690c27c
12908
```

You have two nearly identical 12K objects on your disk. Wouldn't it be nice if Git could store one of them in full but then the second object only as the delta between it and the first?

It turns out that it can. The initial format in which Git saves objects on disk is called a loose object format. However, occasionally Git packs up several of these objects into a single binary file called a packfile in order to save space and be more efficient. Git does this if you have too many loose objects around, if you run the `git gc` command manually, or if you push to a remote server. To see what happens, you can manually ask Git to pack up the objects by calling the `git gc` command:

```
$ git gc
Counting objects: 17, done.
Delta compression using 2 threads.
Compressing objects: 100% (13/13), done.
Writing objects: 100% (17/17), done.
Total 17 (delta 1), reused 10 (delta 0)
```

If you look in your objects directory, you'll find that most of your objects are gone, and a new pair of files has appeared:

```
$ find .git/objects -type f
.git/objects/71/08f7ecb345ee9d0084193f147cdad4d2998293
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
.git/objects/info/packs
.git/objects/pack/pack-7a16e4488ae40c7d2bc56ea2bd43e25212a66c45.idx
.git/objects/pack/pack-7a16e4488ae40c7d2bc56ea2bd43e25212a66c45.pack
```

The objects that remain are the blobs that aren't pointed to by any commit — in this case, the "what is up, doc?" example and the "test content" example blobs you created earlier. Because you never added them to any commits, they're considered dangling and aren't packed up in your new packfile.

The other files are your new packfile and an index. The packfile is a single file containing the contents of all the objects that were removed from your filesystem. The index is a file that contains offsets into that packfile so you can quickly seek to a specific object. What is cool is that although the objects on disk before you ran the `gc` were collectively about 12K in size, the new packfile is only 6K. You've halved your disk usage by packing your objects.

How does Git do this? When Git packs objects, it looks for files that are named and sized similarly, and stores just the

deltas from one version of the file to the next. You can look into the packfile and see what Git did to save space. The `git verify-pack` plumbing command allows you to see what was packed up:

```
$ git verify-pack -v ¥
.git/objects/pack/pack-7a16e4488ae40c7d2bc56ea2bd43e25212a66c45.idx
0155eb4229851634a0f03eb265b69f5a2d56f341 tree    71 76 5400
05408d195263d853f09dca71d55116663690c27c blob   12908 3478 874
09f01cea547666f58d6a8d809583841a7c6f0130 tree    106 107 5086
1a410efbd13591db07496601ebc7a059dd55cfe9 commit 225 151 322
1f7a7a472abf3dd9643fd615f6da379c4acb3e3a blob    10 19 5381
3c4e9cd789d88d8d89c1073707c3585e41b0e614 tree    101 105 5211
484a59275031909e19aad7c92262719cfcdf19a commit 226 153 169
83baae61804e65cc73a7201a7252750c76066a30 blob    10 19 5362
9585191f37f7b0fb9444f35a9bf50de191beadc2 tag     136 127 5476
9bc1dc421dcd51b4ac296e3e5b6e2a99cf44391e blob     7 18 5193 1
05408d195263d853f09dca71d55116663690c27c ¥
  ab1afef80fac8e34258ff41fc1b867c702daa24b commit 232 157 12
cac0cab538b970a37ea1e769cbbde608743bc96d commit 226 154 473
d8329fc1cc938780ffdd9f94e0d364e0ea74f579 tree     36 46 5316
e3f094f522629ae358806b17daf78246c27c007b blob   1486 734 4352
f8f51d7d8a1760462eca26eebafde32087499533 tree    106 107 749
fa49b077972391ad58037050f2a75f74e3671e92 blob     9 18 856
fdf4fc3344e67ab068f836878b6c4951e3b15f3d commit 177 122 627
chain length = 1: 1 object
pack-7a16e4488ae40c7d2bc56ea2bd43e25212a66c45.pack: ok
```

Here, the `9bc1d` blob, which if you remember was the first version of your `repo.rb` file, is referencing the `05408` blob, which was the second version of the file. The third column in the output is the size of the object in the pack, so you can see that `05408` takes up 12K of the file but that `9bc1d` only takes up 7 bytes. What is also interesting is that the second version of the file is the one that is stored intact, whereas the original version is stored as a delta — this is because you're most likely to need faster access to the most recent version of the file.

The really nice thing about this is that it can be repacked at any time. Git will occasionally repack your database automatically, always trying to save more space. You can also manually repack at any time by running `git gc` by hand.

The Refspec

Throughout this book, you've used simple mappings from remote branches to local references; but they can be more complex. Suppose you add a remote like this:

```
$ git remote add origin git@github.com:schacon/simplegit-progit.git
```

It adds a section to your `.git/config` file, specifying the name of the remote (`origin`), the URL of the remote repository, and the refspec for fetching:

```
[remote "origin"]
  url = git@github.com:schacon/simplegit-progit.git
  fetch = +refs/heads/*:refs/remotes/origin/*
```

The format of the refspec is an optional `+`, followed by `<src>:<dst>`, where `<src>` is the pattern for references on the remote side and `<dst>` is where those references will be written locally. The `+` tells Git to update the reference even if it isn't a fast-forward.

In the default case that is automatically written by a `git remote add` command, Git fetches all the references under `refs/heads/` on the server and writes them to `refs/remotes/origin/` locally. So, if there is a `master` branch on the server, you can access the log of that branch locally via

```
$ git log origin/master
$ git log remotes/origin/master
$ git log refs/remotes/origin/master
```

They're all equivalent, because Git expands each of them to `refs/remotes/origin/master`.

If you want Git instead to pull down only the `master` branch each time, and not every other branch on the remote server, you can change the fetch line to

```
fetch = +refs/heads/master:refs/remotes/origin/master
```

This is just the default refspec for `git fetch` for that remote. If you want to do something one time, you can specify the refspec on the command line, too. To pull the master branch on the remote down to `origin/mymaster` locally, you can run

```
$ git fetch origin master:refs/remotes/origin/mymaster
```

You can also specify multiple refspecs. On the command line, you can pull down several branches like so:

```
$ 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
```

In this case, the master branch pull was rejected because it wasn't a fast-forward reference. You can override that by specifying the `+` in front of the refspec.

You can also specify multiple refspecs for fetching in your configuration file. If you want to always fetch the master and experiment branches, add two lines:

```
[remote "origin"]
  url = git@github.com:schacon/simplegit-progit.git
  fetch = +refs/heads/master:refs/remotes/origin/master
  fetch = +refs/heads/experiment:refs/remotes/origin/experiment
```

You can't use partial globs in the pattern, so this would be invalid:

```
fetch = +refs/heads/qa*:refs/remotes/origin/qa*
```

However, you can use namespacing to accomplish something like that. If you have a QA team that pushes a series of branches, and you want to get the master branch and any of the QA team's branches but nothing else, you can use a config section like this:

```
[remote "origin"]
  url = git@github.com:schacon/simplegit-progit.git
  fetch = +refs/heads/master:refs/remotes/origin/master
  fetch = +refs/heads/qa/*:refs/remotes/origin/qa/*
```

If you have a complex workflow process that has a QA team pushing branches, developers pushing branches, and integration teams pushing and collaborating on remote branches, you can namespace them easily this way.

Pushing Refspecs

It's nice that you can fetch namespaced references that way, but how does the QA team get their branches into a `qa/` namespace in the first place? You accomplish that by using refspecs to push.

If the QA team wants to push their master branch to `qa/master` on the remote server, they can run

```
$ git push origin master:refs/heads/qa/master
```

If they want Git to do that automatically each time they run `git push origin`, they can add a push value to their config file:

```
[remote "origin"]
  url = git@github.com:schacon/simplegit-progit.git
  fetch = +refs/heads/*:refs/remotes/origin/*
  push = refs/heads/master:refs/heads/qa/master
```

Again, this will cause a `git push origin` to push the local master branch to the remote `qa/master` branch by default.

Deleting References

You can also use the refspec to delete references from the remote server by running something like this:

```
$ git push origin :topic
```

Because the refspec is <src>:<dst>, by leaving off the <src> part, this basically says to make the topic branch on the remote nothing, which deletes it.

Transfer Protocols

Git can transfer data between two repositories in two major ways: over HTTP and via the so-called smart protocols used in the file://, ssh://, and git:// transports. This section will quickly cover how these two main protocols operate.

The Dumb Protocol

Git transport over HTTP is often referred to as the dumb protocol because it requires no Git-specific code on the server side during the transport process. The fetch process is a series of GET requests, where the client can assume the layout of the Git repository on the server. Let's follow the http-fetch process for the simplegit library:

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

The first thing this command does is pull down the info/refs file. This file is written by the update-server-info command, which is why you need to enable that as a post-receive hook in order for the HTTP transport to work properly:

```
=> GET info/refs
ca82a6dff817ec66f44342007202690a93763949    refs/heads/master
```

Now you have a list of the remote references and SHAs. Next, you look for what the HEAD reference is so you know what to check out when you're finished:

```
=> GET HEAD
ref: refs/heads/master
```

You need to check out the master branch when you've completed the process. At this point, you're ready to start the walking process. Because your starting point is the ca82a6 commit object you saw in the info/refs file, you start by fetching that:

```
=> GET objects/ca/82a6dff817ec66f44342007202690a93763949
(179 bytes of binary data)
```

You get an object back — that object is in loose format on the server, and you fetched it over a static HTTP GET request. You can zlib-uncompress it, strip off the header, and look at the commit content:

```
$ git cat-file -p ca82a6dff817ec66f44342007202690a93763949
tree cfd3bf379e4f8dba8717dee55aab78aef7f4daf
parent 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
author Scott Chacon <schacon@gmail.com> 1205815931 -0700
committer Scott Chacon <schacon@gmail.com> 1240030591 -0700
```

```
changed the version number
```

Next, you have two more objects to retrieve — cfd3b, which is the tree of content that the commit we just retrieved points to; and 085bb3, which is the parent commit:

```
=> GET objects/08/5bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
(179 bytes of data)
```

That gives you your next commit object. Grab the tree object:

```
=> GET objects/cf/da3bf379e4f8dba8717dee55aab78aef7f4daf
(404 - Not Found)
```

Oops — it looks like that tree object isn't in loose format on the server, so you get a 404 response back. There are a couple of reasons for this — the object could be in an alternate repository, or it could be in a packfile in this repository. Git checks for any listed alternates first:

```
=> GET objects/info/http-alternates
(empty file)
```

If this comes back with a list of alternate URLs, Git checks for loose files and packfiles there — this is a nice mechanism for projects that are forks of one another to share objects on disk. However, because no alternates are listed in this case, your object must be in a packfile. To see what packfiles are available on this server, you need to get the `objects/info/packs` file, which contains a listing of them (also generated by `update-server-info`):

```
=> GET objects/info/packs
P pack-816a9b2334da9953e530f27bcac22082a9f5b835.pack
```

There is only one packfile on the server, so your object is obviously in there, but you'll check the index file to make sure. This is also useful if you have multiple packfiles on the server, so you can see which packfile contains the object you need:

```
=> GET objects/pack/pack-816a9b2334da9953e530f27bcac22082a9f5b835.idx
(4k of binary data)
```

Now that you have the packfile index, you can see if your object is in it — because the index lists the SHAs of the objects contained in the packfile and the offsets to those objects. Your object is there, so go ahead and get the whole packfile:

```
=> GET objects/pack/pack-816a9b2334da9953e530f27bcac22082a9f5b835.pack
(13k of binary data)
```

You have your tree object, so you continue walking your commits. They're all also within the packfile you just downloaded, so you don't have to do any more requests to your server. Git checks out a working copy of the master branch that was pointed to by the HEAD reference you downloaded at the beginning.

The entire output of this process looks like this:

```
$ git clone http://github.com/schacon/simplegit-progit.git
Initialized empty Git repository in /private/tmp/simplegit-progit/.git/
got ca82a6dff817ec66f44342007202690a93763949
walk ca82a6dff817ec66f44342007202690a93763949
got 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Getting alternates list for http://github.com/schacon/simplegit-progit.git
Getting pack list for http://github.com/schacon/simplegit-progit.git
Getting index for pack 816a9b2334da9953e530f27bcac22082a9f5b835
Getting pack 816a9b2334da9953e530f27bcac22082a9f5b835
  which contains cfda3bf379e4f8dba8717dee55aab78aef7f4daf
walk 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
walk a11bef06a3f659402fe7563abf99ad00de2209e6
```

The Smart Protocol

The HTTP method is simple but a bit inefficient. Using smart protocols is a more common method of transferring data. These protocols have a process on the remote end that is intelligent about Git — it can read local data and figure out what the client has or needs and generate custom data for it. There are two sets of processes for transferring data: a pair for uploading data and a pair for downloading data.

Uploading Data

To upload data to a remote process, Git uses the `send-pack` and `receive-pack` processes. The `send-pack` process runs on the client and connects to a `receive-pack` process on the remote side.

For example, say you run `git push origin master` in your project, and `origin` is defined as a URL that uses the SSH protocol.

Git fires up the send-pack process, which initiates a connection over SSH to your server. It tries to run a command on the remote server via an SSH call that looks something like this:

```
$ ssh -x git@github.com "git-receive-pack 'schacon/simplegit-progit.git'"
005bca82a6dff817ec66f4437202690a93763949 refs/heads/master report-status delete-refs
003e085bb3bcb608e1e84b2432f8ecbe6306e7e7 refs/heads/topic
0000
```

The git-receive-pack command immediately responds with one line for each reference it currently has — in this case, just the master branch and its SHA. The first line also has a list of the server's capabilities (here, report-status and delete-refs).

Each line starts with a 4-byte hex value specifying how long the rest of the line is. Your first line starts with 005b, which is 91 in hex, meaning that 91 bytes remain on that line. The next line starts with 003e, which is 62, so you read the remaining 62 bytes. The next line is 0000, meaning the server is done with its references listing.

Now that it knows the server's state, your send-pack process determines what commits it has that the server doesn't. For each reference that this push will update, the send-pack process tells the receive-pack process that information. For instance, if you're updating the master branch and adding an experiment branch, the send-pack response may look something like this:

```
0085ca82a6dff817ec66f44342007202690a93763949 15027957951b64cf874c3557a0f3547bd83b3ff6 refs/heads/master report-status
0067000000000000000000000000000000000000 cdfdb42577e2506715f8cfeacdbabc092bf63e8d refs/heads/experiment
0000
```

The SHA-1 value of all '0's means that nothing was there before — because you're adding the experiment reference. If you were deleting a reference, you would see the opposite: all '0's on the right side.

Git sends a line for each reference you're updating with the old SHA, the new SHA, and the reference that is being updated. The first line also has the client's capabilities. Next, the client uploads a packfile of all the objects the server doesn't have yet. Finally, the server responds with a success (or failure) indication:

```
000Aunpack ok
```

Downloading Data

When you download data, the fetch-pack and upload-pack processes are involved. The client initiates a fetch-pack process that connects to an upload-pack process on the remote side to negotiate what data will be transferred down.

There are different ways to initiate the upload-pack process on the remote repository. You can run via SSH in the same manner as the receive-pack process. You can also initiate the process via the Git daemon, which listens on a server on port 9418 by default. The fetch-pack process sends data that looks like this to the daemon after connecting:

```
003fgit-upload-pack schacon/simplegit-progit.git%0host=myserver.com%0
```

It starts with the 4 bytes specifying how much data is following, then the command to run followed by a null byte, and then the server's hostname followed by a final null byte. The Git daemon checks that the command can be run and that the repository exists and has public permissions. If everything is cool, it fires up the upload-pack process and hands off the request to it.

If you're doing the fetch over SSH, fetch-pack instead runs something like this:

```
$ ssh -x git@github.com "git-upload-pack 'schacon/simplegit-progit.git'"
```

In either case, after fetch-pack connects, upload-pack sends back something like this:

```
0088ca82a6dff817ec66f44342007202690a93763949 HEAD%0multi_ack thin-pack %
side-band side-band-64k ofs-delta shallow no-progress include-tag
003fca82a6dff817ec66f44342007202690a93763949 refs/heads/master
003e085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7 refs/heads/topic
0000
```

This is very similar to what receive-pack responds with, but the capabilities are different. In addition, it sends back the

HEAD reference so the client knows what to check out if this is a clone.

At this point, the fetch-pack process looks at what objects it has and responds with the objects that it needs by sending "want" and then the SHA it wants. It sends all the objects it already has with "have" and then the SHA. At the end of this list, it writes "done" to initiate the upload-pack process to begin sending the packfile of the data it needs:

```
0054want ca82a6dff817ec66f44342007202690a93763949 ofs-delta
0032have 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
0000
0009done
```

That is a very basic case of the transfer protocols. In more complex cases, the client supports `multi_ack` or `side-band` capabilities; but this example shows you the basic back and forth used by the smart protocol processes.

Maintenance and Data Recovery

Occasionally, you may have to do some cleanup — make a repository more compact, clean up an imported repository, or recover lost work. This section will cover some of these scenarios.

Maintenance

Occasionally, Git automatically runs a command called "auto gc". Most of the time, this command does nothing. However, if there are too many loose objects (objects not in a packfile) or too many packfiles, Git launches a full-fledged `git gc` command. The `gc` stands for garbage collect, and the command does a number of things: it gathers up all the loose objects and places them in packfiles, it consolidates packfiles into one big packfile, and it removes objects that aren't reachable from any commit and are a few months old.

You can run `auto gc` manually as follows:

```
$ git gc --auto
```

Again, this generally does nothing. You must have around 7,000 loose objects or more than 50 packfiles for Git to fire up a real `gc` command. You can modify these limits with the `gc.auto` and `gc.autopacklimit` config settings, respectively.

The other thing `gc` will do is pack up your references into a single file. Suppose your repository contains the following branches and tags:

```
$ find .git/refs -type f
.git/refs/heads/experiment
.git/refs/heads/master
.git/refs/tags/v1.0
.git/refs/tags/v1.1
```

If you run `git gc`, you'll no longer have these files in the `refs` directory. Git will move them for the sake of efficiency into a file named `.git/packed-refs` that looks like this:

```
$ cat .git/packed-refs
# pack-refs with: peeled
cac0cab538b970a37ea1e769cbbde608743bc96d refs/heads/experiment
ab1afef80fac8e34258ff41fc1b867c702daa24b refs/heads/master
cac0cab538b970a37ea1e769cbbde608743bc96d refs/tags/v1.0
9585191f37f7b0fb9444f35a9bf50de191beadc2 refs/tags/v1.1
^1a410efbd13591db07496601ebc7a059dd55cfe9
```

If you update a reference, Git doesn't edit this file but instead writes a new file to `refs/heads`. To get the appropriate SHA for a given reference, Git checks for that reference in the `refs` directory and then checks the `packed-refs` file as a fallback. However, if you can't find a reference in the `refs` directory, it's probably in your `packed-refs` file.

Notice the last line of the file, which begins with a `^`. This means the tag directly above is an annotated tag and that line is the commit that the annotated tag points to.

Data Recovery

At some point in your Git journey, you may accidentally lose a commit. Generally, this happens because you force-delete a branch that had work on it, and it turns out you wanted the branch after all; or you hard-reset a branch, thus abandoning

commits that you wanted something from. Assuming this happens, how can you get your commits back?

Here's an example that hard-resets the master branch in your test repository to an older commit and then recovers the lost commits. First, let's review where your repository is at this point:

```
$ git log --pretty=oneline
ab1afef80fac8e34258ff41fc1b867c702daa24b modified repo a bit
484a59275031909e19aadb7c92262719cfcdf19a added repo.rb
1a410efbd13591db07496601ebc7a059dd55cfe9 third commit
cac0cab538b970a37ea1e769cbbde608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

Now, move the master branch back to the middle commit:

```
$ git reset --hard 1a410efbd13591db07496601ebc7a059dd55cfe9
HEAD is now at 1a410ef third commit
$ git log --pretty=oneline
1a410efbd13591db07496601ebc7a059dd55cfe9 third commit
cac0cab538b970a37ea1e769cbbde608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

You've effectively lost the top two commits — you have no branch from which those commits are reachable. You need to find the latest commit SHA and then add a branch that points to it. The trick is finding that latest commit SHA — it's not like you've memorized it, right?

Often, the quickest way is to use a tool called `git reflog`. As you're working, Git silently records what your HEAD is every time you change it. Each time you commit or change branches, the reflog is updated. The reflog is also updated by the `git update-ref` command, which is another reason to use it instead of just writing the SHA value to your ref files, as we covered in the "Git References" section of this chapter earlier. You can see where you've been at any time by running `git reflog`:

```
$ git reflog
1a410ef HEAD@{0}: 1a410efbd13591db07496601ebc7a059dd55cfe9: updating HEAD
ab1afef HEAD@{1}: ab1afef80fac8e34258ff41fc1b867c702daa24b: updating HEAD
```

Here we can see the two commits that we have had checked out, however there is not much information here. To see the same information in a much more useful way, we can run `git log -g`, which will give you a normal log output for your reflog.

```
$ 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 a bit

It looks like the bottom commit is the one you lost, so you can recover it by creating a new branch at that commit. For example, you can start a branch named `recover-branch` at that commit (`ab1afef`):

```
$ 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
```

Cool — now you have a branch named `recover-branch` that is where your master branch used to be, making the first two commits reachable again. Next, suppose your loss was for some reason not in the reflog — you can simulate that by removing `recover-branch` and deleting the reflog. Now the first two commits aren't reachable by anything:

```
$ git branch -D recover-branch
$ rm -Rf .git/logs/
```

Because the reflog data is kept in the `.git/logs/` directory, you effectively have no reflog. How can you recover that commit at this point? One way is to use the `git fsck` utility, which checks your database for integrity. If you run it with the `--full` option, it shows you all objects that aren't pointed to by another object:

```
$ git fsck --full
dangling blob d670460b4b4aece5915caf5c68d12f560a9fe3e4
dangling commit ab1afef80fac8e34258ff41fc1b867c702daa24b
dangling tree aea790b9a58f6cf6f2804eeac9f0abbe9631e4c9
dangling blob 7108f7ecb345ee9d0084193f147cdad4d2998293
```

In this case, you can see your missing commit after the dangling commit. You can recover it the same way, by adding a branch that points to that SHA.

Removing Objects

There are a lot of great things about Git, but one feature that can cause issues is the fact that a `git clone` downloads the entire history of the project, including every version of every file. This is fine if the whole thing is source code, because Git is highly optimized to compress that data efficiently. However, if someone at any point in the history of your project added a single huge file, every clone for all time will be forced to download that large file, even if it was removed from the project in the very next commit. Because it's reachable from the history, it will always be there.

This can be a huge problem when you're converting Subversion or Perforce repositories into Git. Because you don't download the whole history in those systems, this type of addition carries few consequences. If you did an import from another system or otherwise find that your repository is much larger than it should be, here is how you can find and remove large objects.

Be warned: this technique is destructive to your commit history. It rewrites every commit object downstream from the earliest tree you have to modify to remove a large file reference. If you do this immediately after an import, before anyone has started to base work on the commit, you're fine — otherwise, you have to notify all contributors that they must rebase their work onto your new commits.

To demonstrate, you'll add a large file into your test repository, remove it in the next commit, find it, and remove it permanently from the repository. First, add a large object to your history:

```
$ curl http://kernel.org/pub/software/scm/git/git-1.6.3.1.tar.bz2 > git.tbz2
$ git add git.tbz2
$ git commit -am 'added git tarball'
[master 6df7640] added git tarball
 1 files changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 git.tbz2
```

Oops — you didn't want to add a huge tarball to your project. Better get rid of it:

```
$ git rm git.tbz2
rm 'git.tbz2'
$ git commit -m 'oops - removed large tarball'
[master da3f30d] oops - removed large tarball
 1 files changed, 0 insertions(+), 0 deletions(-)
 delete mode 100644 git.tbz2
```

Now, gc your database and see how much space you're using:

```
$ git gc
Counting objects: 21, done.
Delta compression using 2 threads.
Compressing objects: 100% (16/16), done.
Writing objects: 100% (21/21), done.
```

Total 21 (delta 3), reused 15 (delta 1)

You can run the `count-objects` command to quickly see how much space you're using:

```
$ git count-objects -v
count: 4
size: 16
in-pack: 21
packs: 1
size-pack: 2016
prune-packable: 0
garbage: 0
```

The `size-pack` entry is the size of your packfiles in kilobytes, so you're using 2MB. Before the last commit, you were using closer to 2K — clearly, removing the file from the previous commit didn't remove it from your history. Every time anyone clones this repository, they will have to clone all 2MB just to get this tiny project, because you accidentally added a big file. Let's get rid of it.

First you have to find it. In this case, you already know what file it is. But suppose you didn't; how would you identify what file or files were taking up so much space? If you run `git gc`, all the objects are in a packfile; you can identify the big objects by running another plumbing command called `git verify-pack` and sorting on the third field in the output, which is file size. You can also pipe it through the `tail` command because you're only interested in the last few largest files:

```
$ git verify-pack -v .git/objects/pack/pack-3f8c0...bb.idx | sort -k 3 -n | tail -3
e3f094f522629ae358806b17daf78246c27c007b blob 1486 734 4667
05408d195263d853f09dca71d55116663690c27c blob 12908 3478 1189
7a9eb2fba2b1811321254ac360970fc169ba2330 blob 2056716 2056872 5401
```

The big object is at the bottom: 2MB. To find out what file it is, you'll use the `rev-list` command, which you used briefly in Chapter 7. If you pass `--objects` to `rev-list`, it lists all the commit SHAs and also the blob SHAs with the file paths associated with them. You can use this to find your blob's name:

```
$ git rev-list --objects --all | grep 7a9eb2fb
7a9eb2fba2b1811321254ac360970fc169ba2330 git.tbz2
```

Now, you need to remove this file from all trees in your past. You can easily see what commits modified this file:

```
$ git log --pretty=oneline -- git.tbz2
da3f30d019005479c99eb4c3406225613985a1db oops - removed large tarball
6df764092f3e7c8f5f94cbe08ee5cf42e92a0289 added git tarball
```

You must rewrite all the commits downstream from 6df76 to fully remove this file from your Git history. To do so, you use `filter-branch`, which you used in Chapter 6:

```
$ git filter-branch --index-filter ✗
'git rm --cached --ignore-unmatch git.tbz2' -- 6df7640^..
Rewrite 6df764092f3e7c8f5f94cbe08ee5cf42e92a0289 (1/2)rm 'git.tbz2'
Rewrite da3f30d019005479c99eb4c3406225613985a1db (2/2)
Ref 'refs/heads/master' was rewritten
```

The `--index-filter` option is similar to the `--tree-filter` option used in Chapter 6, except that instead of passing a command that modifies files checked out on disk, you're modifying your staging area or index each time. Rather than remove a specific file with something like `rm file`, you have to remove it with `git rm --cached` — you must remove it from the index, not from disk. The reason to do it this way is speed — because Git doesn't have to check out each revision to disk before running your filter, the process can be much, much faster. You can accomplish the same task with `--tree-filter` if you want. The `--ignore-unmatch` option to `git rm` tells it not to error out if the pattern you're trying to remove isn't there. Finally, you ask `filter-branch` to rewrite your history only from the 6df7640 commit up, because you know that is where this problem started. Otherwise, it will start from the beginning and will unnecessarily take longer.

Your history no longer contains a reference to that file. However, your reflog and a new set of refs that Git added when you did the `filter-branch` under `.git/refs/original` still do, so you have to remove them and then repack the database. You need to get rid of anything that has a pointer to those old commits before you repack:

```
$ rm -Rf .git/refs/original
$ rm -Rf .git/logs/
$ git gc
Counting objects: 19, done.
Delta compression using 2 threads.
Compressing objects: 100% (14/14), done.
Writing objects: 100% (19/19), done.
Total 19 (delta 3), reused 16 (delta 1)
```

Let's see how much space you saved.

```
$ git count-objects -v
count: 8
size: 2040
in-pack: 19
packs: 1
size-pack: 7
prune-packable: 0
garbage: 0
```

The packed repository size is down to 7K, which is much better than 2MB. You can see from the size value that the big object is still in your loose objects, so it's not gone; but it won't be transferred on a push or subsequent clone, which is what is important. If you really wanted to, you could remove the object completely by running `git prune --expire`.

Summary

You should have a pretty good understanding of what Git does in the background and, to some degree, how it's implemented. This chapter has covered a number of plumbing commands — commands that are lower level and simpler than the porcelain commands you've learned about in the rest of the book. Understanding how Git works at a lower level should make it easier to understand why it's doing what it's doing and also to write your own tools and helping scripts to make your specific workflow work for you.

Git as a content-addressable filesystem is a very powerful tool that you can easily use as more than just a VCS. I hope you can use your newfound knowledge of Git internals to implement your own cool application of this technology and feel more comfortable using Git in more advanced ways.