

大串 肇・久保靖資・豊沢泰尚 [共著]



Git

大ヒットした
入門書が
ついに改訂！



が、おもしろいほどわかる 基本の使い方 33

改訂新版

[Chapter 1]
バージョン
管理の基本



[Chapter 2]
Gitの
基本的な
使い方



[Chapter 3]
複数メンバー
での運用

バージョン管理で
もう失敗しない。
これだけ覚えれば
現場でも大丈夫！

[Chapter 4]
Gitを使った
実践開発

間違って上書きしちゃった…。

最新ファイルがわからない…。

book.MdN.jp
MdN
ミディアムドキュメント

Git

が、おもしろいほどわかる
基本の使い方 33

改訂新版

大串 肇・久保靖資・豊沢泰尚
[共著]

エムディエヌコーポレーション

はじめに

現

在、世界の多くのプログラマがGit（ギット）を利用してコードを効率的に管理するようになり、生産性が大幅に向かっています。本書を手にとっていただいたみなさんも、バージョン管理の重要性やGitについて耳にすることがあり、「便利そうだな」「使ってみたいな」と思ったことがあるのではないでしょうか。

一方、Gitに関する情報はプログラマがプログラマ向けに発信するものが多く、ターミナル（いわゆる黒い画面）でコマンドを入力しなければ活用できない、敷居の高いツールというイメージを持っている方も多いと思います。

ただし、プログラマ以外でも、たとえばWebクリエイターやWebディレクターであっても、業務にGitを取り入れることはかなり有効です。実際、仕事にGitを取り入れることで、次のような課題は半自動的に解消されるでしょう。

- ・詳細な作業履歴をメンバーと共有していく、どうしても漏れが出ててしまう

- ・複数人で同時に同じコンテンツの編集作業をしていると「先祖返り」が起きてしまう

- ・同時進行していると、変更したファイルが行方不明になってしまう
- ・バックアップは定期的に取っていても、実際に調べるとなると骨が折れる

こうした悩みは、プログラマだけのものではないはずです。

本書は、プログラマではない方でも、まずは手軽に仕事の中にGitを取り入れて、生産性の向上に役に立てていただくための入門書となることを目指しています。そのため、コマンドを入力しなくてもGitの恩恵を得ることのできる“Sourcetree”というGUIツールを利用することを前提としています。

また、主にクライアントワークにGitを活用することを想定し、オープンソースでの開発の中心となっている“GitHub”ではなく、非公開プロジェクトの管理に特化した“Bitbucket”というホスティングサービスを利用して学習することをおすすめしています。

さらに、初心者の方にもGitやSourcetreeを利用している状況をイメージしやすいように、イラストや図のほかに、実際のツールの画面も多く掲載しています。

同時に、現場での仕事で使うシーンをイメージしやすいように、特に後半の活用編では架空のWeb制作現場の作業者たちがGitを利用することで共同作業をスムーズに進めていく様子を通して、Gitの機能の活用方法を紹介しています。ぜひ、ご自身の仕事の現場の状況に置き換えてイメージしながら読み進めてください。

[本書の主な対象読者]

- ・Web制作の現場で仕事をするデザイナー、マークアップエンジニア、ディレクターの方
- ・Gitがどのように仕事に役立つか興味のある方

[本書の構成]

前半は、GitおよびSourcetreeのインストールから、基本的な使い方を紹介しています。

後半は、機能よりも、活用すると便利な状況にフォーカスしながらGitおよびSourcetreeの機能を紹介しています。

筆者は、実際に多くのWeb制作企業へのGit導入を支援することで、共同作業における生産性の改善を行っています。その経験を踏まえて、本書を通じてGitを仕事に取り入れることで、明日からのあなたの仕事の生産性が向上すると信じています。

そして、「なぜ今までこんなに便利なものを使ってこなかったんだろう……」と感じただければ幸いです。

前著を上梓したときからすでに4年が経過しました。その間にSourcetree、およびGit、Bitbucketも進化し、インターフェイスも変更されています。今回の改訂にあたっては2019年5月時点での最新版をベースとして、解説中のメニュー名や画面などを全ページに渡って改編しています。ただし、Sourcetree、Bitbucketともに今後もインターフェイスなどに変更があることをあらかじめご了承ください。

本書の執筆にあたっては、多くの方々のご協力を賜りました。ここに感謝の意を表します。

2019年5月
共著者一同

CONTENTS

—目次—

本書の使い方	006
本書のデモリポジトリについて	008

Chapter 1 バージョン管理の基本

1-01 Gitを使ったバージョン管理	010
1-02 Git機能を提供するWebサービス	014
1-03 Sourcetreeをインストールする	018
1-04 ローカルリポジトリをつくる	022
1-05 コミットする手順を覚える	027
1-06 本書のデモリポジトリを利用する	034
1-07 Bitbucketの画面を確認する	039
1-08 GitHubアカウントを連携する	045

Chapter 2 Gitの基本的な使い方

2-01 変更をコミットする	048
2-02 Bitbucketにプッシュしてみよう	057
2-03 コミットログ(履歴)を見てみよう	062
2-04 間違って操作してしまったら?	066
2-05 ファイルをGitの管理から外す	069

Chapter 3 複数メンバーでの運用

3-01 Gitの作業環境を準備しよう	074
3-02 複数のPCでクローンするには	080
3-03 ファイルのマージを理解しよう	084
3-04 コンフリクトはなぜ起こる?	090
3-05 コンフリクトを解決する	094
3-06 コンフリクトしない状態を保つ	100
3-07 ブランチでフォルダを管理しよう	104
3-08 タグやスタッッシュを使いこなそう	109
3-09 ひとりでブランチを活用してみる	114
3-10 複数人でブランチを運用する	122

Chapter 4 Gitを使った実践開発

4-01 作業内容を過去の状態に戻す	132
4-02 未コミットの作業を取り消す(破棄)	138
4-03 過去の修正を元に戻す(打ち消し)	140
4-04 リモートリポジトリの最新状態を取得	143
4-05 コミットツリーから何がわかるか	149
4-06 コミットログを整理しよう	152
4-07 ファイルの移動、複製の追跡	157
4-08 Git Flowを使った運用	160
4-09 便利なGitの機能を活用	165
4-10 Webサイトの更新をGitで行う	172

用語索引	181
執筆者プロフィール	183

本書の使い方

本書は、Gitを使ったバージョン管理をはじめて行う方に向けた解説書です。ホスティングサービス「Bitbucket」とクライアントソフト「Sourcetree」を用いて、Gitを操作する方法を解説しています。

本書は次のような構成になっています。

2-01

記事タイトル&冒頭のイラスト

記事のテーマタイトルと、各記事で解説している内容的理解を助けるイラストです。

ポイント

各記事を通じて解説している内容や、得られる情報を箇条書きで挙げています。

使用するコマンド

各記事で解説している操作をSource treeからではなく、コマンドライン(CUI)を使って行う際のコマンドを記載しています。記事内容によっては記載されていない場合があります。

使用するSourcetreeの機能

各記事で使用するSourcetreeの機能を記載しています。記事内容によっては記載されていない場合があります。

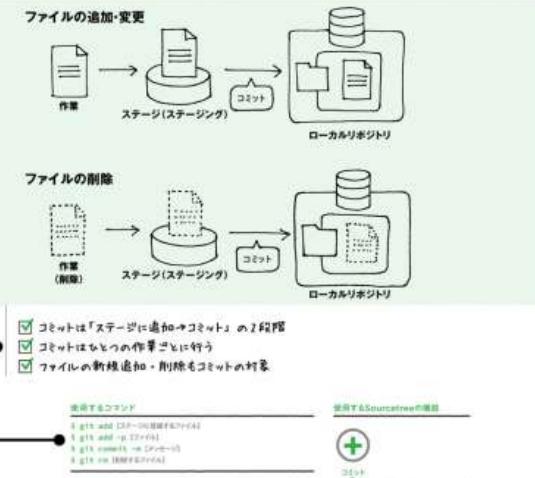
Chapter 1

バージョン管理の基本

SourcetreeでGitを操作する前に「バージョン管理」の考え方を見ていきます。Sourcetreeのインストールと基本設定、Bitbucketの概要についても解説します。

変更をコミットする

本節では、いよいよSourcetreeを操作して、Gitのバージョン管理がどのようなものか学んでいきましょう。本節では、まず最初に見えるべき、ファイルの変更履歴を保存する「コミット」という操作を取り上げます。ファイルの新規作成、削除、ファイルの内容変更など、さまざまな変更はすべてコミットの対象となります。



D48

Chapter 2

Gitの基本的な使い方

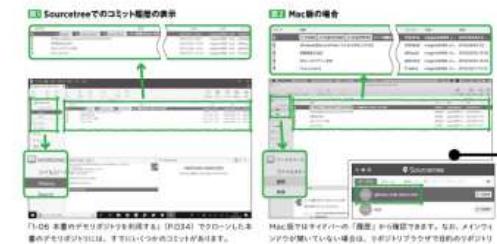
Gitの操作方法の解説となります。「コミット」、「プッシュ」、「ブル」などの基本的な操作を、コマンドラインを使わずにSourcetreeの画面上から行う方法を学んでいきましょう。

▶ コミットとは？

既に開発が始まると、新しいファイルを作成したり、不要なファイルを削除したり——さまざまなファイル操作が行われています。それを変更履歴として記録するのが「コミット」です。

「P-05 ほのぼのと手を弄す」(P-027) では、まず

リポジトリ内では、コミットを行うことで、変更の履歴がついた形で記録されています。利用者は、これまでに、どのファイルのどの部分が変更されたのか、またその変更はいつ誰が行ったのなどを知ることができます。また、コミット単位で、過去におけるばっつファイルを取得したり、現時のファイルをある時点まで戻したりすることができます。この小さなコミットの原理は、Sourcetreeでもおさげでいる「History」(Macは「履歴」)を頭に置いておきましょう。



D49

Chapter 3

複数メンバーでの運用

本章からは、1つの成果物を複数人が共同作業で開発・制作していく場合の使い方を説明していきます。上手に取り入れることで、チーム体制で同時に進める作業を効率化できます。

Chapter 4

Gitを使った実践開発

開発や制作の過程でよく起る事例をもとに、実践的な使用方法を解説します。機能を知ると同時にどういう状態を理解することが、Gitを深く使いこなす力になります。

解説文

バージョン管理や該当するGitの機能の考え方、Sourcetreeの操作方法など解説しています。

解説文を補足するための仕組みを示した図や、操作画面の説明、Gitのツリー図などを掲載しています。

▶ コミットは「ステージに追加→コミット」の2段階で行う

ま
じ
に
コ
ミ
ッ
ト

じ
に
追
加
→
コ
ミ
ッ
ト

い
う
2
段
階
で
行
う
と
い
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

ど
う
で
て
接
続
す
ま
し
や

こ
ミ
ッ
ト
を
行
う
と
い
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

こ
そ
を
覚
え
て
お
き
ま
し
ょ
う

1

Chapter 1 バージョン管理の基本

本書のデモリポジトリについて

本書では、ホスティングサービスBitbucketとクライアントソフトSourcetreeを用いて、Gitを使う方法を解説しています。本書の解説内容を学習していただくために、Bitbucketにデモリポジトリを作成しています。URLは下記の通りです。

▶ <https://bitbucket.org/GitBook-MdN/gitbook-mdn-demo>

デモリポジトリの使用方法

- ① デモリポジトリを利用するにはBitbucketのアカウントが必要です。本書の「Chapter 1-03 Sourcetreeをインストールする」(18ページ)を参照のうえ、Bitbucketのアカウントを取得し、Sourcetreeをインストールしてください。
- ② 次に、本書のデモリポジトリをご自身のBitbucketアカウントにフォークします。さらに、そのクローンをお使いのPCに作成します。操作方法については本書の「Chapter 1-06 本書のデモリポジトリを利用する」(34ページ)を参照してください。この作業を行うことで、元のデモリポジトリとは別にご自身のリポジトリとして、デモリポジトリに用意したリソースを利用できるようになります。

デモリポジトリに関するご注意

- ・本書で提供するデモリポジトリのファイルは、本書執筆後に細かな修正・調整を行っています。本書の解説中の図などで示しているコミット履歴と実際のものが異なる場合がありますので、あらかじめご了承ください。
- ・本書のデモリポジトリのファイルは、本書の解説内容をご理解いただくために、ご自身で試される場合にのみ使用できる参考用データです。その他の用途での使用や配布などは一切できませんので、あらかじめご了承ください。ただし、サンプルファイル中に含まれるJavaScriptや各種プラグインなどの一般に公開されているライブラリ類については、各ライブラリのライセンス条件に従います。
- ・本書のデモリポジトリのデータの著作権は、それぞれの制作者に帰属します。
- ・本書のデモリポジトリのデータを実行した結果については、著者および株式会社エムディエヌコーポレーションは一切の責任を負いかねます。お客様の責任においてご利用ください。

©2019 Hajime Ogushi, Yasushi Kubo, Yasutaka Toyosawa. All rights reserved.

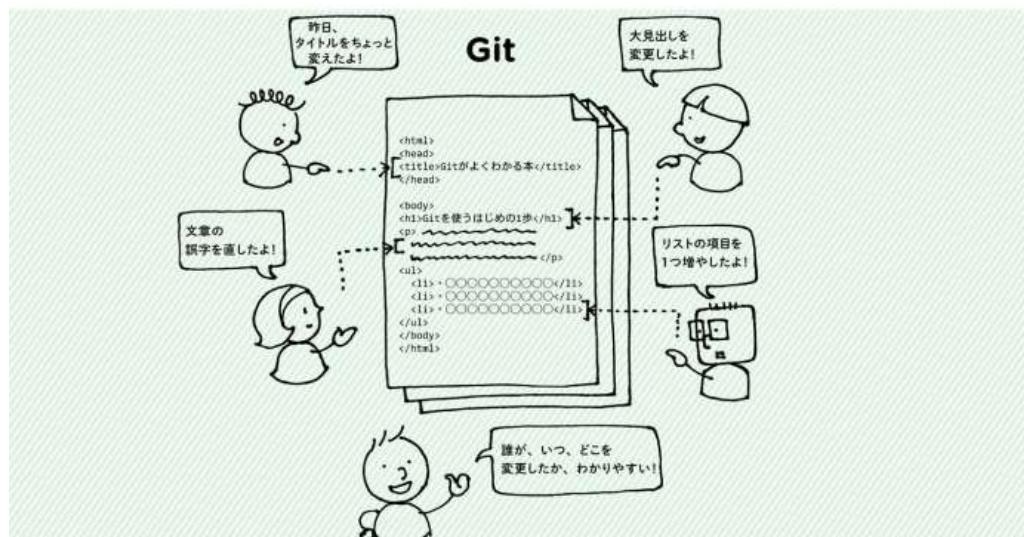
本書は著作権法上の保護を受けています。著作権者、株式会社エムディエヌコーポレーションとの書面による同意なしに、本書の一部あるいは全部を無断で複写、複製、転記、転載することは禁止されています。

本書は2019年5月現在の情報を元に執筆されたものです。これ以降の仕様等の変更によっては、記載された内容と事実が異なる場合があります。本書をご利用の結果生じた不都合や損害について、著作権者及び出版社はいかなる責任も負いません。

Gitを使ったバージョン管理

チームで開発を行う上で、もはや欠かせない存在となった Git。

でも、Gitを利用したことがない方、バージョン管理がよくわからない方は、Gitが何をするものなのかもよくわからないでしょう。まずはGitによるバージョン管理の大まかなイメージをつかんでおきましょう。



- 分散型バージョン管理システムとは
- ファイルのバージョンを進めたり戻したりできる
- チームで開発を行う場合の問題を解決できる

▶ 誰がいつ、どのファイルの何を変更したかを管理

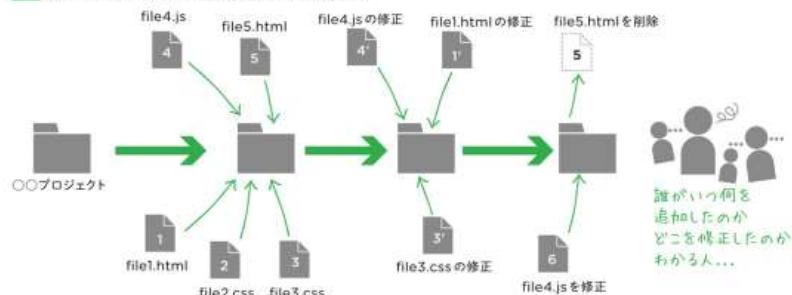
ある開発プロジェクトをチームで行っている現場。「おーい、このファイル誰が更新した?」「あ、それボクです。」「え、どこ変えたの?」「えーとそれは……何項目だったけな……」「1カ所だけか?」「いや、何カ所か。あれ、メモったんだけど、えーとそのメモをどこのフォルダに入れたかと……」「あ…誰だよ、ファイル上書きしちゃったやつ!?」「あ、それ私。」「おれの書いたところが元に戻ってんじゃん!」というわけであつて、という間に1日過ぎてしまった……ではお話になりませんね(図1)。

それにしてもこれだけいろいろなソフトが開発されているのですから、どこかにこういうことを全部管理してくれる便利なシステムはないものか。それがGitなどの「バージョン管理システム」(Version Control System = VCS)です。

図1 自分でバージョンを管理するのは難しい

えーいまさら4月9日の修正のこと言われても忘れちゃいましたよ	Index_20190429final3.html	2019/04/29 16:05
あー修正漏れ...	Index_20190418final2.html	2019/04/18 15:29
よし、これでFinish!	Index_20190418final.html	2019/04/18 14:31
あれ、この修正誰がどこを直したんだろう?	Index20190409-2.html	2019/04/09 14:15
あれー修正入っちゃったよ	Index20190409.html	2019/04/09 11:43
よし、終わったー	Index.html	2019/04/05 15:30

図2 グループでバージョンを管理するのはもっと難しい



▶ Gitによるバージョン管理の利点

Gitによるバージョン管理では、管理されたプロジェクトにおいて、いつ、誰が、どのファイルのどの箇所を、どういった目的で作成、変更、削除したかという履歴を残し、共有することができます。たとえば、最新のファイルとは、履歴の最も新しいファイル

ルということになります。過去の状態を確認したり、その時点に復元したりすることもできます(図3)。

誤ってファイルを削除してしまった場合でも、直前の履歴にファイルを戻すことで、削除したファイルを復元することができるです。

図3 Gitは変更者、変更日時、変更箇所とその内容まで記録している



▶ Gitにおける作業の流れ

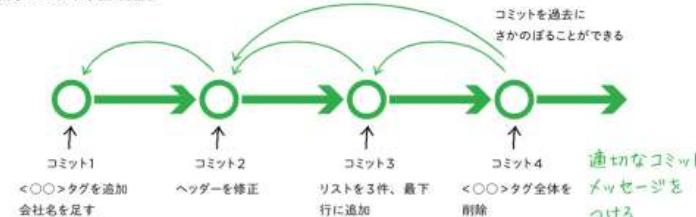
Gitがバージョン管理下に置く場所を「リポジトリ」と呼びます。リポジトリには、「ローカルリポジトリ」と「リモートリポジトリ」の2種類がありますが、この説明はP.015で行います。ここではGitが管理するリポジトリという場所があると覚えてください。

ユーザーが変更の履歴を記録する作業を「コミット」(commit)と呼びます。コミットは時系列のつながりを持っています。順々に記録されるので、コミットを過去にさかの

ぼることができます。ファイルをコミットの単位で戻すことができます。コミットには必ず「コミットメッセージ」(→P.029)というコメントをつける必要があります。誰にもわかりやすいコミットメッセージをつけることで、履歴がよりわかりやすくなります(図4)。

Gitを利用した作業は、基本的に「修正→コミット」の繰り返しです。ユーザーはコミットメッセージを参考に、コミット時のファイルの状態を確認できます。

図4 時系列にコミット単位で記録



▶ 作業の流れを分岐するブランチ

また、Gitには、「ブランチ」(branch)と呼ばれる履歴を枝分かれさせて記録する機能があります。たとえば、メインの開発をマスター・ブランチとして、それとは別にサブ的な機能を分けて開発を進める、あるいはバグフィックスを切り分けて行うというような場合に、ブランチを分けて開発を進めることができます。

ブランチはいくつでも作ることができます。ブランチに

よって枝分かれした履歴は、ほかのブランチの影響を受けることはなく、ブランチごとに作業を進められます。

さらに、枝分かれしたブランチは「マージ」(merge)と呼ばれるブランチ同士を結合する機能によって、ひとつにまとめることが可能です(図5)。元となるブランチから作業のブランチに枝分かれし、作業が完了したら、元のブランチにマージして、ひとつにまとめることも可能です。

図5 枝分かれとマージの例



▶ 分散型と集中型の違い

バージョン管理システムの主な種類として「集中型」と「分散型」があります。Gitは分散型のバージョン管理システムです。集中型のバージョン管理システムで有名なものにはSVN(Subversion: サブバージョン)があります。

集中型ではリポジトリはひとつであり、サーバ上に中央リポジトリが設置されます。利用者はそれに対してチェックアウトとコミットを行うことで、開発を進めていきます。このため、最新の情報を取得するにも、何か作業を行ったあとにコミットを行うときにも、必ず中央サーバに接続する必要があります。また、万が一中央リポジトリが破損した場合は、復旧が難しくなる恐れがあります(図6)。

分散型では、それぞれの端末ごとにローカルのリポジトリが作成されます(図7)。集中型がひとつの中央リポジトリで管理されるのに対し、複数のリポジトリが作成されるので、分散型と呼ばれます。自分の端末に作成したローカルリポジトリに対してはいつでも変更の登録(コミット)が可能なので、リモートリポジトリにプッシュしたり、リモートリポジトリからプルしたりする作業を除けば、オフラインでも作業が可能です。また、複数のリポジトリが作成されることで、どこかのリポジトリが破損した場合にも、比較的容易に復旧することが可能です。

図6 集中型のバージョン管理システム

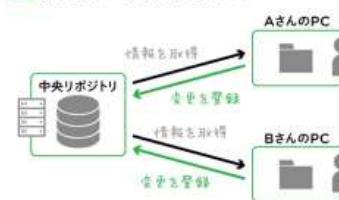
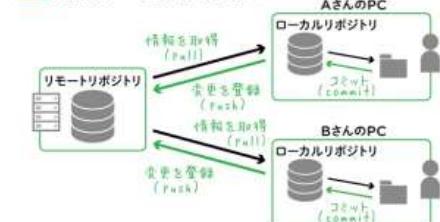
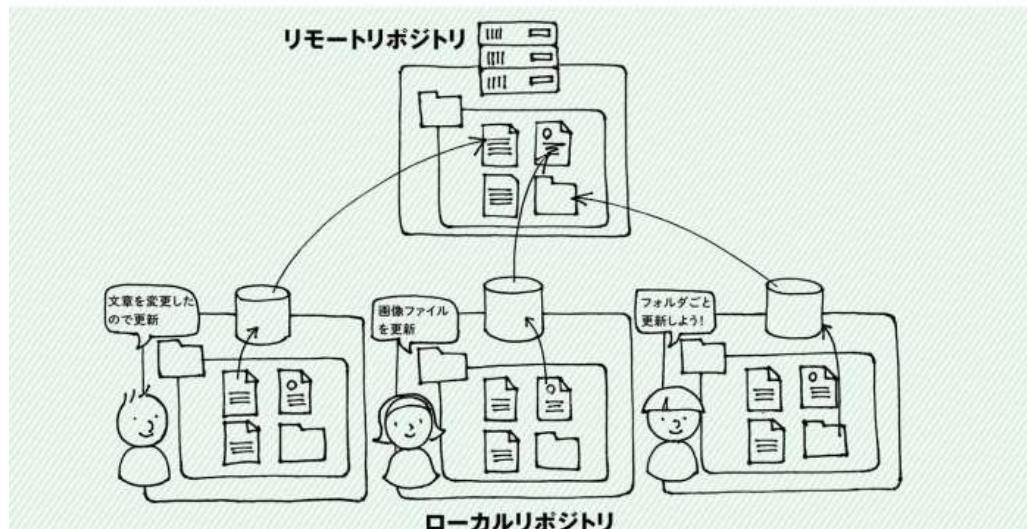


図7 分散型のバージョン管理システム



Git機能を提供するWebサービス

ファイルの変更、追加、修正などを管理する場所である「リポジトリ」は、自分のPCにも、ネットを介したサーバ上にも作成できます。サーバを設定するには専門知識を必要としますが、サーバ機能を提供してくれる便利なWebサービスのひとつがBitbucketです。



- ローカルリポジトリとリモートリポジトリの違い
- BitbucketやGitHubが主に提供するのはリモートリポジトリ
- Bitbucketではプライベートリポジトリを無料で作ることができる

使用するコマンド

```
$ git --version
```

▶ ローカルリポジトリとリモートリポジトリ

前述(P.012)したように、Gitによってバージョン管理を行う場所として指定した場所を「リポジトリ」といいます。自分ひとりだけで開発・制作、バージョン管理を行う場合は、リポジトリは自分のPCにあればよいでしょう。

しかし、リポジトリをサーバに用意して、ネットを介してどこからでも利用したい、あるいはチームでひとつのリポジトリを利用して開発したい、というようなときもあるでしょう。いずれにしても、この場合はサーバ上にリポジトリを置くこ

とになります。このようなリポジトリを「リモートリポジトリ」といいます。

前者の自分のPCに置くだけで完結するようなリポジトリを「ローカルリポジトリ」といいます。^{図1}

複数の人で作業をする場合は、一般にリモートリポジトリのクローン(→P.034)を各自のPCにつくり、ローカルリポジトリとして作業を進めていきます。そして作業が終わったら、変更をリモートリポジトリに反映(プッシュ)します^{図2}。

図1 ローカルリポジトリ

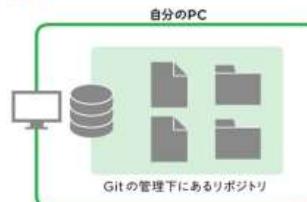
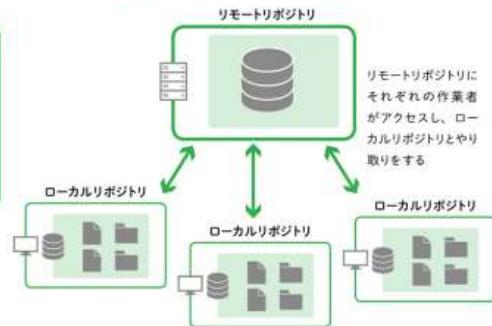


図2 リモートリポジトリとローカルリポジトリ



▶ リポジトリのホスティングサービス

リモートリポジトリを利用する場合、自分でGit用のサーバを構築できればいいのですが、それには専門知識を要しますし、費用なども考えるとなかなか敷居が高いと言えます。幸い、ネット上にはリモートリポジトリをホ

スティングするサービスを提供しているWebサービスがいくつあります。本書で扱うBitbucketもそのひとつです^{図3}。また、オープンソースの開発で広く利用されているGitHubも有名です^{図4}。

図3 Bitbucket (<https://bitbucket.org>)



図はサインインしてリポジトリにアクセスした画面

図4 GitHub (<https://github.com>)



図はサインインしてオープンソースのリポジトリにアクセスした画面

▶ BitbucketとGitHubの違い

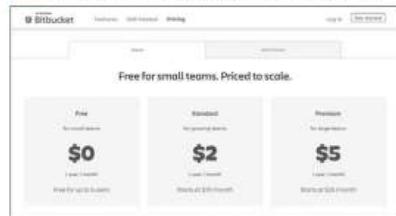
本 書で取り上げるBitbucketは、リポジトリの共同利用ユーザーが5名以内であれば、無料で公開、非公開を問わずリポジトリを無制限に作成することが可能です。また、後述するGitのGUIツールとして有名な「Sourcetree」と同じ開発元Atlassian社が提供しているため、Sourcetreeと一緒に組み合わせて利用しやすくなっています。**図5**。

なお、同じGitのWebサービスとして有名なGitHub

も同様に、無料で非公開のリポジトリを無制限で作れます。GitHubの場合、公開リポジトリであれば、ユーザーは無制限です。**図6**。

値段を比較すると、全体的にBitbucketのほうが安価になっています。コスト面を重視するのであれば、商用の開発や、外部には出せないプロジェクトなどではBitbucket、オープンソース系のプロジェクトではGitHubというように使い分けてもよいかもしれません。

図5 Bitbucketの料金 (<https://bitbucket.org/plans>)



Bitbucketは非公開リポジトリのユーザー数によって料金が決まります。5ユーザーまでであれば非公開リポジトリを無料で作成できます。

図6 GitHubの料金 (<https://github.com/pricing>)



GitHubは非公開リポジトリのユーザー数によって料金が決まります。5ユーザーまでであれば非公開リポジトリを無料で作成できます。

▶ リポジトリを操作するGitのGUIクライアント「Sourcetree」

G itそのものはターミナル（コンソール）上で「git add index.html」「git checkout master」というようにコマンドを打ち込んで操作するコマンドラインベースのシステムです。慣れない人にとってはいちいちコマンドを打ち込むのは大きな負担になりますし、そもそもコマンド操作は敷居が高いという方も多いでしょう。

というわけで、Gitをウインドウやアイコンで操作できるようにしたGUIクライアントソフトが多く開発されています。

Sourcetreeは、Gitを見たままに操作できるように、わかりやすく表示してくれることで定評のあるソフトです。**図7**。

SourcetreeはWindowsとMacの両対応で、日本語にも対応しており、さらに無料です。こういった魅力的なソフトが出てきたことで、これまでエンジニアにしか使われなかったGitが、Webデザイナーやディレクター、Web担当者にまで広まるようになったとも言えるでしょう。

また、本書でも取り扱う、BitbucketやGitHubといったリモートリポジトリを提供するサービスとの連携についても扱いやすくなっています。すでにアカウントを持っている場合は、IDとパスワードを入力することで、既存のリポジトリをすぐに利用することができます。

図7 Sourcetreeの操作画面



▶ Gitを使うには

本 書では基本的にSourcetreeが内蔵しているGitを利用します。Sourcetreeのインストールが完了していれば、追加の操作は不要です。もしSourcetreeを使わずにGitを利用したいという場合は、別途Gitのインストールが必要になります。

Windowsの場合は、Gitの公式サイト (<http://git-scm.com/>) よりGitのインストーラーをダウンロードしてインストールしましょう。**図8**。

インストールするとGit GUIというSourcetreeのようにマウスでGitを操作できるソフトと、Git BashというコンソールでGitを操作できるソフトが利用できるようになります。

Macの場合は、もともとGitがインストールされていますが、Sourcetreeを使わずにコマンド操作でもGitを利用したいという場合は、コマンドライン・デベロッパ・ツールのインストールが必要です。「アプリケーション」→「ユーティリティ」から「ターミナル」を開いて

`$ git --version` return

と入力してみてください。

`git version 2.20.1`

というようにバージョン情報が表示されれば、利用できる状態です。

もしバージョン情報が表示されずには、**図9**のようにコマンドライン・デベロッパ・ツールのインストールを促された場合は、インストールを行いましょう。

インストール後、再度

`$ git --version` return

と入力してバージョン情報が表示されれば、完了です。

なお、MacではHomebrew (<http://brew.sh/>) というパッケージ管理システムを利用してGitをインストールする方法もよく利用されますが、本書では割愛します。

図8 Gitの公式サイト (<https://git-scm.com>)



MEMO 本書の執筆環境

本書でインストールしているSourcetreeのバージョンは、Windowsでは3.1.3、Macでは3.1.2です。お使いのSourcetreeのバージョンによっては、操作画面やメニュー名、ボタン名などが異なる場合があります。なお、GitのバージョンはWindowsでは2.21.0、Macでは2.20.1を使用しています。

図9 コマンドライン・デベロッパ・ツールのインストール (Mac)

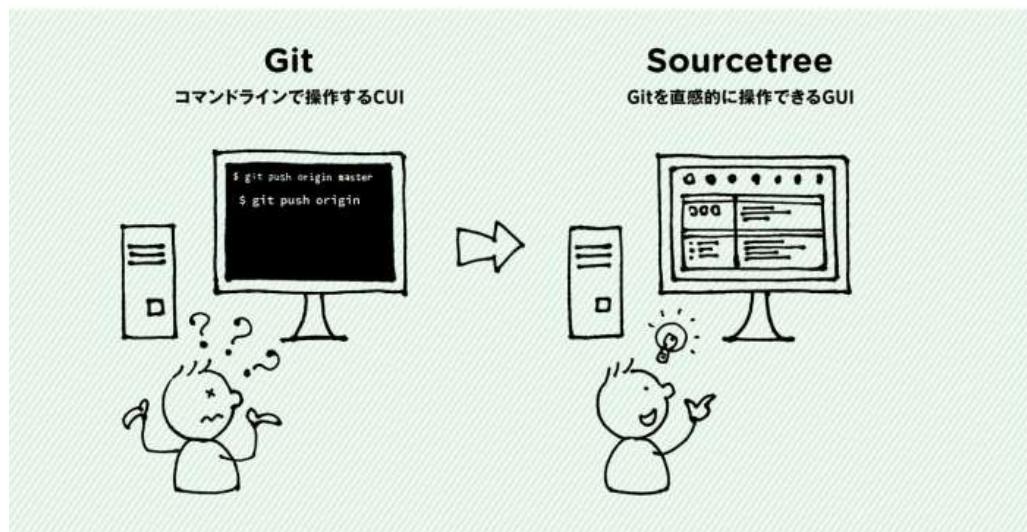


1-03

Chapter 1
バージョン管理の基本

Sourcetreeをインストールする

Sourcetreeは、Gitのコマンド操作をボタンなどのGUIで行えるようにしたGitクライアントツールで、CUIのコマンド操作は苦手、Gitを使うのは初めてという方に最適なツールです。本書はSourcetreeを使ってGitの基本操作を学んでいきます。まずはSourcetreeのインストールからです。



- ✓ SourcetreeのWebサイトにアクセスしてダウンロード
- ✓ Windowsではインストーラを使う。Macは解凍して「アプリケーション」フォレダにコピー

▶ インストーラのダウンロード

Sourcetreeは、以下の公式サイトから入手できます。

<https://www.sourcetreeapp.com>

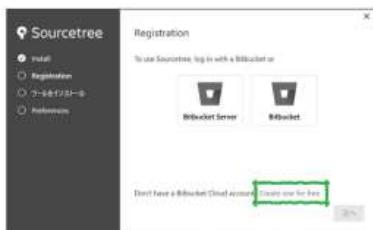
WindowsとMac、使用OSが自動的に判別されダウンロードページが表示されますので、「Download for Windows」(もしくは「Download for Mac OS X」)をクリックして、インストーラをダウンロードしてください。**図1**

① Sourcetreeインストーラのダウンロード



▶ Windows版のインストール

ダウンドロードできたら、インストーラを起動し、指示に従ってインストールを行います。



① インストーラ起動後最初の画面。Bitbucket Cloudのアカウントとのひも付けが必要となるため、下部の「Create one for free...」のリンクをクリックします。



② ブラウザが立ち上がり、アカウントを作成する画面が開きます。メールアドレスを入力します。



③ 続けて、ユーザー名とパスワードを入力し、reCAPTCHAの「私はロボットではありません」にチェックを付けます。



④ 画像によるテストが始まる場合は、質問の内容に該当する画像をすべてクリックした後、「確認」ボタンをクリックします。その後、「Agree and sign up」と書いてある緑のボタンをクリックします。



⑤ 入力したメールアドレスに認証用のメールが送られます。メールを確認します。



⑥ 認証用のメールを確認し、「Verify my email address」をクリックします。



⑦ Bitbucket Cloudの一意のユーザー名を作成するということで、任意のユーザー名を入力し、「続行」と書いている緑のボタンをクリックします。



⑧ アンケートが始まる場合は、答えた後、「送信」ボタンをクリック。もしも「スキップ」をクリックします。



⑨ アカウントの登録が完了し、Bitbucketの「あなたの作業」のページが表示されます。以上でアカウントの作成は完了です。インストール画面に戻ります。



⑩ インストール画面において、右側の「Bitbucket」と書かれているボタンをクリックします。



⑪ Bitbucketにログインしている状態であれば、OAuth認証の確認画面が開きます。「アクセスを許可する」と書いてある青いボタンをクリックします。



⑫ インストール画面において、「登録が完了しました!」というメッセージが表示されたら、連携が完了した状態です。「次へ」をクリックします。



⑬ GITと同じバージョン管理ツールとしてMercurialをいっしょにインストールするかどうかを聞かれますが、本書では利用しませんので、チェックを外した状態で「次へ」をクリックします。



⑭ SSHキーを読み込むかどうかのダイアログです。いったん「いいえ」をクリックしてスキップします。もし、すでにSSHキーを持っている場合は追加してもかまいません。（P.178参照）



⑮ Sourcetreeを利用してGITを操作する際のユーザー名とメールアドレスを入力します。こちらは任意のものでOKです。「この詳細をすべてのリポジトリに使用する」にチェックを付けると、今後すべてのリポジトリでの作業がこちらのユーザー名とメールアドレスにひも付きます。リポジトリごとに変更が必要がない場合はチェックを付けてから「次へ」をクリックします。



⑯ Sourcetreeの初期画面が開きました。こちらでインストールは完了です。

Mac版のインストール

Windowsと同様に「<https://www.sourcetreeapp.com>」にアクセスし、「Download for Mac OS X」ボタンをクリックすると、インストーラがzip形式の圧縮ファイルでダウンロードされますので、解凍してください（図2）。



① 解凍したアプリケーションファイルである「Sourcetree.app」のアイコンを「アプリケーション」フォルダに移動します。その後、ダブルクリックすると、Windowsと同様にセットアップがはじまります。以降は、Windowsと基本的に同じ手順となります。

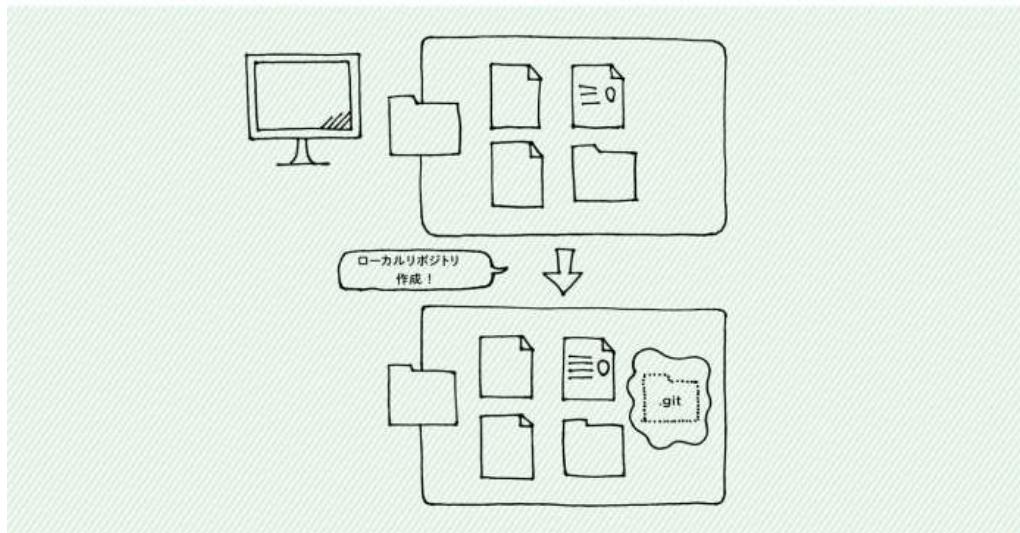
2 Sourcetreeインストーラのダウンロード（Mac）



② ステップを進めていく上で、ユーザーアカウント登録後、OAuth認証を行なうところ（前ページ⑩の手順）では「Bitbucketクラウド」を選択します。

ローカルリポジトリをつくる

Sourcetree をインストールしたら、まずは自分の PC にローカルの練習用リポジトリを作ってみましょう。本節より、この練習用ローカルリポジトリを使って、Sourcetree や Git のごく基本的なことを学んでいきます。それでは始めましょう！



- ローカルリポジトリを作る
- リポジトリフォレダには「.git」フォレダが作られる
- ローカルリポジトリを削除する

使うコマンド

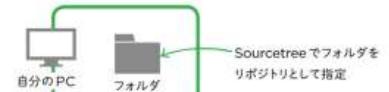
\$ git init

▶ ローカルリポジトリを作る

Git は複数のメンバーでひとつの成果物を作り上げていくときのバージョン管理に威力を発揮しますが、個人のファイル管理、バージョン管理にも十分活用できます。まずは練習として、個人用に自分の PC にリポジトリを作成して、いろいろ操作してみましょう。

リポジトリの作成手順は特に難しいものではありません。バージョン管理を行いたい PC 上のフォルダを Sourcetree でリポジトリとして指定するだけです。Git の設定は、Sourcetree が自動でやってくれます（図1）。

図1 PCのフォルダをローカルリポジトリに設定



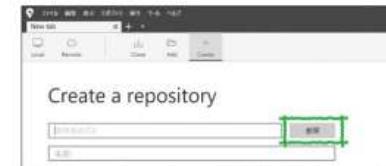
1 Sourcetree のツールバーにある「Create」ボタンをクリックします。



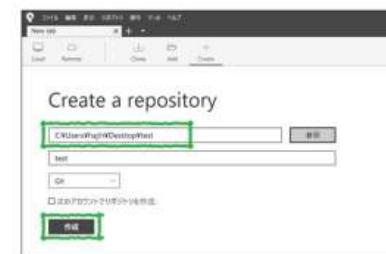
3 「保存先のパスを選択して下さい」ダイアログが表示されます。ここでローカルリポジトリとするフォルダを選択します。ここでは、あらかじめ作成しておいた空の「test」フォルダを選択し、「フォルダの選択」ボタンをクリックします。



5 「出力先ディレクトリエラー」が表示される場合がありますが、指定したディレクトリが空であれば問題ありませんので、「はい」をクリックします。



2 「Create a repository」の画面が表示されますので、「参照」ボタンをクリックします。



4 「保存先のパス」に、先ほど指定したフォルダのパスが入力されます。「作成」ボタンをクリックします。



6 指定したフォルダがリポジトリとして追加され、タブとして表示されます。

▶ 通常は見えない「.git」フォルダがリポジトリの証し

リポジトリに指定したフォルダを見ても、特に変化はありません。そこで、本当にリポジトリに設定されているのか確認してみましょう。

リポジトリに指定されたフォルダには、「.git」というフォルダが作成されます。これがいればリポジトリだとわかるわけです。

ただし、Windows の初期設定では、名前が「.」(ドット)で始まるフォルダやファイルは表示されないようになっています（これらを「隠しファイル（フォルダ）」「ドットファイル（フォルダ）」などと呼ぶことがあります）。

「.」のついたファイルやフォルダを表示するには、リポジ

トリに指定したフォルダのウィンドウを開き、ウィンドウ上部の「表示」メニューから「表示」を選択して、表示関連のリボンを表示します。

「表示／非表示」にある「隠しファイル」にチェックをつけると、「.」のついたファイルやフォルダが表示されるようになります（図2）。

「.git」というフォルダが表示されればリポジトリであることを示します。Gitでは、基本的にこの「.git」フォルダが配置されているフォルダを「リポジトリ」として使用します。「.git」フォルダの中には、リポジトリに対する設定や、今後追加していく履歴などが保存されます。

図2 「.git」フォルダの表示



▶ リポジトリを削除する方法

リポジトリを削除する場合、リポジトリとして使用するのを止める（＝バージョンを管理しない）、あるいはリポジトリとなっているフォルダそのものを削除するという二通りの方法があります。

まず、Sourcetree のタブの右側にある「+」マークを

クリックします（図3）。

表示されるローカルリポジトリ一覧より、削除したいリポジトリを右クリックして表示されるコンテキストメニューで「削除」を選びます（図4）。

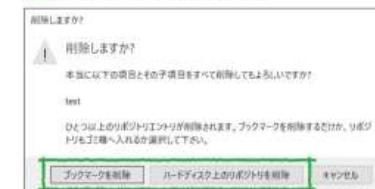
図3 Sourcetreeでタブの右側にある「+」をクリック



すると削除方法を選択するダイアログが表示されます（図5）。

「ブックマークを削除」は、Sourcetree のブックマークのみを削除するもので、ハードディスク内のファイルなどは削除されずに残ります。「ハードディスク上のリポジトリを削除」は、Sourcetree のブックマークだけでなく、ハードディスク内のリポジトリ管理されているファイルやフォルダすべてを削除します。

図5 リポジトリを削除する方法を選ぶ



▶ Macの場合

Mac の場合も操作は Windows とほぼ同じですが、多少ウィンドウの表示やオプション名が変わります。

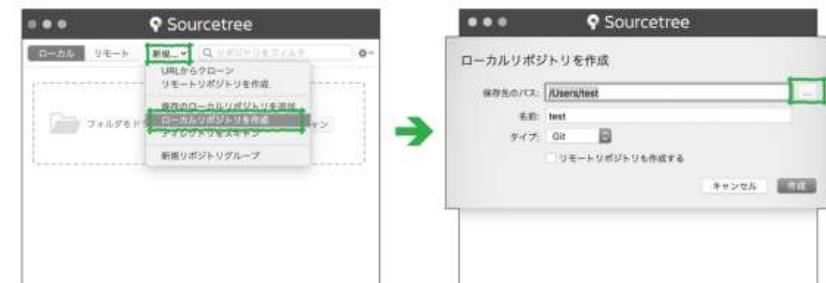
リポジトリの追加はリポジトリブラウザから

ブックマークは、Windows ではタブウィンドウに表示されますが、Mac の場合には「リポジトリブラウザ」として別ウィンドウで表示されます。新規リポジトリの追加などもリポジトリブラウザで行います（図6）。リポジトリ名をダブルクリックすると、メインのウィンドウが表示されます。

「.git」フォルダの表示

Mac でも、先頭に「.（ドット）」のついたフォルダやファイルは通常 Finder には表示されませんが、隠しファイルを表示するショートカットキー（[command] + [shift] + 「.（ドット）」キー）を押すと表示できます。再度同じショートカットキーを押すと非表示になりますので、誤操作を防ぐために通常は非表示にしておくとよいでしょう。

図6 リポジトリの追加はリポジトリブラウザから



リポジトリブラウザの「新規...」をクリックして、表示されるメニューから「ローカルリポジトリを作成」を選択。

「保存先のパス」の「...」ボタンをクリックして、リポジトリとするフォルダを指定します。

リポジトリの削除

こちらもWindowsとは異なり、リポジトリブラウザから行います。リポジトリブラウザにおいて、削除したいリポジトリを選択して右クリックし、メニューから「削除」を選択します^[図7]。

削除方法を選択するダイアログが表示されます。内容はWindowsと同様ですが、一部Mac特有の表記にな

っています。

「ブックマークを削除」は、Sourcetreeのブックマークのみを削除するもので、ハードディスク内のファイルなどは削除されずに残ります。

「ゴミ箱にも移動」は、Sourcetreeのブックマークだけでなく、ハードディスク内のリポジトリ管理されているファイルやフォルダすべてがゴミ箱に移動します。

[図7] リポジトリの削除

リポジトリブラウザで削除したいリポジトリを右クリックし、メニューから「削除」を選択。

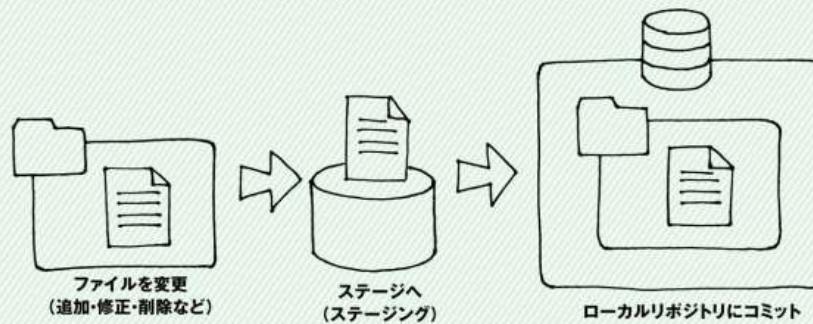


1-05

コミットする手順を覚える

リポジトリができるので、早速ファイルを作成し、コミット（変更履歴を記録）してみましょう。

まずは、ファイルを変更したら「ステージ」と呼ばれる領域にアップ、メッセージをつけてリポジトリにコミットする、この基本手順をマスターしましょう。また、本節では最後にコミット間を移動してGitの機能を確認してみます。



- 変更ファイルをステージに移動する
- メッセージをつけて、リポジトリにコミットする
- コミット間を移動する

使用するコマンド

```
$ git add [ステージに登録するファイル]
$ git commit -m [メッセージ]
$ git checkout [ブランチ名]
```

使用するSourcetreeの機能



▶ ローカルリポジトリにファイルを追加する

ま ず、前節で作成したリポジトリにファイルを追加してみましょう。メモ帳などで「test.txt」というファイルを作り、リポジトリとして登録した「test」フォルダに保存します。ファイルの内容は何でもかまいません（図1・図2）。

この状態でSourcetreeを開き、左サイドバーの「ファイルステータス」をクリックしてみましょう（図3）。

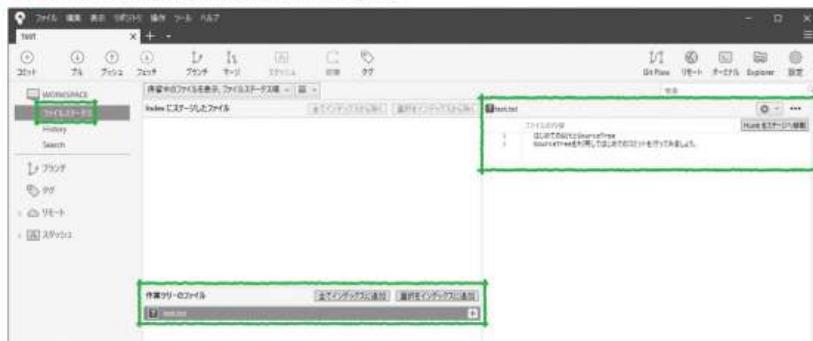
図1 任意の文章を入力



図2 リポジトリのフォルダ内に「test.txt」として保存



図3 ファイルが新規作成されたことをSourcetreeが感知する



▶ コミットする前にステージに移動する

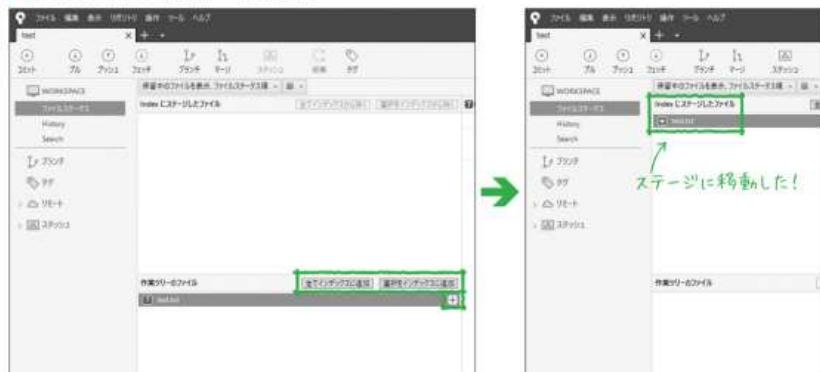
では追加されたファイルを、コミットするためのファイルとして追加してみましょう。test.txtを選択した状態で「+」ボタンをクリックするか、もしくは「全てインデックスに追加」や「選択をインデックスに追加」ボタンをクリックします。すると中央上部の「Indexにステージしたファイル」というエリアにtest.txtが移動します（図4）。

このように、変更したファイルをコミットする場合は、変

更作業を行ったファイルを、ステージ（ステージングエリア、ステージ領域）に移動する行程が必要です。複数のファイルの変更があった場合に、変更の内容によってコミットを分けたいといった場合に利用できます。詳しくは「2-OI 变更をコミットする」（P.048）で説明します。

ここでは、コミットするためにはまずステージに移動するということを覚えておいてください。

図4 変更をコミットする前にステージに移動する



▶ メッセージとともにコミットする

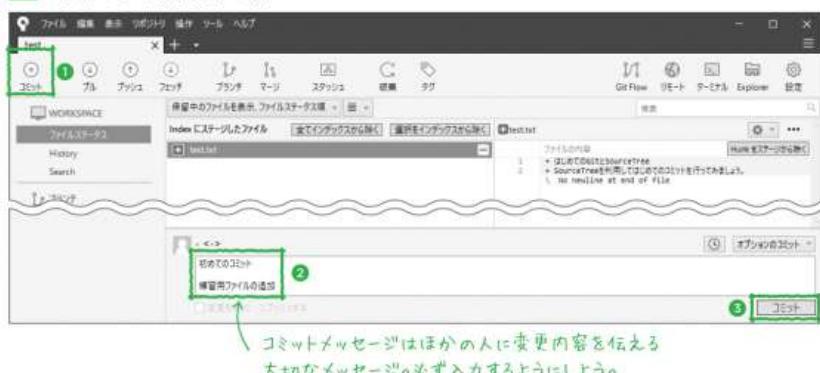
ス テージに移動したら、いよいよコミット（commit）を行います（図5）。コミットするには画面上部のツールバーにある「コミット」ボタン①をクリックします。

すると、下部にコミットメッセージの入力欄が表示されます。ここに変更内容についてのコメントを入力します②。各コミットにはコミットメッセージを付けます。コミットメッセージの書き方についてはP.056「コミットメッセージをわかりやすく書くコツ」で説明しますが、ここでは、コミット

を行うためにはメッセージが必要ということを覚えておきましょう。コミットメッセージを入力したら、画面右下の「コミット」ボタン③をクリックします。

「変更をすぐに-にプッシュする」というチェックボックスがありますが、ここはひとまずチェックなしのままにしておきましょう。リモートリポジトリと連携する際に利用します（→P.060）。

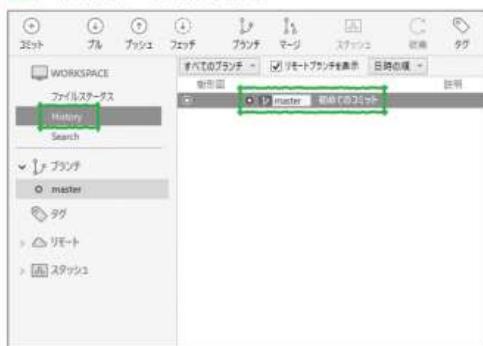
図5 コミットメッセージを入力してコミット



さて、以上でコミットが完了しました。左サイドバーの「ブランチ」→「master」をクリックすると、最初のコミットが表示されるはずです（図6）。これで、初めてのコミットに成功しました。

MEMO addとcommit
CUIのコマンドでは、ステージに移動する操作に「git add」コマンド、コミットする操作に「git commit」コマンドを使います。

図6 コミット履歴にコミットが表示される



▶ 変更内容をコミットする

再度、test.txtファイルの中身を変更してみましょう。ここでは1行追加しました（図7）。

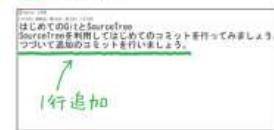
ファイル変更の確認

Sourcetreeを開くと、先ほどの最初のコミットの上に、「コミットされていない変更があります」という表示がなされました（図8）。その部分をクリックすると、下部に先ほど変更したtest.txtが表示されています。また、その右側には変更の内容が赤と緑の色つきで表示されています。

図8 変更があるとその内容が表示される



図7 内容を変更

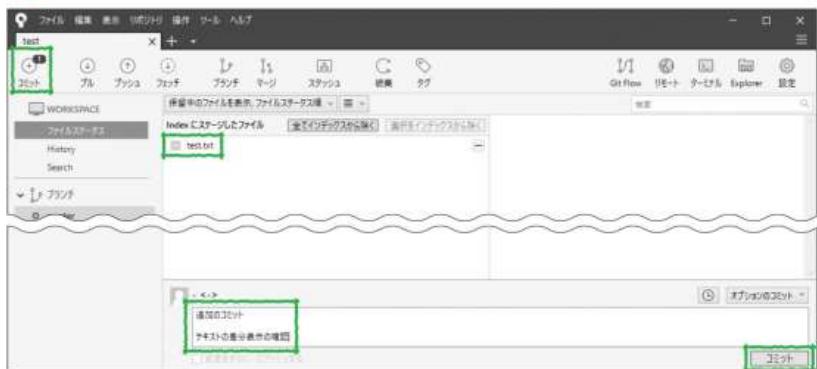


ステージに移動してコミットする

先ほどと同じように、test.txtの右にある「+」ボタンをクリックするなどして、ステージに移動してコミットを行ってみましょう。コミットメッセージを入力したら、コミットをクリックします（図9）。

ざっと急ぎ足で、コミットを行ってみましたが、**基本的に変更→追加→コミット**の繰り返しとなります。何度か、この内容を繰り返して、コミットを行うということに慣れていきましょう。

図9 「内容の変更→ステージに移動→コミットメッセージを入力してコミット」を繰り返す

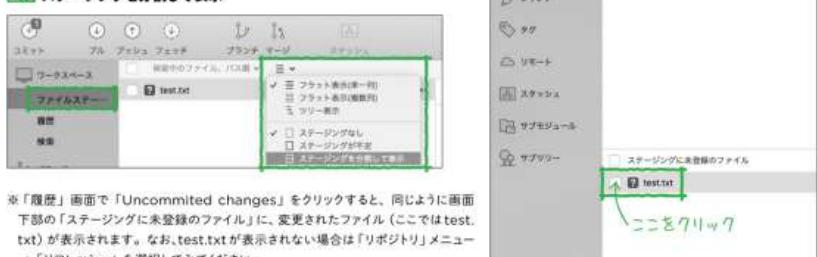


▶ Macの場合

Macの場合は、サイドバーの「ワークスペース」→「ファイルステータス」で、≡のブルダウンメニューから「ステージングを分割して表示」を選択します（図10）。

図11のように表示が切り替わります。「ステージングに未登録のファイル」に新たに追加したファイルtest.txtが表示されるので、ファイル名左側のチェックボックスにチェックを入れます。

図10 ステージングを分割して表示



※「履歴」画面で「Uncommitted changes」をクリックすると、同じように画面下部の「ステージングに未登録のファイル」に、変更されたファイル（ここではtest.txt）が表示されます。なお、test.txtが表示されない場合は「リポジトリ」メニュー→「リフレッシュ」を選択してみてください。

すると、test.txtが画面上部の「ステージ済みのファイル」に移動します①。この状態で、ツールバーの「コミットボタン」②をクリックしてコミットメッセージを入力し③、画面右下の「コミット」ボタン④をクリックします⑤。コミット履歴が表示されない場合は、サイドバーの「ブランチ」にカーソルを重ねて、右側に「表示」と現れたらこれをクリックします⑥。「master」というブランチ名が表示されるのでクリックすると、コミット履歴が表示されます。あるいは、「ワークスペース」→「履歴」を選択してもコミット履歴を見ることができます。

図12 コミットメッセージを入力してコミットする



図13 コミット履歴の表示



▶ コミット間を移動してみる

図12 ミットは時系列のつながりを持っているので、コミットの単位で内容を戻すことができます(P.O12 「Gitにおける作業の流れ」参照)。

実際にそのことを確認してみましょう。ここまで操作でふたつのコミットが行われ、現在test.txtのファイルの中身は図14のようになっています。ここで、コミットをひとつ前のものに戻してみましょう。

図14 test.txtのファイルの状態



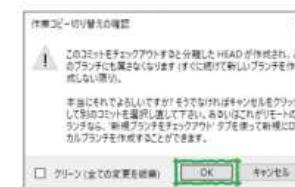
① Sourcetreeのコミット履歴欄で、最初のコミットである「初めてのコミット」をダブルクリックします。



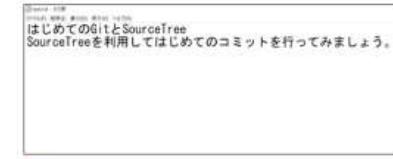
② コミット履歴に「HEAD」という表示がされました。HEADについての詳細な説明はここでは割愛しますが、ひとまずこの場所に移動したと考えましょう。ひとつ前の状態に戻ったのでファイルが戻っているはずです。test.txtを開いてみましょう。



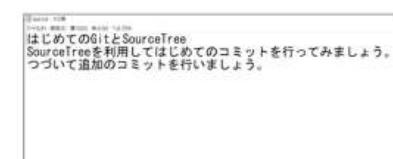
③ 今度は逆に最新のコミットに進んでみましょう。コミット履歴欄で、新しいコミットをダブルクリックします。これでHEADがなくなりました。これは最新のところに移動したと考えましょう。



② 「作業コピーの切り替えの確認」ダイアログが表示されます。「OK」をクリックしましょう。



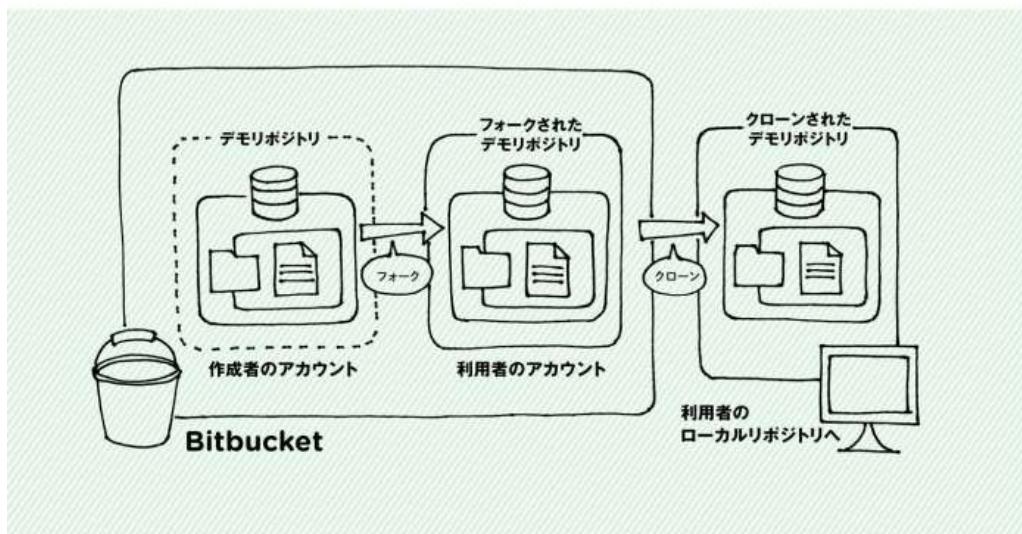
④ ファイルの中身がひとつ前のコミットの状態に戻っていることがわかります。



⑤ test.txtを開くと最新の状態に戻っています。このように簡単な操作で、コミット間を移動することが可能でした。以上、コミットについて急ぎ足でしたが、ひと通りの作業を行いました。コミットについてのより詳しい内容は2章(→P.047)で紹介します。

本書のデモリポジトリを利用する

Bitbucket のアカウントを取得したら、本書のデモリポジトリを利用する環境を整えましょう。
操作は2段階になります。まず、本書のデモリポジトリを自分のアカウントに「フォーク」します。
次に、フォークしたリポジトリを自分のPCに「クローン」(ダウンロード)します。



- 本書のデモリポジトリを自分のアカウントのリポジトリとして複製する（フォーク）
- 自分のアカウントに複製したリポジトリを、自分のPCにダウンロード（クローン）

使用するコマンド

```
$ git clone [クローンするリポジトリURL]
```

▶ 自分のアカウントにデモリポジトリの複製を作成

いよいよ Bitbucket から、今後 Git を学ぶために利用するデモデータを取得します。
リポジトリの URL はこちらです。

<https://bitbucket.org/GitBook-MdN/gitbook-mdn-demo>

このデモリポジトリのファイルをもとに、Chapter 2 では Git の使い方を学んでいきます。

Bitbucket に登録されているリポジトリの利用方法は、そのリポジトリの管理者のポリシーによって異なりますが、一般的に次のようなものと考えられます。

- ・ そのリポジトリのチームとなる（作成者の許可が必要）
- ・ リモートリポジトリの複製（コピー）を自分のPCにダウンロードしてローカルリポジトリとする（クローン）
- ・ リモートリポジトリを複製（コピー）して、自分の Bitbucket アカウントに別のリモートリポジトリとして作成する（フォーク）

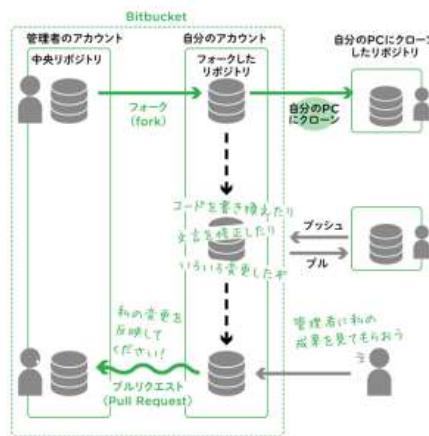
本書のデモリポジトリのように、公開されているリポジトリには管理者が存在します。もちろん、そのリポジトリは本書の読者であるあなた自身のものではありません。そのため、今後そのリポジトリにコミットなどを続けても、基本的に反映することができません。

ただし、フォークすることで、元のデモリポジトリとは別に自身のリポジトリとして新たにデモリポジトリを複製し、継続して利用することが可能になります。

また、実際の運用では、フォークしたあとに変更を加えた場合、その変更を元のリポジトリの管理者に対して、取り入れてもらえるようにリクエストすることが可能です。これは「プルリクエスト」と呼ばれています^{図1}（ただし、本書のデモリポジトリではプルリクエストは受け付けていません。→下記 MEMO 参照）。

現時点では、本書のデモリポジトリをフォークしてから作業を始めると覚えておいてください。

図1 フォークとプルリクエスト



一般にプルリクエストはプロジェクト参加者とディスカッションするなどして採用するかどうか決定されます。

MEMO フォークについて

本書では練習用にフォークしていますが、実際のクレーズドな開発現場ではリポジトリが不用意に分散するためフォークは避け、チーム内のメンバーが個々に自分のPCにクローンして、ローカルリポジトリで作業するという場合もあります。

MEMO 本書のデモリポジトリについて

本書のデモリポジトリは学習用に提供するもので、その運用結果については著者及び出版社はいかなる責任も負いません。また、フォーク、クローンは自由に行えますが、プルリクエストには応答いたしません。あらかじめご了承ください。



① 自分のアカウントでBitbucketにログイン。デモリポジトリのサイト (<https://bitbucket.org/GitBook-MdN/gitbook-mdn-demo>) にアクセスし、左側の+マークをクリックします。



② 開いたバブルメニューの下部にある「このリポジトリをフォークする」をクリックします。



③ ここでは「名前」を「GitBook-MdN-Demo-Fork」、「説明」を「GITの練習用」としました。「アクセスレベル」は練習のために「これは非公開リポジトリです。」をチェック。そのままでは「リポジトリをフォーク」をクリックしてフォークします。



④ 自分のアカウントにデモリポジトリが作成されます。



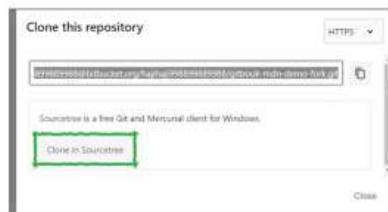
⑤ 「あなたの作業」のリポジトリの一覧にも追加されます。

▶ フォークしたデモリポジトリのクローンを自分のPCに作成する

フ オークができたら、次に自分のPCにデモリポジトリをクローンしましょう。この作業は、Bitbucket上にあるリポジトリをダウンロードするようなイメージで考えてください。



⑥ 先程のフォークしたリポジトリの画面で、右上にある「クローンの作成」をクリックします。



⑦ 表示されるポップアップウィンドウで「Clone in Sourcetree」ボタンをクリックします。



⑧ Sourcetreeが起動し、「Clone」というタブウィンドウ（Macは「新規にクローン」ダイアログ）が開きます。ここで自分のPCのどこにリポジトリを保存するかを指定できます。



⑨ Sourcetreeの画面で、左サイドバーの「ブランチ」→「master」をクリックするとクローンしたリポジトリこれまでのコミットの履歴が閲覧できます。



⑩ また、⑨で保存先として指定したフォルダにデモリポジトリのファイル群が保存されていることも、併せて確認しておきましょう。

1-07

▶ デモリポジトリのファイル

本 書のデモリポジトリのファイルは架空のホテルサイトのトップページとなっています。ファイル構成は図2のようになっています。「index.html」をブラウザで開いて確認してみてください。図3。

Sourcetreeの画面でクローンしたりポジトリを見てみましょう図4。いくつかのコミットがあるのがわかります。このように、フォークした場合には、元のリポジトリにおいて作業してあったコミットがそのまま引き継がれます。これらを見ることで、どういった目的でいつ誰が、どのファイルを変更したのかということがわかります。

たとえば図4で「見出しのデザイン変更」というコミットをクリックすると、①の領域にコミットや変更内容の情報が表示されます。ここでは、megane9988さんが修正したということがわかります。また、②の領域で、修正したファイルはcss/main.cssであり、③で該当箇所は105行目付近であるということもわかります。

図2 デモリポジトリのファイル

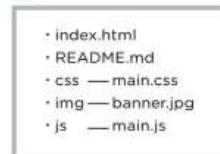


図3 デモリポジトリのindex.html

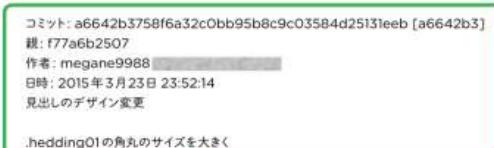
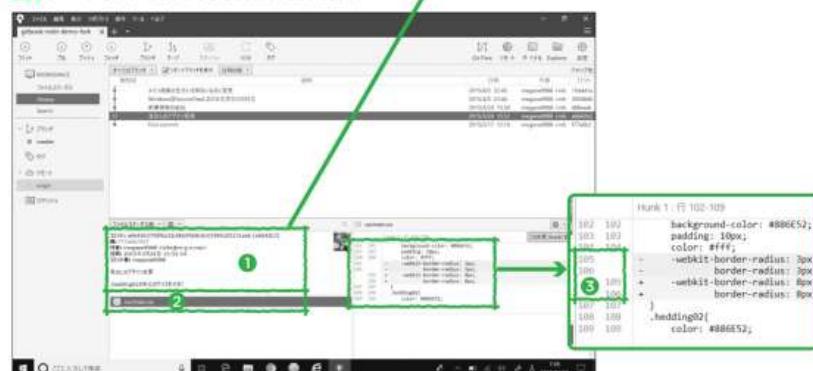


図4 クローンしたデモリポジトリをSourcetreeで開く



Bitbucketの画面を確認する

お使いのPC上のローカルリポジトリの状態はSourcetreeで管理できます。

一方、リモートリポジトリの状態はBitbucketで確認します。Sourcetreeについては、次章で学んでいくとして、ここではBitbucketの画面の見方についてざっと確認しておきましょう。

- 「あなたのページ」で、自分がかかわるリポジトリを一覧で確認できる
- 「リポジトリ」から、個別のリポジトリごとの詳細を確認できる
- 青い左サイドバーにある「+」を押すと、新しいリポジトリの作成やリポジトリのクローンなどを行えるメニューが表示される

▶ あなたの作業に表示される情報

B bitbucketにログインすると表示される「あなたの作業」には、自分のアカウントに関する情報が表示されます^{図1}。「あなたの作業」が表示されていない場合は左側の青い背景部分にあるBitbucketのロゴアイコン、もしくはメニューの「あなたの作業」をクリックしましょう。

① 「あなたの作業」

自分やチームが利用していたり、ウォッチしていたりするリポジトリが一覧表示されます。

② 「プルリクエスト」

フォークしたリポジトリからコミットについて、元のリポジトリへ取り込むことを依頼するのがプルリクエストです。ここでは自分に関係するプルリクエストの一覧が表示され

図1 「あなたの作業」の画面

The screenshot shows the Bitbucket 'Your Work' dashboard. On the left, there's a sidebar with a user icon, a search bar, and navigation links for 'Repositories', 'Pull Requests', and 'Issues'. The main area is titled 'あなたの作業' (Your Work) and lists two repositories: 'Gitbook-MdN-Demo-Fork' and 'Gitbook-MdN-Demo'. The 'Gitbook-MdN-Demo-Fork' repository has a green circle with '2' indicating new activity. The 'Pull Requests' section on the right shows a list of pull requests from the forked repository, each with a green circle containing a number (e.g., 1, 2, 3, 4) and a small icon.

Bitbucket からログアウトしている場合は、「<https://id.atlassian.com/>」からログインできます。

▶ リポジトリの表示

F オークした本書のデモリポジトリを見てみましょう。「あなたの作業」のページでリポジトリのリストにある「Gitbook-MdN-Demo」をクリックすると、リポジトリの情報画面に遷移します^{図2}。

画面左に表示内容を選択するメニュー、画面右に選択したメニューの内容が表示されます。

① 左メニュー

メニュー内にはリポジトリを操作するコマンドなどが表示されています。ソースやコミットの確認、ブランチの作成、プルリクエスト、ブランチ間の比較などが行えます。

② 「ソース」

「ソース」では、リポジトリで管理されているファイルが表示されます^{図3}。index.htmlをクリックしてみましょう。

図2 リポジトリの「概要」画面

The screenshot shows the 'Overview' page for the 'Gitbook-MdN-Demo-Fork' repository. The left sidebar shows the repository structure: 'Gitbook-MdN-Demo-Fork' (selected), 'Issues', 'Commits', 'Branches', 'Pull Requests', 'Pipelines', 'Downloads', 'Boards', and 'Settings'. The right panel displays basic repository information: 'Name: Gitbook-MdN-Demo-Fork', 'Owner: GitBook-MdN-Demo', 'Created: 2015-03-24', 'Last updated: 2015-04-05', and a brief description. Below this is a file list table:

Name	Size	Last Commit	Message
git clone		2015-03-24	Bitbucketのデザインを変更しました。
index.html		2015-04-05	メイン画像の色合いを明るいものに変更。他の魅力をよりよく見せるためです。
js		2015-02-17	First commit

図3 リポジトリの「ソース」画面

The screenshot shows the 'Source' page for the 'index.html' file in the 'Gitbook-MdN-Demo-Fork' repository. The left sidebar shows the repository structure with 'index.html' selected. The right panel displays the file content:

```

<!DOCTYPE html>
<html>
<head>
    <title>GitBook-MdN-Demo</title>
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <link rel="stylesheet" href="style.css" />
</head>
<body>
    <h1>GitBook-MdN-Demo</h1>
    <p>This is a test repository for the GitBook-MdN-Demo project. It contains a single file, index.html, which is a simple HTML page with some CSS styling. The repository was created on March 24, 2015, and has been updated once since then. The file 'index.html' was last modified on April 5, 2015, at 10:45 UTC. The file size is 237 kB and it contains the following content:</p>
<pre><code><!-- This is a test repository for the GitBook-MdN-Demo project. It contains a single file, index.html, which is a simple HTML page with some CSS styling. The repository was created on March 24, 2015, and has been updated once since then. The file 'index.html' was last modified on April 5, 2015, at 10:45 UTC. The file size is 237 kB and it contains the following content:</code></pre>
</body>
</html>

```

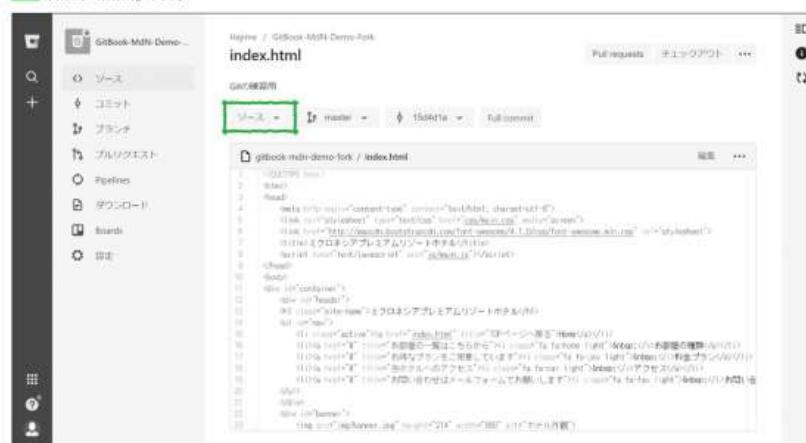
A green box highlights the file content, and a green arrow points to the word 'クリップ' (Clip) in the bottom right corner.

index.html の内容が表示されます **図4**。「ソース」のプルダウンをクリックすると「差分」「履歴」を選択できます。これらを利用することで、変更内容を具体的にチェックできます。例えば、「差分」をクリックすると、直近の変更についての差分が表示されます **図5**。比較したいコミットを選択することで、差分を比較することもできます。

「履歴」をクリックすると、コミットの一覧が表示されます。

「コミット」 **図6** では、リポジトリにおけるコミットの一覧が表示されます。コミットID（番号）をクリックすると、変更されたファイル名と変更箇所などが表示されます **図7**。

図4 「index.html」の表示



「プランチ」はすべてのプランチを一覧表示します。初期状態ではプランチが1つしかない状態のため表示されません。

「ブルリクエスト」は、リポジトリへのブルリクエストが一覧表示されます。

「ダウンロード」は、リポジトリの管理者が提供する、リポジトリのファイルの最新版や、タグ、プランチなど、さまざまなものダウンロードできます。

「設定」はリポジトリの管理画面です **図8**。

更されたファイル名と変更箇所などが表示されます **図7**。

図6 「コミット」画面

GitBook-MDN-Demo-Fork	
ソース	コミット
+	すべてのプランチ
ソース	プランチ
ブルリクエスト	ブルリクエスト
Pipelines	Pipelines
ダウンロード	ダウンロード
Boards	Boards
設定	設定

リスト内に表示される各コミットには、作成者、日付、ビルド数が記載されています。各コミットをクリックすると、その詳細が表示される **図7**。

図7 コミットIDをクリックすると変更内容が表示される



図5 「差分」の表示



図8 「設定」画面



なお、画面左下の人型のマークをクリックすると図9、パスワードの変更や、Bitbucketのプラン変更、通知メールの受信設定、SSH鍵の登録など、自身のアカウントに関する情報の変更・管理を行えます図10。退会する場合もこちらから行います。



図10 「アカウント設定」画面

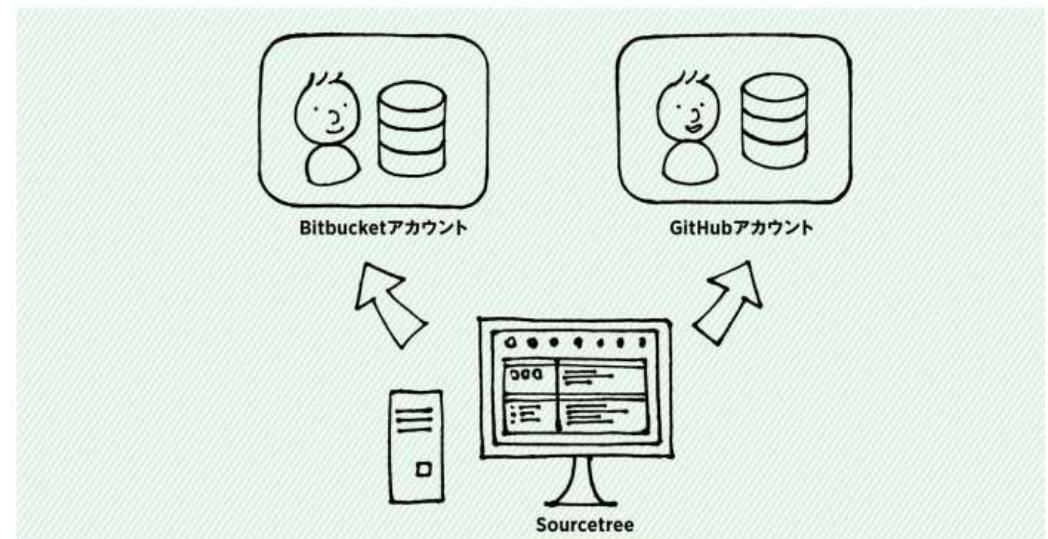


1-08

GitHubアカウントを連携する

制作現場ではリモートリポジトリのホスティングサービスとして、BitbucketではなくGitHubが使われるケースも多いでしょう。

本章の最後に、SourcetreeとGitHubのアカウントを連携させる手順を解説します。



- GitHubのアカウントを取得しておく
- SourcetreeにGitHubのアカウントを追加する
- GitHubアカウントの認証作業を行う

使用するコマンド

```
$ git remote add
```

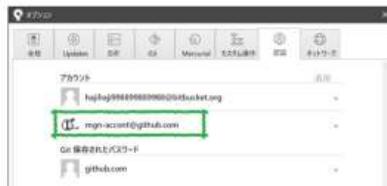
▶ GitHubアカウントの追加

Sourcetree では GitHub など、Bitbucket 以外のホスティングサービスを連携して操作することも可能です。GitHub のアカウントの取得方法は割愛しますので、公式サイト (<https://github.com/>) から予め GitHub のアカウント取得した上で、作業を行ってください。

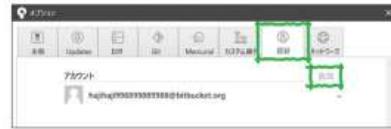


② 「ホスティングアカウントを設定」で、「Host」にある「ホスティングサービス」に「GitHub」を選択します。「Credentials」にある「認証」は「OAuth」と「Basic」の2種類から選択でき、OAuth で認証を行う場合は、「OAuth トークンを読み込み」をクリックします。

ブラウザが開き、GitHub の OAuth 認証用のログインページが表示されますので、GitHub に登録しているユーザー名もしくはメールアドレスとパスワードを入力して、「Sign in」を押します。



④ 認証が成功したら、OK を押してダイアログを閉じます。Sourcetree の「認証」ダイアログを確認すると、アカウントの一覧に GitHub のアカウントが表示されているはずです。



① Windows では Sourcetree のメニューから「ツール」→「オプション」を選びます。「オプション」ダイアログで「認証」タブを選択し、「追加」をクリックします。



③ Basic で認証を行う場合はユーザー名を入力したあと「パスワードを再読み込み」をクリックして、表示される画面でパスワードを入力します。



⑤ Mac の場合は、リポジトリブラウザの右上にある歯車マークをクリックし（右図）、表示される「アカウント」のダイアログで「追加」を選びます（上図）。

⑥ 表示されるダイアログで、ホストに「GitHub」を選びます。次に「アカウントを接続」をクリックすると、ブラウザが開き、GitHub の OAuth 認証用のログインページが表示されます。認証の手順は Windowsと同じです。認証が完了すると、新たに追加した GitHub アカウントが一覧に表示されます。

2

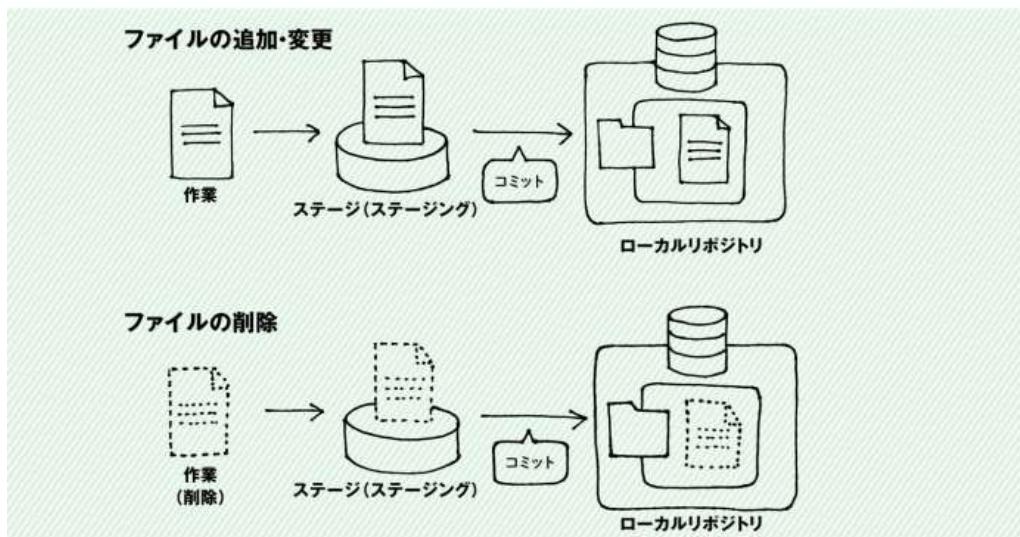
Chapter 2

Gitの基本的な使い方

バージョン管理システム「Git」と、コマンドラインを使わずにデスクトップから Git を操作できるクライアントソフト「Sourcetree」を組み合わせた、基本的な使い方を解説していきます。

変更をコミットする

それでは、いよいよ Sourcetree を操作して、Git のバージョン管理がどのようなものか学んでいきましょう。本節では、まず最初に覚えるべき、ファイルの変更履歴を保存する「コミット」という操作を取り上げます。ファイルの新規作成、削除、ファイルの内容変更など、さまざまな変更はすべてコミットの対象となります。



- コミットは「ステージに追加→コミット」の2段階
- コミットはひとつの作業ごとに使う
- ファイルの新規追加・削除もコミットの対象

使用するコマンド

```
$ git add [ステージに登録するファイル]
$ git add -p [ファイル]
$ git commit -m [メッセージ]
$ git rm [削除するファイル]
```

使用するSourcetreeの機能

コミット

▶ コミットとは？

実

際に開発が始まると、新しいファイルを作成したり、既存のファイルの内容を書き替えたり、不要なファイルを削除したり、ファイル名を変更したり……さまざまな編集作業が行われていきます。そうした変更にメッセージをつけてリポジトリに登録する操作を「コミット」(commit)といいます。

「1-05 コミットする手順を覚える」(P.027)では、まずはコミットしてみることを目的に、操作方法を駆け足で説明しましたが、ここでもう少し詳しく見ていきましょう。

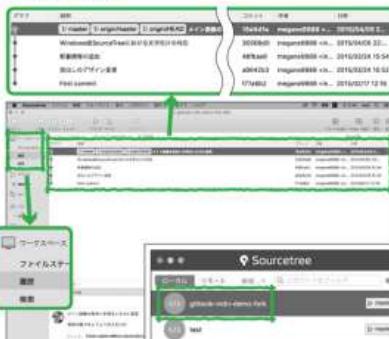
リポジトリ内では、コミットを行うことで、**変更の履歴**が時系列につながった形で記録されていきます。利用者は、これまでに、どのファイルのどの箇所が変更されたのか、またその変更是いつ誰が行ったのかなどを知ることができます。また、コミット単位で、過去にさかのぼってファイルを取得したり、現状のファイルをある時点まで戻したりすることもできます。このようなコミットの履歴は、Sourcetree では左サイドバーの「History」(Macは「履歴」)を選択して表示できます【図1】[図2]。

図1 Sourcetreeでのコミット履歴の表示



「1-06 本書のデモリポジトリを利用する」(P.034)でクローンした本書のデモリポジトリには、すでにいくつかのコミットがあります。

図2 Mac版の場合



Mac版ではサイドバーの「履歴」から確認できます。なお、メインウィンドウが開いていない場合は、リポジトリブラウザで目的のリポジトリ名をダブルクリックします。

▶ コミットは「ステージに追加→コミット」の2段階で行う

ま

ず、コミットはステージに追加→コミットという2段階で行うということを覚えておきましょう【図3】。

どうして2段階なのかは後述しますので、早速実際にコミットを行ってみましょう。

図3 コミットまでの流れ



ファイルの変更には、ファイルの追加や編集のほかに、削除も含まれます。ファイルに変更がなければ、コミットする必要はありません。

コミットしたいファイルを選択し、ステージに追加します。その際、変更内容がわかるようなメッセージも付けるようにします。コミットメッセージについては後述します。

- 1 サンプルには「J-06 本書のデモリポジトリを利用する」(P.034)でクローン済みのローカルリポジトリを利用します。index.html の title の「ミクロネシア」をハイアンに変更してみます。

```
<title>ミクロネシアプレミアムリゾートホテル</title>
```



```
<title>ハイアンプレミアムリゾートホテル</title>
```

- 2 「History」(Mac は「履歴」) に「コミットされていない変更があります」(Mac は「Uncommitted changes」と表示されます)。これをクリックすると、下部に「作業ツリーのファイル」(Mac は「ステージングに未登録のファイル」として「index.html」が表示されます)。

Sourcetree が反応を示さない場合は「表示」メニューの「更新」(Mac は「リポジトリ」メニューの「リフレッシュ」) を選択してみてください。

- 3 右カラムには、ソースコードの追加(緑+)と削除(赤-)の状態が、ひとつのコミットと比較されて表示されます。

- 4 「index.html」を選択してから、「全てインデックスに追加」などのボタンをクリックします(Mac は「index.html」左側のチェックボックスにチェックを入れる)。「index.html」が上側の「Index」にステージしたファイル(Mac は「ステージング済みのファイル」)の欄に移動します。これでステージに追加されました。

- 5 画面上部ツールバーの「コミット」ボタンをクリックしましょう。

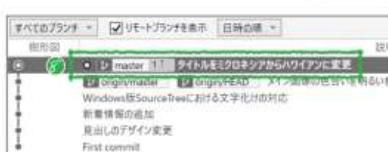
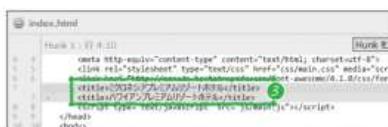
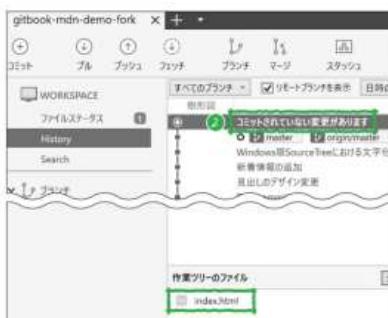


- 6 画面下側のテキスト入力欄にコミットメッセージを入力します。ここでは「タイトルをミクロネシアからハイアンに変更」と入力しました。

- 7 画面右下の「コミット」ボタンをクリックします。



- 8コミットが完了し、1つグラフ(コミットツリー)が伸びたことが確認できます。右下のカラムでコミットの内容を閲覧することもできます。



MEMO 1コミット先行

前ページの手順②の図を見ると、行ったコミットに「1↑」(Mac は「1コミット先行」と表示されています)と表示されています。これはクローン元である Bitbucket に対して 1コミット先行 したということを示しています。そのすぐ下に、origin/master と origin/HEAD という

表示がありますが、こちらは Bitbucket 上のブランチの最新の位置を示しています。このように、Sourcetree では、ローカル(自分の PC)とリモート(今回 Bitbucket)のそれぞれのコミットの状態を簡単に一覧してみることができます。

▶ コミットはひとつの作業ごとに行う

フ ァイルを編集していく過程で、どのタイミングでコミットするのが正しいのか、さまざまな考え方がありますが、基本的にはひとつの作業を終えるごとにコミットすることが望ましいでしょう(図4)。これは、コミットごとの作業が明確になるとともに、もしコミットした内容が間違っていた場合にも修正を行いやすくなるという意味があります。コミット単位でしかファイルの復元ができないため、多くの作業を同時にコミットすると、それらをすべて戻さないといけないケースも発生してしまいます。

特に、同時に複数の人が作業している場合には、より1作業ごとにコミットし、変更情報を共有していくことを心がけましょう。プロジェクト全体の実作業の進捗具合が明確になります。また、メンバー間での作業の重複や、衝突を避けることにもつながります。

もしプロジェクトにおいて、課題管理ツールなどを利用している場合は、その1課題ごとにコミットを行うということを目安にするとよいでしょう。

図4 コミットは一作業ごとに



MEMO ステージの役割

変更したファイルをコミットする前にステージに追加するという作業は不要にも感られます。では、なぜこの作業を行なうように設計されているのでしょうか。これには理由があります。前述した通り、コミットは一作業ごとに行なうのが基本ですが、作業を集中して進めた結果、複数の作業を一度に進めてしまうこともあります。そんなときに、変更されたファイルから1作業ごとにステージにピックアップすることで、1コミットで1作業を実現することができます。

MEMO コミットメッセージの内容

コミットには「コミットメッセージ」と呼ばれる、コミットの説明文の記載が必要です。端的に誰が見てもわかりやすいコミットメッセージを記載するように心がけましょう。1コミットで1作業とも目的が重なりますが、複数の作業を一度に行って「いくつかの修正に対応」などとメッセージをつけてまとめてコミットなどというやり方では、ほかのメンバーは何をどう変更したのかさっぱりわかりません。せっかくの Git によるバージョン管理の利点が失われてしまいます。

▶ ファイルを新規に追加する

次 ファイル名を `access.html` に変更してみましょう
図5 「1-O5 コミットする手順を覚える」(P.027) で見たように、リポジトリ内にファイルが新たに追加されると、作業ツリー上にファイルが表示されます。その場合、アイコンは■マークで表示されます。これまでの管理履歴にはない不明なファイルという意味です**図6**。

ここから先は P.050 の手順と同様です。ステージに、

図5 Index.htmlのコピーして名前をaccess.htmlに変更



図6 新たに追加されたファイルには■マークがつく

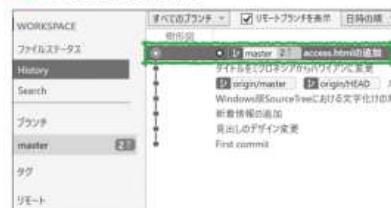
新規追加したファイル `access.html` を移動し、コミットを行います。ここでは「`access.html` の追加」というコミットメッセージでコミットを行いました**図7**。結果として、樹形図が1つ進んでいるのがわかります**図8**。

新規に何かファイルもしくはファイル群を追加する場合は、この作業を行います。ほかのファイルも増やしてみたりして、コミットを何度か試し、慣れておきましょう。

図7 新たに追加されたファイルには■マークがつく



図8 コミット履歴に反映される



▶ ファイルを削除する

ファイルの追加と同様に、ファイルを削除したときもコミットを行います。先ほど追加し、コミットした `access.html` をエクスプローラー上 (MacではFinder上) で削除してみましょう。`access.html` に■マークが表示されます。これはファイルが削除されたということを示しています**図9**。

この変更をこれまでと同様に、ステージに移動し、メッセージをつけてコミットしてみましょう**図10**。

削除してなくなったファイルをステージに移動するのは最初は慣れないかもしれません。これはファイルを移動しているのではなく、**ファイルに対して行った変更を記録する**ために行っているということを理解しておきましょう。

図9 ファイルの削除も検知される



ファイルが削除される場合もソースの一部が削除される場合も、削除の場合は赤い表示になります。こちらも結果として樹形図がひとつ進みます。このようにファイルの内容の編集、ファイルの追加／削除を問わず、コミットごとに樹形図がひとつずつ進むということを理解しましょう。

図10 ファイルを削除した場合もコミットする



▶ 複数の変更ファイルを同時にコミットする

次 について見てみましょう。ここではデモリポジトリ内の `index.html` と `css` フォルダの `main.css`両方で複数の変更を加えます。実は、両ファイルともに内容にタイプミスがあります。コンテンツ内の見出しのクラス設定が「`.hedding`」となっていますが、これは「`.heading`」の

間違いです。これらを修正しましょう。

`index.html` では 27 行目、34 行目付近、`main.css` では 101 行目、108 行目付近にある `hedding` を `heading` に修正します**図11**。

SourceTree で見てみると 2 つのファイルが変更されたものとして検知されているのがわかります**図12**。

図11 heddingをheadingに修正

○ index.html 27 行目付近

```
<h1 class="hedding01">当ホテルからのご案内</h1>
```


○ index.html 34 行目付近

```
<h2 class="hedding02">オンライン宿泊予約特典</h2>
```

○ main.css 101 行目、108 行目付近

```
.hedding01{  
background-color: #886E52;  
border-radius: 8px;  
}  
.hedding02{  
color: #886E52;  
}
```

図12 index.htmlとmain.cssの変更が検知される



このように、複数のファイルを変更した場合も、すべてが検知され表示されます。

修正したファイルをステージに移動し、コミットします。一括でファイルを登録したい場合は、「全てインデックスに追加」ボタンをクリックしましょう（Macは「ステージング

に未登録のファイル」左側のチェックボックスにチェックを入れる）。あとはこれまで同様にコミットします（図13）。

以上のように、複数のファイルが変更された場合に、それらをまとめて1つのコミットにできることを覚えましょう。

図13 ふたつの変更をまとめてコミット



Windows

Mac

▶ 複数の変更箇所を同時にコミットする

先 の例では、index.html、main.cssをそれぞれ複数箇所変更して、ファイル単位で一括してコミットしましたが、ファイル単位ではなく、変更箇所単位でコミットすることもできます。

まず、index.htmlに複数の変更を加えましょう。29行目付近のp要素の記述「特別料金」に「(20%OFF)」、40行目付近にあるli要素の下に「全室wifi完備」と追記します（図14）。

SourceTreeに戻ると、index.htmlのファイルの2箇

所が変更箇所として検知されたのがわかります（図15）。

このようにひとつのファイル内でも行が異なる箇所を変更した場合、それぞれ別個の編集として検知されます（編集箇所一つひとつを「Hunk」と呼びます）。

複数の変更をまとめてコミットする場合は、これまでと同じく、「作業ツリーのファイル」をステージに移動すればOKです。複数のHunkをまとめてコミットすることができます。

図14 index.htmlの内容を変更

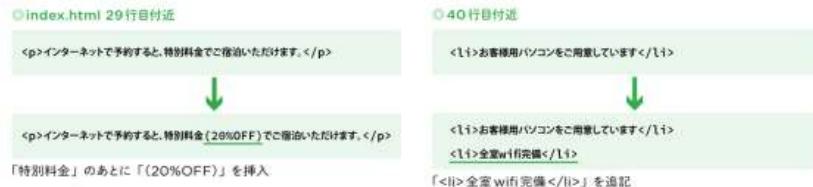


図15 2カ所の変更が検知される

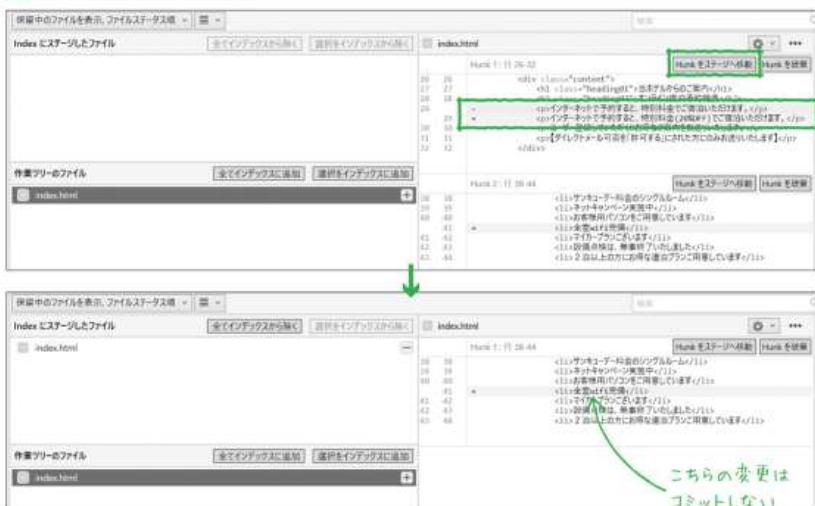


▶ 複数の変更を個別にコミットする

前述の「複数の変更箇所を同時にコミットする」では複数のファイルの内容を変更し、それらをまとめて一度にコミットしましたが、「選択をインデックスに追加」ボタンや各ファイル右側の「+」ボタンをクリックすれば（Macは各ファイル左側のチェックボックスにチェックを入れる）、ファイル単位で個別にコミットすることも可能です。

また、前ページの「複数の変更箇所を同時にコミットする」のようにひとつのファイル内にふたつ以上の変更箇所がある場合は、コミットしたい変更箇所の「Hunkをステージへ移動」ボタン（Macは「Hunkをステージングに移動」ボタン）をクリックすることで、該当の変更箇所のみをステージに追加することができます（図16）。

図16 変更箇所を個別にステージに追加



右側のプレビューでコミットしたい変更箇所を判断して、「Hunkをステージへ移動」ボタンをクリックします。その部分のみステージに追加されるのでコミットします。なお、残った変更はそのままなので、コミット履歴には引き続き「コミットされていない変更があります」（Macでは「Uncommitted changes.」）と表示されます。もし変更を破棄したい場合は「2-04 間違って操作してしまったら？」（P.066）を参照してください。

2-02

▶コミットメッセージをわかりやすく書くコツ

以上、いろいろなコミット方法を学んできましたが、コミットするたびに記載するコミットメッセージについても少しふれておきましょう。

メッセージ入力欄の1行目に書いたメッセージは、そのままコミットの一覧に表示されます。ここにはコミットのタイトルや概要を記入しましょう。さらに詳しく書き添えたい場合は、一般的には2行目を空白とし、3行目から詳細を書くようにします(図17)。

コミットメッセージは、端的で誰が見てもわかるようなものにする、というのが大原則です。わかりやすいコミットメッセージにするためには、まず1コミット1課題を基本とし

て、一言でまとめられる作業程度でコミットを行っておく必要がります。そのうえで、具体的に何を変更したのかを記すようにしましょう。

下図にわかりやすいメッセージと、わかりにくいメッセージについてまとめましたので参考にしてください(図18・図19)。慣れてきたら、今度はどんな目的で変更を行ったのかを書くようにします。そうすることでよりわかりやすいコミットになります。

特に、チームで開発している場合は、コミットの方針、メッセージの書き方についてよく相談しておきましょう。

図17 コミットメッセージの書き方

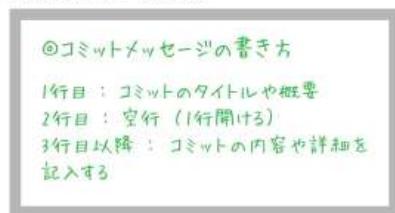


図18 わかりやすいコミットメッセージの例

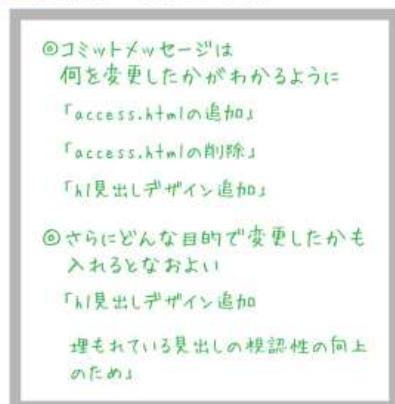
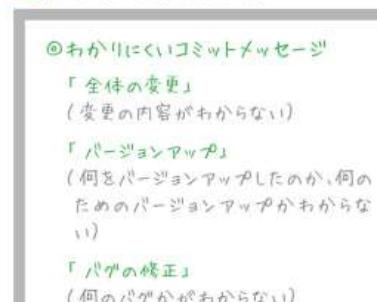


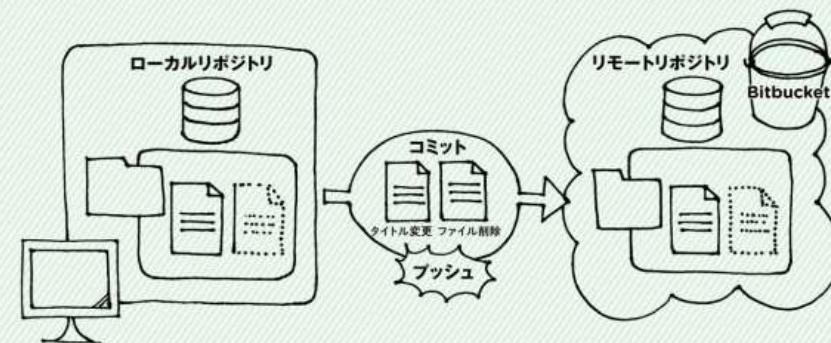
図19 わかりにくいコミットメッセージの例



Bitbucketにプッシュしてみよう

Chapter 1では本書のデモリポジトリをフォークしたものを手元のローカル環境にクローンしました。

前節のコミット操作は、このクローンしたローカルのリポジトリに対して行ってきたわけです。ここでは、ローカルのリポジトリに加えた変更を、Bitbucket上のリモートリポジトリに反映させる方法を説明します。



- 変更をリモートリポジトリに反映するのが「プッシュ」
- リモートリポジトリから変更情報を取得してマージする「プル」
- Bitbucketの「コミット」で変更履歴を確認

使用するコマンド

```
$ git push [リモートリポジトリ] [プッシュするブランチ]  
$ git pull [リモートリポジトリ] [プルするブランチ]
```

使用するSourceTreeの機能



▶ ローカルリポジトリの変更をリモートリポジトリに反映させる

前 節では、自分のPCにあるローカルリポジトリ上で、ファイルの追加・削除、ファイルの内容の変更などをSourcetreeを利用して、コミット履歴に残してきました。このローカルリポジトリは、Bitbucketにある本書のデモリポジトリを、自分のアカウントにフォークしたもの、さらに手元のPCにクローンしたリポジトリです(P.034「1-06 本書のデモリポジトリを利用する」)。

ここで、これまで行ってきた変更を、Bitbucketのフォ

ークしたリポジトリ(リモートリポジトリ)に反映させてみましょう。このように変更をほかのリポジトリ(多くはリモートリポジトリ)に反映する操作を「プッシュ(push)」といいます。

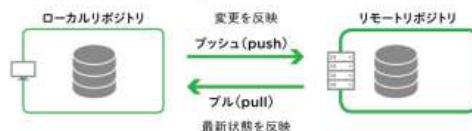
なお、逆にリモートリポジトリの最新の状態を、手元のローカルリポジトリに反映させる作業を「プル(pull)」といいます。

▶ プッシュ(Push)とプル(Pull)

前 述したようにローカルリポジトリからリモートリポジトリへ、変更履歴をアップロードするときには「**プッシュ(push)**」を利用します。現在のローカルリポジトリの状態よりもリモートリポジトリのコミットが進んでいた場合は、それらがローカルリポジトリに反映されます。ファイルのダウンロードに近いイメージです。ローカルリポジトリの状態がリモートリポジトリよりも進んでいたり、同じ場合には何も反映されません(図1)。

一方、リモートリポジトリから、ローカルリポジトリへ変更履歴を取得するときには「**プル(pull)**」を利用します。現在のローカルリポジトリの状態よりもリモートリポジトリのコミットが進んでいた場合は、それらがローカルリポジトリに反映されます。ファイルのアップロードに近いイメージです。ローカルリポジトリの状態がリモートリポジトリよりも遅れていたり、2つが同じ場合には何も反映されません。

図1 プッシュ(push)とプル(pull)



MEMO フェッチ(fetch)とマージ(merge)

ローカルリポジトリでは、リモートリポジトリの状態を最後に取得したときの情報を保持します。ローカルでの作業と、リモートリポジトリの状態との差異は、この情報で計られます(次ページ「originとmaster」参照)。しかし、実際のリモートリポジトリのほうではほかの人の作業などが反映されて、状態が変更されているかもしれません。そこで、リモートリポジトリの最新の状態を取得する作業が必要となります。これが「**フェッチ(fetch)**」です。

ただし、フェッチはリモートリポジトリに関する情報を更新するだけで、ローカルの状態をそれに合わせるわけではありません。ローカルの状態を、取得した最新のリモートリポジトリの状態に合わせるには「**マージ(merge)**」という操作が必要になります。実は「**プル(pull)**」は、このフェッチとマージを合わせた操作になっています。詳しくは「リモートブランチの同期(フェッチ)」(P.144)を参照してください。

▶ originとmaster

ここまで進めてきたSourcetreeのコミット履歴を見てみましょう。「origin/master」と「master 5↑(5コミット先行)」という箇所が見られます(図2)。

「origin/master」の「origin」とは、リモートリポジトリの場所(URL)の別名です。本書ではBitbucketのリポジトリのことです。「master」とはリモートリポジトリのブランチの名前です。いずれもGitのデフォルト設定でつけられた名前です。

一方、「master 5↑」の「master」はローカルリポジトリのブランチです(これもGitのデフォルト設定名です)。

実は前節のコミット操作はローカルのmasterに対して行ってきました。そのため、クローンした時点では同じ状態にあったorigin/masterブランチとローカルのmasterブランチですが、その後のローカルのmasterブランチにおけるコミット操作の結果、それが生じてしまつたわけです。

「master 5↑」という表示はそのことを示しています。origin/masterつまりBitbucket上のmasterブランチに比べて、ローカルのmasterブランチが5コミット分先行していますよ、ということです。

図2 「origin/master」ブランチと「master」ブランチ



▶ Bitbucketのリモートリポジトリにプッシュ(push)する

先 行しているコミットをリモートリポジトリに反映させるために「**プッシュ(push)**」を行ってみましょう。

Sourcetree上部のアイコンメニューにある「**プッシュボタン**」をクリックすると、プッシュの設定を行うダイアログが開きます(図3)。

「プッシュ先」のリポジトリは「origin」、「プッシュするブランチ」は「master」となっていますので、そのまま「**プッシュ**」をクリックします。あっけないですが、これでプッシュは完了です。

コミット履歴を見てみると「origin/master」「origin/HEAD」となっているところがコミットの一一番新しいローカルリポジトリの「master」と描ったことがわかります(図4)。

つまり、ローカルリポジトリでのコミットが問題なくリモートリポジトリに反映され、ローカルリポジトリと同じところまでリモートリポジトリが進んだということがわかります。

図3 プッシュの設定

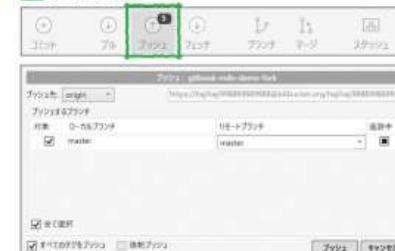


図4 ローカルリポジトリとリモートリポジトリのコミットが揃う



▶ Bitbucket上で確認

「プッシュ(push)」した内容が正しくリモートリポジトリに反映されたかどうか、Bitbucket上の表示とSourcetreeを見比べながら確認してみましょう。

ブラウザでBitbucketにログインして、フォークしたりリポジトリのページを表示し、左のメニューより「コミット」をクリックします。画面の右側に反映されたコミットの一覧が表示されます^{図5}。Sourcetreeのコミット一覧を見

図5 Bitbucket上のリモートリポジトリ



図6 Sourcetree上のローカルリポジトリ

比べると、同じ数と内容とコミットが確認できます^{図6}。正しく反映されたようです。

Bitbucket上でもコミットのID番号をクリックすると、変更したファイルとその変更箇所が表示されます^{図7}。Sourcetreeと同様に追加された内容は緑、削除された内容は赤で示されていることが確認できます。

図7 Bitbucket上のリモートリポジトリのコミットを確認



▶ コミットと同時にプッシュする

Sourcetreeではコミット時に、コミットメッセージ欄の下の「変更をすぐに~にプッシュする」(Macでは「コミットをただちに~プッシュする」)をチェックしておくことで、コミットと一緒にプッシュすることができます^{図8}。こまめにプッシュすることで、ほかの作業者とのコンフリクト(「3-04 コンフリクトはなぜ起こる?」P.090参照)を避けられることも可能になります。特にプロジェクトでのルールがなければ、コミットと一緒にプッシュを行いましょう。

図8 コミットと一緒にプッシュ



「origin/master」の部分にはプッシュ先のブランチ名が入ります。

▶ Bitbucketのリモートリポジトリからプル(Pull)する

ひとりで作業している場合は、自分以外にリモートリポジトリを操作する人はいませんので、リモートリポジトリからプル(pull)する必要はないかもしれません。

しかし、複数のメンバーがかかるるプロジェクトでは、ほかの人がプッシュしてリモートリポジトリのコミットを進めることがあるでしょう。そこで、リモートリポジトリをプル(pull)^{図9}して、自分のローカルリポジトリを最新の状態に更新する必要があります。

プルを行うにはツールバーの「プル」ボタンをクリックす

図9 「プル」ボタン



図10 プルの設定ダイアログ



ると、プルの設定を行うダイアログが表示されます。「次のリモートからプル」「プルするリモートブランチ」の設定は、ブランチが複数に分岐している場合は必要ですが、ここではまだブランチは分岐していないので、初期設定のままでよいでしょう^{図10}。

「OK」をクリックするとプルが行われます。リモートリポジトリ側でローカルリポジトリのコミットより先のコミットがいる場合は取り込まれます。



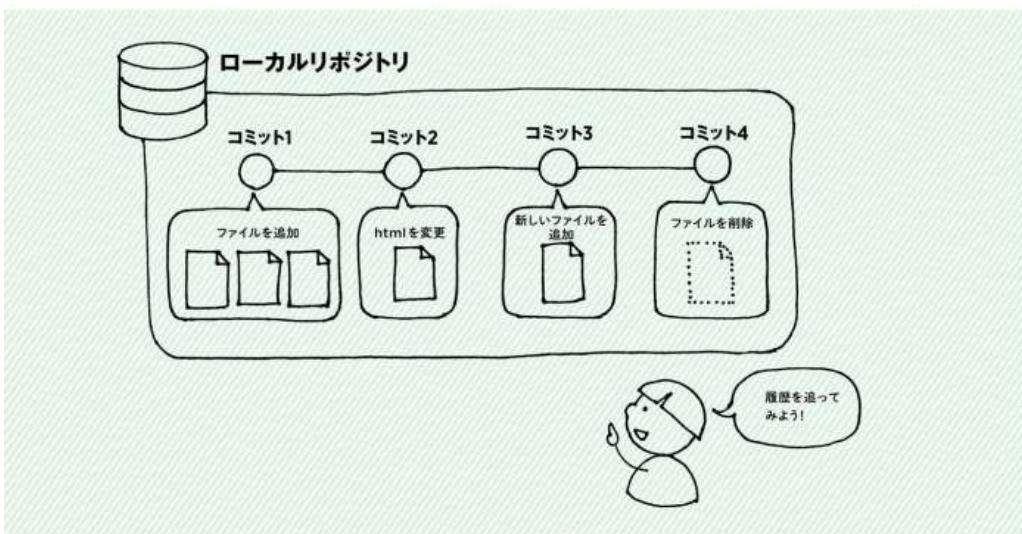
MEMO コンフリクト

複数のメンバーがかかるるリポジトリでは、お互いに同じファイルの同じ場所を修正してプッシュしようと、「コンフリクト」が発生します。また、プルの場合も、コミット情報のマージ時に、自分のコミットとぶつかるのでコンフリクトが発生します。このような場合はプッシュもプルもできなくなります。コンフリクトしたときの解決策、またはコンフリクトの回避策については「3-04 コンフリクトはなぜ起こる?」(P.090)で説明しています。

コミットログ(履歴)を見てみよう

これまで行ってきたコミットについてのログ(履歴)を見ることで、作業を振り返りましょう。

Sourcetreeでは過去のコミットの一覧を見ることができます。また、コミットを選択すればコミットした人、変更日時、コミット時につけたメッセージ、変更箇所などが一目瞭然です。



- Sourcetreeでは選択したブランチのコミット履歴が表示できる
- Bitbucketでモリモートリポジトリのコミット履歴を確認できる

使用するコマンド

```
$ git log
```

Sourcetreeでの履歴の確認方法

① インペーでブランチ(ここでは「master」)を選択すると②、画面の右側に、選択したブランチのコミット履歴の一覧が表示されます③(図1 図2)。

変更履歴には、これまで本書で行ってきたコミットよりもさらに前のものがありますが、これは事前に著者がコミットしていた履歴です。このようにGitを利用してフォーク

したリポジトリでは、フォークする前に行われたコミットについてもその履歴を閲覧することができます。

コミットを選択すると、画面の下部に、コミットID、コミットした人、コミット日時などの情報が表示されます④。また、その右側には変更箇所がdiff形式(差分を示す形式)で表示されます⑤。

図1 コミット一覧(Windows)



コミット情報の「親」は直前の参照しているコミットを指しています。

図2 コミット一覧(Mac)



変更箇所を示すエリアでは、赤色で表示されているのが削除されたコード（マイナス記号「-」の行）、緑色は新たに追加されたコード（プラス記号「+」の行）です（図3）。

コミットメッセージとソースコードの変更箇所を確認することで、どのファイルをどういった目的で編集したかということが把握できます。

なお、サンプルのリポジトリは今のところ1つのブランチしかないので、コミット履歴左端のブランチの樹形図（Macではグラフ）は一直線に伸びているだけですが、ブランチが増えた場合には、枝分かれするような図になります（図4）。今後チームで開発を続ける際に、同じリポジトリ内で複数の人々が同時に作業を進めていく場合、お互いの作業を打ち消してしまわないように、ブランチをそれぞれ分けて作業を進めていく形となります。結果として、このように多くの枝分かれが進んでいくことでしょう。

図3 変更前と後の状態が表示される

```
Hunk 1: 行 4-10
4 4 <meta http-equiv="content-type" content="text/html; charset=UTF-8">
5 5 <link rel="stylesheet" type="text/css" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css" />
6 6 <title>マイクロアソシエイティムリゾートホテル</title>
7 7 <script type="text/javascript" src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/js/bootstrap.min.js">
8 8 </head>
9 9 <body>
10 10 <div class="container">
```

図4 ブランチが枝分かれしている例



▶ Bitbucketでの履歴の見方

これまでSourcetreeでのコミットログの見方を説明してきましたが、Bitbucketでのコミットログの見方も説明しておきましょう。

Bitbucketのリポジトリのページに移動し、「コミット」

というリンクをクリックすると、これまでに行われたコミットの一覧が表示されます（図5）。

コミットのIDをクリックすると、コミットの詳細ページに移動します。

図5 Bitbucket上のコミットログ

作成者	コミット	メッセージ
megane9988	bc47eef	お知らせの更新
megane9988	91f6d84	見出し-classのタイプ修正 heading => heading
megane9988	bfb0245	access.htmlの削除
megane9988	81849a9	access.htmlの追加

コミットの詳細ページでは、変更されたファイルそれぞれにおける、変更箇所がSourcetreeと同様に赤と緑で表示されています（図6）。デフォルトでは差分が上下に表示されます。「横並び比較」をクリックするとモーダルウィンドウで差分が左右に並ぶ形で表示されます。

PCなどの広い画面で閲覧できる場合はこちらのほうが見やすいかもしれません。見やすいほうを利用しましょう。

図6 変更箇所の表示



デフォルトの縦並び表示



横並び表示

画像の変更履歴

画像の変更があった場合は、変更前と変更後の画像を以下の4つの方法で比較することができます（図7）。

- 「左右に並べる」（横に並べて表示）
- 「ブレンド」（上部のスライダーで変更前と変更後の画像を、透明度を変えながら表示）
- 「縦分割」（画像上のカーソルを右方向に動かすと変更後、左方向に動かすと変更前の画像を表示）
- 「ピクセル差分」（新旧の画像を重ね合わせた状態で、ピクセル単位での差分を表示）

図7 変更箇所の表示

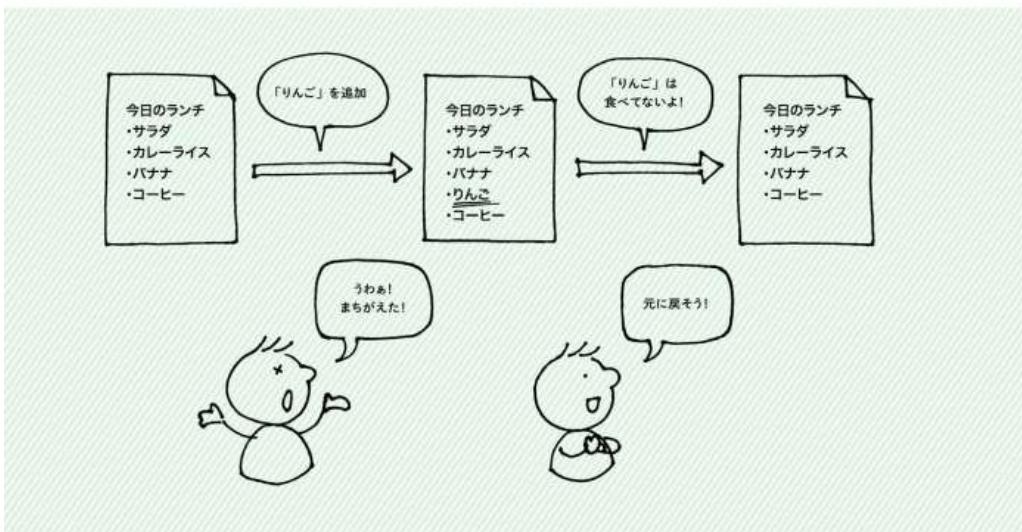


画像の変更履歴

間違って操作してしまったら？

変更箇所や記述を間違えた、必要なファイルを削除してしまった、不要なファイルを追加してしまったなど、もし何か間違った操作を行ってもコミットする前であれば簡単に元に戻せます。

もちろん、コミットしてしまっても元に戻すことはできますが、戻し方によって結果が異なるので注意が必要です。



- コミット前の変更を元に戻すには「破棄」(Macは「リセット」)
- ファイルの追加、削除も元に戻せる
- コミットを元に戻す場合は戻し方に注意

使用するコマンド

```
$ git reset [ファイル]  
$ git reset --hard [過去のコミットID]
```

04 間違って操作してしまったら？

▶ステージに追加したファイルを戻す

ステージに追加したファイルを作業ツリーに戻すには、Indexにステージしたファイルの右にある「-」ボタンをクリックするか、戻したいファイルを選択して「選択をインデックスから除く」ボタンをクリックします。「全てインデックスから除く」ボタンをクリックすると、ステージに追加

した全てのファイルを作業ツリーに戻します図1。Macの場合は、「ステージング済みのファイル」にある各ファイル左側のチェックを外すと、「ステージングに未登録のファイル」に戻ります。全てのファイルを一度に戻すには「ステージング済みのファイル」左側のチェックを外します。

図1 ステージのファイルを元に戻す



▶変更箇所を元に戻す

ステージ追加前に変更内容の誤りを見つけた場合は、以下の方法で元に戻すことができます。

変更箇所を取り消す

「作業ツリーのファイル」(Macは「ステージングに未登録のファイル」)にある目的のファイルを右クリックして表示されるメニューから「破棄」(Macは「リセット」)を選択します図2。確認のダイアログで「OK」をクリックすると変更箇所が破棄されます。

削除してしまったファイルを戻す

この方法は誤ってファイルを削除してしまった場合にも有効です。「作業ツリーのファイル」で~~■~~のついたファイルを先ほどと同じように右クリックし、「破棄」(Macは「リセット」)すれば削除記録が破棄され、ファイル自体を復帰させることができます図3。

不要なファイルを追加してしまった場合

この場合は、同じく右クリックで表示されるメニューから「削除」を選択し、単純にファイルを削除します。確認のダイアログで「OK」をクリックすると削除されます図4。

先ほどの「破棄」はファイルに加えられた変更を破棄する、この「削除」はファイル自身を削除するという意味です。似たような名前で紛らわしいので注意しましょう。

図2 変更を「破棄」



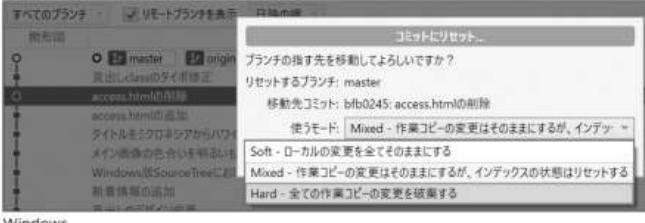
図3 「ファイルを削除」という変更を「破棄」



図4 不要なファイルは「削除」

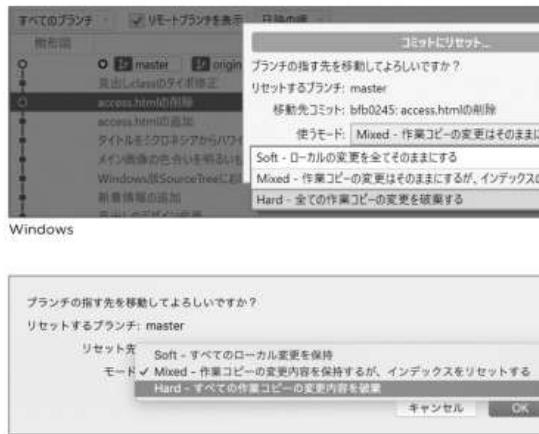


▶ コミットを戻す

戻すのプランチをこのコミットまでリセット」(Macは「□□□(プランチ名)をこのコミットまで戻す」)を選択します。すると、「コミットにリセット...」というダイアログが表示され、下記の3つの中から戻し方を選ぶことができます。

- ・「Soft」: ローカルでのこれまでの変更はそのままに、コミットだけを移動します。
- ・「Mixed」: 作業コピーの変更はそのままにするが、インデックスの状態はリセットします。
- ・「Hard」: すべてのファイルの変更を破棄して、そのコ

図5 コミットを戻す



Mac

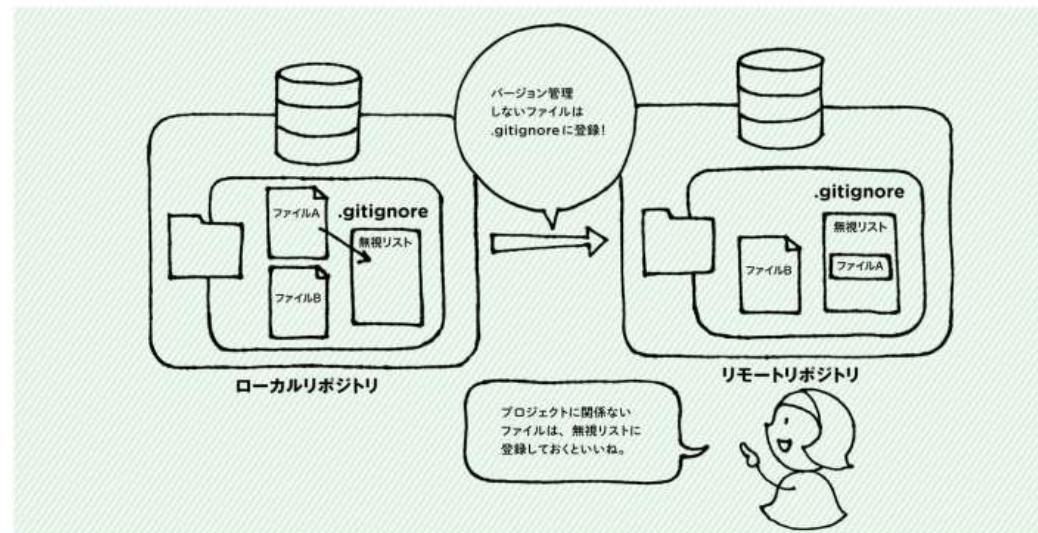


2-05

ファイルをGitの管理から外す

Gitでバージョン管理しているフォルダの中にはあっても、バージョン管理が不要だったり、バージョン管理したくないファイルが出てくることもあります。

ここではそういったファイルをGitのバージョン管理の対象から外す方法について説明します。



- 特定のファイルをバージョン管理の対象外に設定
- 「.gitignore」ファイルで指定する
- コミットしたファイルをバージョン管理から外す

使用するSourceTreeの機能



設定

▶ バージョンを管理しなくてもよいファイル

Webサイトを制作するうえで、通常バージョンを管理する必要のないファイルとして、次の2種類が考えられます。ひとつはOSやアプリケーションが自動的に生成するようなファイルで、Webサイトなどを制作する際には、不要となるようなファイル群です。もうひとつはファイルとしては必要ですが、セキュリティ面の考慮や更新頻度、ファイル容量などの観点からあえて管理しないファイルです。

④ 自動生成される不要なファイルの例

```
Thumbs.db (windows用)
.DS_Store (mac用)
.sass-cache (Sassのcacheディレクトリ)
.idea (PhpStormの設定ファイル)
```

⑤ 管理しないファイルの例

(WordPressの場合)

⑥ wp-config.php

このファイルはデータベースへの接続情報のURLやパスワードなどが記載されているため管理を行いません。

⑦ wp-content/uploads/

WordPressから画像がアップロードされた場合に保存されるディレクトリです。更新頻度が高く、また容量が大きくなりやすいため管理を行いません。

▶ 管理しないファイルを指定する「.gitignore」

バージョン管理しないファイルを指定するには、**「.gitignore」**（ギットイグノア）という名前のファイルを作成します。Gitでは、.gitignoreファイルの中に書かれたファイルはバージョン管理を行わないようになっています。

Sourcetreeではリポジトリにおいて、「設定」→「詳細」タブで表示される「リポジトリ固有の無視リスト」（Macは「設定」→「高度な設定」タブの「リポジトリ限定無視リスト」）から.gitignoreファイルの編集を行うことができます（図1）。「編集」ボタンをクリックすると、エディタが起動（あるいはエディタを指定）して、.gitignoreファイルが開きます。

バージョン管理を行わないファイルの名前（ここでは「ignoretest.txt」）を入力し、保存して閉じます（図2）。すると、.gitignoreファイルが「作業ツリーのファイル」（Macは「ステージングに未登録のファイル」）に表示されます。この時点でignoretest.txtファイルは管理の対象から外れます。試しにignoretest.txtをメモ帳などで作成し、管理されているディレクトリに保存してみましょう。作業ツリーには表示されないはずです。

図1 「.gitignore」ファイルの確認



図2 「.gitignore」に管理しないファイル名を記述確認



▶ .gitignoreファイルの書き方

前　述の例では、バージョン管理から除外する対象を「ignoretest.txt」というようにファイル名で指定しましたが、そのほかにも、たとえばディレクトリごと除外

① .gitignoreの主な書式

(ファイル名の例を「test.txt」、ディレクトリ名の例を「testdir」とする)

書式	説明
#	#から始まる行は、コメントとして扱われる。
!	!から始まる行は、その行で指定したものを除外しない。
test.txt	指定したファイルを除外する。
/test.txt	先頭の「/」はルートディレクトリ（.gitがあるディレクトリ）を表す。つまりルートディレクトリ直下にある「test.txt」が無視される。
testdir/	名前が「/」で終わる場合、該当する名前のディレクトリのみを表す。
.psd	「」+拡張子で、該当する拡張子のファイルすべてを除外する。左の例では「.psd」の拡張子がついたファイルすべてが除外される。

したり、拡張子を指定して同一拡張子のファイルはすべて除外するなど、さまざまな方法で指定できます。

表1 表2 に主な指定方法についてまとめておきます。

表2 .gitignoreの記述例

```
# ココはコメントです

# 拡張子が.logのものを除外する
*.log

# 上記で.logのファイルはすべて除外したがtest.logのみは管理に含める
!test.log

# ファイル名sample.htmlを除外する
sample.html

# ルートディレクトリにあるsampledirという名前のディレクトリ以下を除外する
/sampledir

# dirという名前のつくディレクトリ以下を除外する
dir/
```

▶ 「グローバル無視リスト」の設定

個　別のリポジトリごとに.gitignoreファイルを設定するのが効率的でない場合、すべてのGitリポジトリに対して共通の.gitignoreの設定を行うことができます。

Sourcetreeの場合、「ツール」メニュー→「オプション」（Macでは「Sourcetree」メニュー→「環境設定」）のダイアログで「Git」タブを表示し、「グローバル無視リスト」の右にある「ファイルを編集」ボタンをクリックします（図4）。

基本的な書き方は前述の.gitignoreファイルの書き方と同じです。好みのエディタやCSSプリプロセッサなどを常に使う場合は、こちらでそれらの設定ファイルなどを指定しておくといいでしょう。

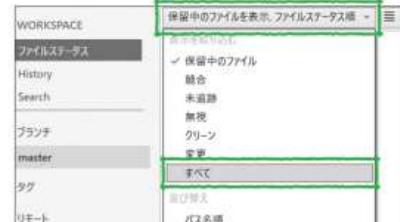
ただし、便利だからといって、すべてをグローバル設定に加えてしまうことはおすすめできません。チームで開発を行う場合は、開発者ごとに設定が異なり、返って管理が面倒になってしまいケースがあるからです。チーム内で相談して、上手に活用するようにしましょう。

図4 グローバル無視リスト



▶ すでにコミットしてしまった内容を管理しないようにする方法

す すでにコミットしているファイルを、バージョン管理しないようにしたいということもあるでしょう。
その場合は
・該当ファイルを追跡しないように設定する
・該当ファイルを`.gitignore`に追加する
という手順でバージョン管理から除外できます。



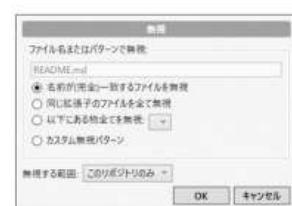
① 「ファイルステータス」の画面で、「保留中のファイルを表示」(Macでは「すべて」)を選択します。



② 「作業ツリーのファイル」(Macは「ステージング」)に未登録のファイルで、表示されたファイルから除外したいものを右クリックし、表示されたメニューから「追跡をやめる」(Macでは「追跡を停止する」)を選択します。



③ ファイルを再度右クリックし、今度は「無視」を選択します。以上で`.gitignore`ファイルへの追記が行われ、今後このファイルはバージョン管理から除外されます。



④ ファイルの除外方法をラジオボタンで選択し、「無視する範囲」(Macは「この無視エントリの追加先」)を選んでから、「OK」をクリックし除外に追加します。
除外したファイルとして削除された変更履歴と`.gitignore`に追記された内容をまとめてコミットしておきましょう。以上で、すでにコミットしてしまった内容を管理しないようにする方法が完了です。

3

Chapter 3

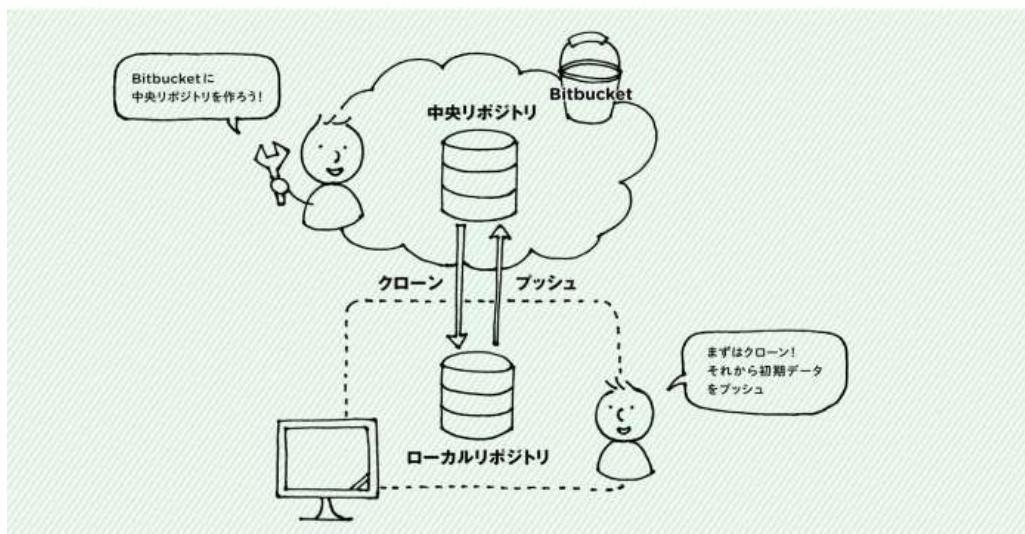
複数メンバーでの運用

複数のメンバーで Web サイトなど1つの成果物を開発する際、
Git と Sourcetree の使い方を紹介していきます。
うまく使いこなすことで、チーム体制で行う制作や開発を効率化できます。

Gitの作業環境を準備しよう

Gitを使って複数メンバーで共同作業を行うには、作業環境の構築が必要になります。

ここでは、まず最初に行うことになる「中央リポジトリ」の構築を含めた、準備作業について紹介します。



- Bitbucketを利用して中央リポジトリを準備する手順
- 中央リポジトリとローカルリポジトリの関係

使用するコマンド

```
$ git add -A
$ git commit -m "[コメント]"
$ git clone [クローンするリポジトリURL]
$ git push [リモートリポジトリ] [ブッシュするブランチ]
```

使用するSourcetreeの機能



▶ 複数メンバーの中心となる中央リポジトリ

Gitを利用して複数メンバーで共同作業を行う場合、メンバー間の連携のための中継点（ハブ）として「中央リポジトリ」を用意することが一般的です。そこで本章では、Webサイト制作の現場を事例とし、次のようなメンバーを想定したうえで解説を進めていくことにします。

ストーリーとしては、某XというホテルのWebサイトを制作するにあたって、ディレクターのCさんの指示のもとで、クリエイターのAさん、Bさんの2人が実制作を担当するというもの。2人のクリエイターは、いずれもHTML/CSSおよびJavaScriptなどのコーディング担当者です。

Gitを利用した共同作業においては、Aさん、Bさんは、それぞれの制作環境（PC）にローカルリポジトリを作成して各自の作業データを管理します。また、各ローカルリポジトリの状態は、中央リポジトリへと通時プッシュ（push）することで、Aさん、BさんともにWebサイト全体の作業

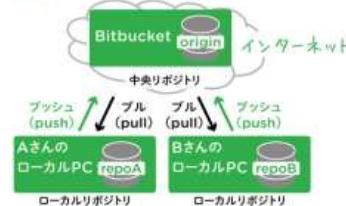
データの最新状態を常に共有できることになります。

このときのリポジトリの関係は図のようになります（図1）。図を見てわかるように、特に複雑な関係ではなくシンプルな関係です。以下、この図に基づき、Aさんのリポジトリを「repoA」、Bさんのリポジトリを「repoB」、そして中央リポジトリを「origin」と呼ぶことにします。

ここで、共同作業を前提とした作業環境を準備するためのワークフローを考えてみましょう。基本的なフローは、図2に示すように、まずはAさんが中央リポジトリを作成し、それをクローンしてローカルリポジトリを作成。その後、Webサイト制作に必要な基本データをローカルリポジトリの管理下に置くことで、Aさんのローカルリポジトリと中央リポジトリを同期するという流れになります。

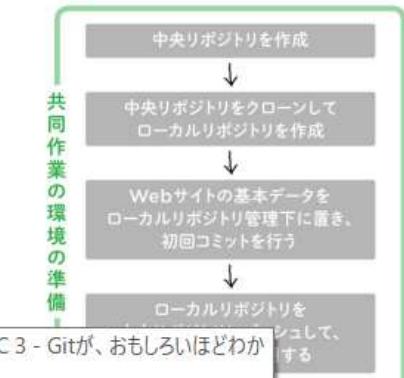
本節では、ここまで流れを具体的に紹介していきましょう。

図1 Gitを利用した共同作業用のリポジトリの関係図



Aさん、Bさんという2人の作業担当者のローカルリポジトリと中央リポジトリ。各担当者ごとに中央リポジトリへプッシュ、ブルすることで、全体的な作業データの状態を同期します。

図2 共同作業のための環境の準備



MEMO 中央伊藤 浩也さんの Kindle for PC 3 - Gitが、おもしろいほどわかる基本の使い方33 改訂新版 Chapter 3

共通利用する「リモートリポジトリ」を便宜的に「中央リポジトリ」と呼んでいます。Gitでは、共通利用を目的とせず、たとえば個人的なバックアップ用リポジトリをリモートリポジトリとして登録することもできます。本書における「中央リポジトリ」は、共同作業する各メンバーのローカルリポジトリ間の連携をとるために、BitbucketやGitHubなどのホスティングサービス上に用意したリポジトリをリモートリポジトリとして登録したものです。

▶ Bitbucketに中央リポジトリを作成する

ま ずは、Aさん、Bさんの作業データを同期するための中継点（ハブ）となる中央リポジトリを作成します。中央リポジトリは、Aさん、Bさんそれぞれの作業現場からアクセスする必要があるため、Bitbucket上に作成することになります。

ブラウザでBitbucketのWebサイトへアクセスして、Aさんのアカウントを作成し（P.018「1-03 Sourcetreeをインストールする」参照）、「XホタルWebサイト」用のリポジトリを作成します。

Bitbucketにログインしたあと、画面左側にある「+」ボタンをクリックして表示されるcreateメニューで「リポジトリ」を選びます。新規リポジトリの作成画面で、リポジトリ名を入力し、アクセスレベルの「これは非公開リポジトリです」のチェックボックスをチェック。リポジトリタイプはGitを選択して、「リポジトリの作成」ボタンをクリックします^{図3}。これで中央リポジトリが作成できました。

このとき、リポジトリにはまだ何のデータも登録されていません。つまり空のリポジトリとして作成したことになります。

▶ 中央リポジトリをクローンしてローカルリポジトリを作成

B itbucketで新規リポジトリを作成すると、リポジトリの概要ページが表示されます。初期状態では、まだデータが何も登録されていない空のリポジトリのため、概要ページにはリポジトリのセットアップメニューが表示されています。

図4 Bitbucketのリポジトリのクローン画面



図3 Bitbucketの新規リポジトリ作成ページ



新規リポジトリの作成画面で、リポジトリ名を入力し、アクセスレベルの「これは非公開リポジトリです。」のチェックボックスをチェック。リポジトリタイプをチェック。リポジトリタイプはGitを選択して、「リポジトリの作成」ボタンをクリックします^{図3}。これで中央リポジトリが作成できました。

「Sourcetreeを開きますか？」（Macは「このページでSourcetreeを開くことを許可しますか？」）という画面が表示されたら、「Sourcetreeを開く」ボタン（Macは「許可」ボタン）をクリック^{図5}。すると、Sourcetreeが自動的に起動されます（Sourcetreeがあらかじめインストールされていることが前提です）。Sourcetreeが起動すると、自動的に「Clone」ウィンドウ（Macは「新規にクローン」ダイアログ）が開きます。ここでローカルリポジトリを作成するディレクトリとブックマーク名を指定し、「クローン」ボタンをクリックしましょう^{図6}。

これでAさんのローカルリポジトリ(repoA)の作成は完了です。



Macの「新規にクローン」ダイアログ

図5 外部プロトコルリクエスト

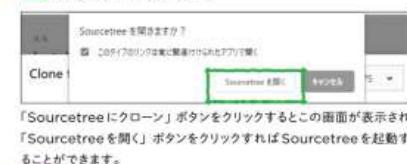
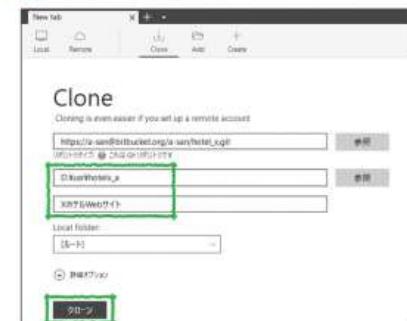


図6 「Clone」ウィンドウ



保存先のパスとブックマークの名前などを指定して「クローン」ボタンをクリックすれば、ローカルリポジトリが作成されます。

▶ ローカルリポジトリ「repoA」にデータを登録して初回コミット

B itbucketからクローンして作成したAさんのローカルリポジトリrepoAには、まだ何のデータも登録されていません^{図7}。つまりBitbucket上に作成した中

央リポジトリと同様に空のリポジトリです。ここに、あらかじめ作成しておいたWebサイト制作用の基本データを登録していきましょう。

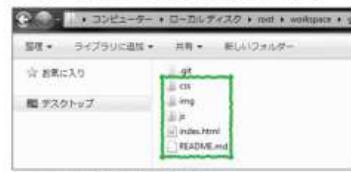
図7 ローカルリポジトリrepoAの初期状態



repoAの保存先ディレクトリとして指定したフォルダウインドウを開き、その中にWebサイト制作用の基本データを保存します^{図8}。基本データとは、Webサイトの基本的なディレクトリ構造や各コーナーごとのテンプレートHTMLおよびCSS、またサイト全体で共有するJavaScriptコードなどの一連のデータです。

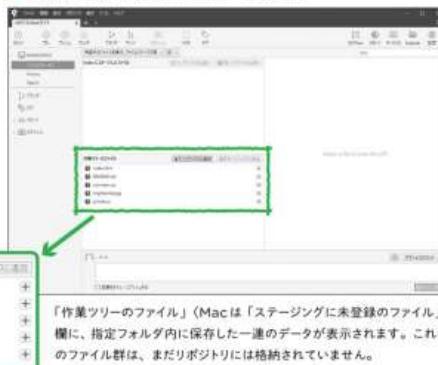
基本データ一式を保存して、再度Sourcetreeの画面を開くと、「ファイルステータス」画面の「作業ツリーのファイル」(Macは「ステージングに未登録のファイル」)欄に先ほど保存した一連のデータが表示されます^{図9}。今回保存したデータはすべて同期するという前提で、まずは「全てインデックスに追加」ボタンをクリックしましょう。(Macは「ステージングに未登録のファイル」左側のチェックボックスにチェックを入れる)。すると、一連のデータが上部の「Indexにステージしたファイル」(Macは「ステージング済みのファイル」)欄に移動します^{図10}。

図8 ローカルリポジトリrepoAの保存先フォルダ



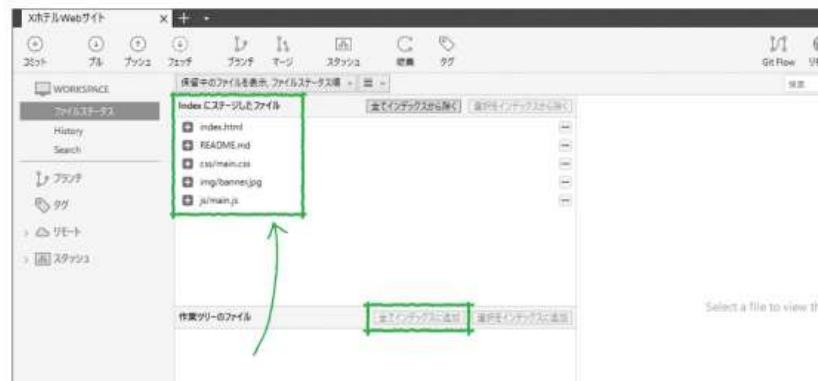
このフォルダにWebサイト制作用の基本データ式を保存します。

図9 Sourcetreeの「ファイルステータス」画面



「作業ツリーのファイル」(Macは「ステージングに未登録のファイル」)欄に、指定フォルダ内に保存した一連のデータが表示されます。これらのファイル群は、まだリポジトリには格納されていません。

図10 リポジトリに格納するファイルをステージに移動



「全てインデックスに追加」ボタンをクリックすると(Macは「ステージングに未登録のファイル」左側のチェックボックスにチェックを入れる)、一連のデータが上部の「Indexにステージしたファイル」欄に移動します。リポジトリに格納するファイルはいったんステージに移動するというのが、Gitの基本的なルールです。

続けて、メニューバーの「コミット」(commit)ボタンをクリックします。画面下部にコミットのコメント入力欄が表示されるので、「初回コミット」などと入力します。また、ここでは「変更をoriginにすぐにプッシュする」(Macは「コミットをただちにorigin/masterプッシュする」)チェックボックスのチェックを外しておきましょう。「コミット」ボタンをクリックします。

これで、AさんのローカルリポジトリrepoA上の初期状態の構築は完了です^{図11}。

図11 Sourcetreeでコミットしてリポジトリに格納



データをリポジトリに格納するために行なう作業がコミットです。コミットするためには、必ずコメントをつけなければなりません。

▶ ローカルリポジトリから中央リポジトリへプッシュ

AさんのローカルリポジトリrepoAには、Webサイト制作用の基本データが登録されて準備完了となりましたが、まだ中央リポジトリは空のままで。

そこで、メニューバーの「プッシュ」ボタンをクリックして、repoAを中央リポジトリに同期します。「プッシュ」ボタンをクリックすると、「プッシュ」画面が開きます^{図12}。プッシュ先リポジトリに、中央リポジトリの情報が表示されているのを確認します。現時点では、ローカルブランチと「master」が表示されていますが、まだ中央リポジ

トリには何のデータも登録されていないため、プッシュするブランチの「リモートブランチ」は空欄になっています。また、プッシュ対象チェックボックスもチェックされていません。そこで、プッシュ対象チェックボックスをチェックしません。

すると、自動的にリモートブランチに「master」がセットされ、同時に追跡中チェックボックスもチェックされます。この状態で「プッシュ」(Macは「OK」)ボタンをクリックしましょう。基本的に、これだけの作業でrepoAを中央リポジトリへ同期することができます。

ここまででの作業で、Gitを利用した共同作業のための基本準備が完了です。この作業環境でBさんが参加することで、Gitを利用した共同作業が可能となります。他のユーザーを参加させる方法は、次節で解説します。



図12 Sourcetreeの「プッシュ」画面

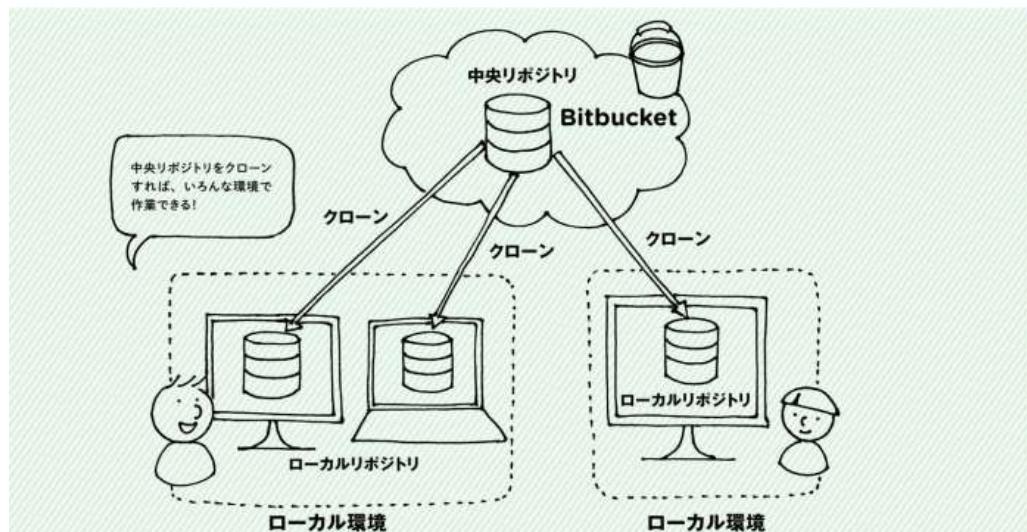
クローンで作成したrepoAは、プッシュ先として中央リポジトリが自動的に設定されています。プッシュするブランチでは、プッシュ対象チェックボックスをチェックすることで、自動的に中央リポジトリのブランチ名と追跡中がセットされます。

3-02

Chapter 3
複数メンバーでの運用

複数のPCでクローンするには

BitbucketなどのGitサービス上で中央リポジトリを作成したら、それを各スタッフごとのローカルPCへクローンすることで、共同作業環境を構築できます。中央リポジトリをローカルリポジトリへクローンする手順を見ていきましょう。



- 中央リポジトリを利用して共同作業を行うためのワークフロー
- 中央リポジトリへプッシュ(Push) / プル(Pull)を行う際の手順 / ルール

使用するコマンド

```
$ git clone [クローンするリポジトリのURL]  
$ git pull [リモートリポジトリ] [マージするブランチ]  
$ git push [リモートリポジトリ] [プッシュするブランチ]
```

使用するSourcetreeの機能



▶ 複数のマシン、複数のユーザーで共同作業するためのクローン

前節では、Webサイト制作現場で複数メンバーが共同作業を行うことを想定して、まずは共同作業の中継点（ハブ）となる中央リポジトリの作成について紹介しました。本節では、共同作業にほかのメンバーを参加させるための基本的な作業フローについて解説します。

すでに中継点となる中央リポジトリが作成されていることが前提ですから、あとはこの中央リポジトリをクローンすることで複数のユーザーが共同作業に参加することができ

ます図1。また、これは作業メンバーがひとりであっても、複数のマシンにクローンすることでさまざまな場所、たとえば事務所のデスクトップPCとモバイル用のノートPCで同期を取りながら開発を進めるといったワークフローにも応用できます図2。

ここでは、ほかのメンバーを共同作業に参加させるための設定と、クローンしたあとの基本的なワークフローについて解説します。

図1 中央リポジトリをハブとして複数メンバーで共同作業

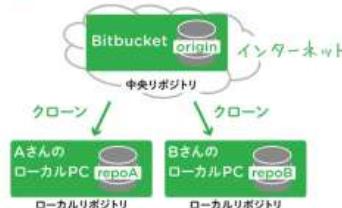
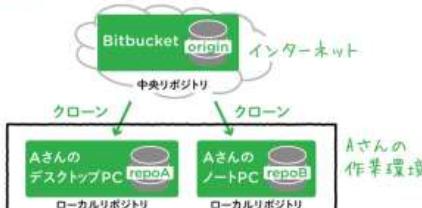


図2 中央リポジトリをハブとして複数の環境で作業



▶ Bitbucketで中央リポジトリに招待する

前節で作成したXホテルの中央リポジトリを例に、ほかのユーザーに共同作業に参加してもらうフローを追ってみましょう。

プライベートリポジトリとして作成した中央リポジトリは、現時点ではその作成者（管理者）であるAさんだけがアクセスすることができ、ほかのユーザーはアクセスすること

はできません。そこで、このプライベートリポジトリにほかのユーザーがアクセスできるように「アクセス管理」を設定する必要があります。

Bitbucketでは、リポジトリの「設定」→「ユーザーとグループのアクセス権」画面でユーザーを登録することで、招待メールを送信することができます図3。

図3 アクセス管理画面でユーザーを追加



「設定」→「ユーザーとグループのアクセス権」画面で、Bitbucketのユーザー名もしくはメールアドレスを指定してリポジトリユーザーを追加できます。ユーザーごとに権限が設定できますが、プッシュを許可するには「書き込み」もしくは「管理」に設定する必要があります。アクセス管理画面では「追加」ボタンをクリックしたタイミングで招待メールが送信されます。

ユーザー登録には、Bitbucketのユーザー名もしくはメールアドレスを指定することができますが、リポジトリにアクセスするためにはBitbucketアカウントが必要なため、あらかじめアカウントを取得しておくほうが効率的です。

XホテルのWebサイト制作を共同で行うBさんは事前

にBitbucketアカウントを取得しているとして、Bさんのユーザー名を登録すれば、Bさん宛に招待メールが送信されます。招待メールを受け取ったBさんは、メールに記載されているリンクにアクセスして「招待を承認」すれば、中央リポジトリにアクセス可能となります。**図4**。

図4 招待の承認ページ

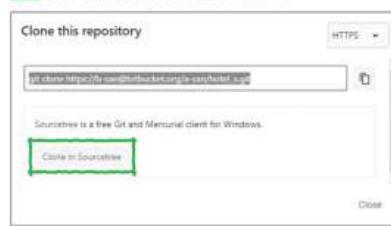
▶ 中央リポジトリをクローンしてローカル作業環境を作る

Aさんが作成した中央リポジトリのユーザーとなったBさんは、その中央リポジトリを手元のPCにクローンすることで、ローカルな作業環境を作ることができます。実際にクローンしてみましょう。

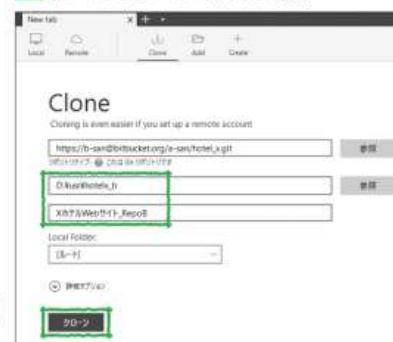
Bitbucketにログインして、Xホテルの中央リポジトリの概要ページを開き、画面左にある「+」ボタンをクリックして表示されるサイドメニューで「このリポジトリをクローンする」を選びます。クローン用のダイアログ画面で

「Clone in Sourcetree」ボタンをクリックします。**図5**。P.077と同様に「Sourcetreeを開く」ボタンをクリック。Sourcetreeが起動して「Clone」画面が表示されます。**図6**。ローカルリポジトリを作成するディレクトリとブックマーク名を指定し、「クローン」ボタンをクリックすれば、Bさんのローカルリポジトリ(repoB)が作成されます。

これでAさん、Bさんによる共同作業環境が整いました。

図5 BitbucketからSourcetreeへクローン

「Clone in Sourcetree」ボタンをクリックします。

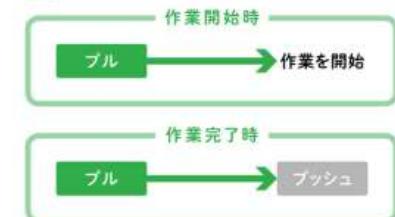
図6 クローンによるローカルリポジトリの作成

Sourcetreeの「Clone」画面(Macは「新規にクローン」)が表示されたら、保存先のパスとブックマークの名前などを指定して「Clone」ボタンをクリックします。

▶ 作業を開始する前、プッシュする前に必ずプルする

Gitを利用して複数メンバーで共同作業を行う場合、中央リポジトリが常に正である状態を保つこと、またそれを意識しておくことです。そのため、ファイルの編集作業を行う際は、作業開始時点で必ずプルするように習慣づけることが大切です。

また、Aさん、Bさんともに、相手がいつプッシュして中央リポジトリがアップデートされるかはわかりません。中央リポジトリの状態は作業中でも変わっている可能性があるのです。したがって、自分の作業が完了して中央リポジトリにプッシュする際も、**プッシュの前には必ずプル**(pull)することを忘れないようにしましょう。**図7**。

図7 プルのタイミング

作業開始時点およびプッシュする前には、必ずプルして中央リポジトリとローカルリポジトリを同期するようにします。

▶ プルしたときに発生する警告

つまり、中央リポジトリがアップデートされ、ローカルよりも先行しているときは必ずプルできるということです。プルできる状態であるにもかかわらず、プルせずにプッシュしようとすると警告が発生します。

プルとは、実際にはフェッチ(fetch)とマージ(merge)という一連の処理です。中央リポジトリとローカルリポジトリに作業履歴の差分があるとき、作業履歴を統合するための処理がマージです。ただし、中央リポジトリではマージを行なうことができません。マージはローカルリポジトリ側

で処理する必要があるため、作業履歴に差分がある状態でプッシュしようとすると「まずはマージしてください」と警告されます。

また、ローカルリポジトリ、つまり手元の作業履歴でプルしたときに、マージ対象となるファイルにコミットされていない作業内容があるときはマージを行なうことができません。したがって、この場合もプルを行うと警告が発せられます。プルを行う際は、まずSourcetreeで現在の作業内容をコミットしてから行なうようにしましょう。**図8**。

図8 コミット、プル、プッシュの順番を守る

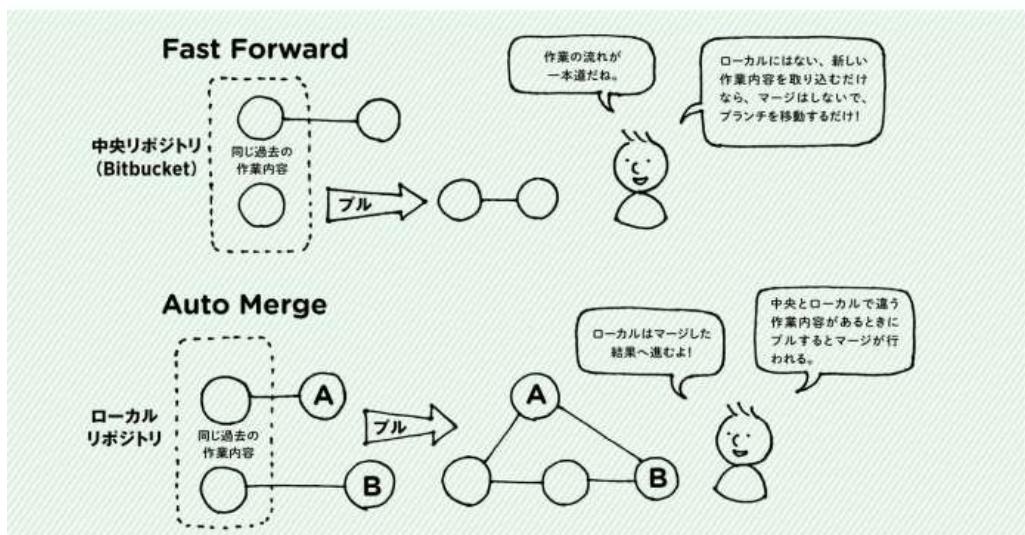
編集作業を終えてSourcetreeで同期をとる際は、必ずコミットしてから、プルし、最後にプッシュするようにしましょう。

ファイルのマージを理解しよう

複数スタッフによる共同作業に限らず、

Gitをひとりで利用する際にも重要なのがマージ（merge）です。

ここでは、マージとは何かをしっかりと理解するようにしましょう。



- マージが発生する状況
- マージによってどのようなことが起きるのか
- マージの結果がどのような状況をあらわしているか

使用するコマンド

```
$ git pull [リモートリポジトリ] [マージするリモートリポジトリのブランチ]
$ git merge [マージするブランチ]
$ git merge --no-commit [マージするブランチ]
```

使用するSourcetreeの機能



▶ マージとは何か

中 央リポジトリを中継点（ハブ）とした共同作業では、常に中央リポジトリからローカルリポジトリにプルして同期を取ることが大切です。そして、プルによって処理されるのがマージ（merge）です。

マージとは、たとえばAさん、Bさんが別々の環境で同時に編集作業を行っていた場合、当然作業内容が異なり、ローカルリポジトリの内容は違ったものとなります。それぞれに作業していた内容を中央リポジトリにプッシュするため、中央リポジトリは進捗状況に応じて変わっていきます。

したがって、たとえばAさんが行った作業内容が同期された中央リポジトリの最新情報を、Bさんは自分のローカ

ルリポジトリに取り込まなければなりません。このように中央リポジトリの最新情報を正として、自分のローカルリポジトリに取り込む作業が共同作業におけるマージです（図1）。

仮にGitを使っていないとした場合、このマージ作業を行うには手作業でお互いの作業データで更新された差分ファイルを洗い出すなど、作業はとても煩雑であり、間違いも起りかねません（図2）。Gitであれば各作業環境でコミットした時点で差分は洗い出されて記録されます。洗い出された差分の記録を精査して、新しい情報を統合していくのがマージであり、Gitが管理してくれるからこそ、ミスを最小限に抑えられることが可能となります。

図1 共同作業におけるマージ

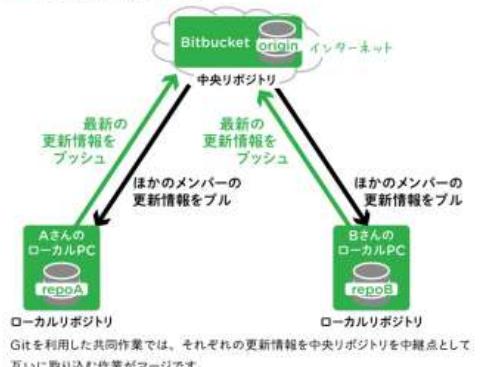


図2 複数メンバーの作業内容を統合するときに必要な差分の洗い出し

複数の作業者の2つの作業環境で、更新前のファイルと、更新のあったファイルを差分として洗い出す

↓

編集したファイルのほかに、ファイルの新規追加や、削除、名前の変更、複数したファイルなどの有無を洗い出す

▶ マージが発生する状況

実 際にマージが発生する状況を考えてみましょう。Gitを利用した共同作業においては、基本的にブールしたときにのみマージが発生します。

たとえば、Aさん、Bさんの2人が同時に作業を行っているとします。Gitを利用した共同作業ですから、Aさん、Bさんともに、まずはブールして中央リポジトリの最新情報を取り込んでから作業を開始します。

仮にAさんの作業が先に完了したとしましょう。Aさんは、

ローカルリポジトリで更新情報をコミットします。さらに、共同作業のルールに従って、中央リポジトリに対してプッシュする前にブールします。このとき、まだBさんはプッシュしていないため、中央リポジトリの情報は作業開始時点から変わっています。そのため、Aさんがブールした際の詳細情報には「Already up-to-date.」（更新なし）と表示されます。これは、「Aさんのローカルリポジトリはすでに中央リポジトリの情報を同期されてますよ」という意味

です。**図3**。Aさんは、問題なくプルできたので、作業内容を中央リポジトリにプッシュします。**図4**。

一方、BさんはAさんよりも遅れて作業が完了し、ローカルリポジトリで更新情報をコミットします。Bさんも共同作業のルールに従って、中央リポジトリにプッシュする前

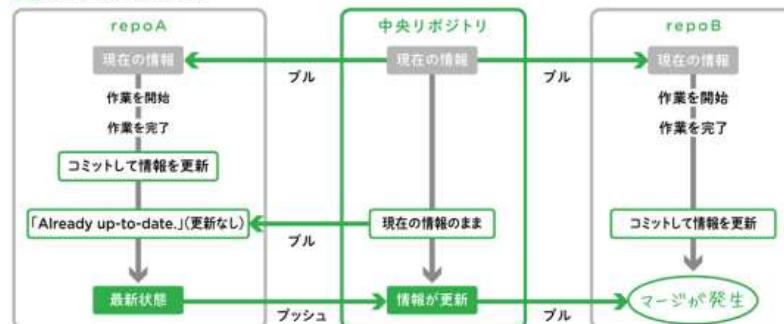
にプルを行います。すると、すでに中央リポジトリにはAさんの更新情報が記録されているため、Bさんの作業開始時より先に進んだ状態となっています。この状態でBさんがプルすると、マージが発生することになります。**図5**。

図3 中央リポジトリが更新されていないときのプル

```
git < diff --no-prefix=false < core.quotePath=false fetch -origin
git < diff --no-prefix=false < core.quotePath=false pull origin master
from https://bitbucket.org/a-san/hotel-a
  * branch master -> FETCH_HEAD
Already up-to-date.
完了しました。
```

中央リポジトリの情報が更新されていない場合にプルを行うと、「Already up-to-date.」というメッセージが表示されます。

図5 マージが発生するケース



▶ マージの結果は大きく分けて3種類

□ 一から環境で作業開始時点での中央リポジトリの情報と、作業終了後にプルした中央リポジトリの情報に差分があるときにマージが発生します。このマージ処理は、基本的にSourcetree（Git）が自動的に処理してくれる所以、問題がない場合は特に何かを意識することなくプッシュを行うなどの作業を進めることができます。

何かしらの問題が発生した場合は、マージ処理エラーとなって表示されます。実際のマージ処理では、次の表のよう4種類の処理結果が詳細情報画面に出力されます。「**Already up-to-date.**」（更新なし）はマージする必要がなかったという結果です。そのほかのマージ結果について詳しく見ていきましょう。

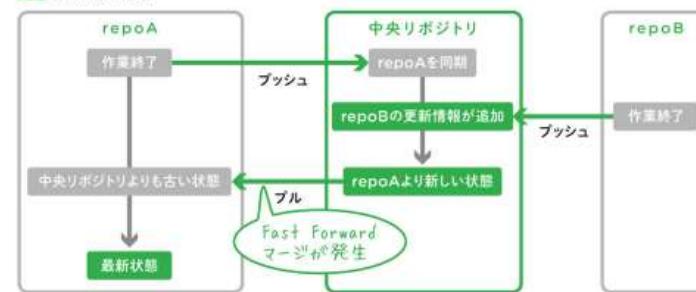
結果(出力)	内容
Already up-to-date	差分を確認したら、両方の環境の内容が同じだったので、マージをする必要がなかった場合
Fast Forward	片方の環境が古いだけで、もう一方の環境でだけ変更がある場合は新しい環境をすべて正として取り込んでしまう場合
Auto Merge	両方の環境の差分について、同じファイル中の同じ箇所の変更がなく、自動でマージが完了できた場合
Conflict	両方の環境で同じファイルの同じ箇所を変更してしまって、作業した人にどうすればよいのか確認が必要になった場合

▶ Fast Forward: 早送りになるマージ

たとえば、Aさんが最後にプル・プッシュして以降に、Bさんがプッシュしていたような場合を考えてみましょう。Aさんのローカルリポジトリのコミット記録はすべて中央リポジトリに記録されていますが、さらにBさんの最新コミットが中央リポジトリには追加されているという状況です。

この状況でAさんが次にプルした場合、repoAは古いままで、中央リポジトリだけが新しくなっているということで、単に中央リポジトリで更新された新しい情報をrepoAに取り込むだけです。これが**Fast Forward**となります。**図6**。

図6 Fast Forward



▶ Auto Merge: 単純なマージ

2つのリポジトリをマージする際、片方だけの情報が更新されているのではなく、両方ともに情報が更新されているような状況を考えてみましょう。

たとえば、Bさんが作業開始時にプルして中央リポジトリとrepoBを同期したあと、Bさんは編集作業などをしてrepoBに新しいコミットを追加したとします。これでrepoBは中央リポジトリには存在しない新しいコミット情報が記録されることになります。

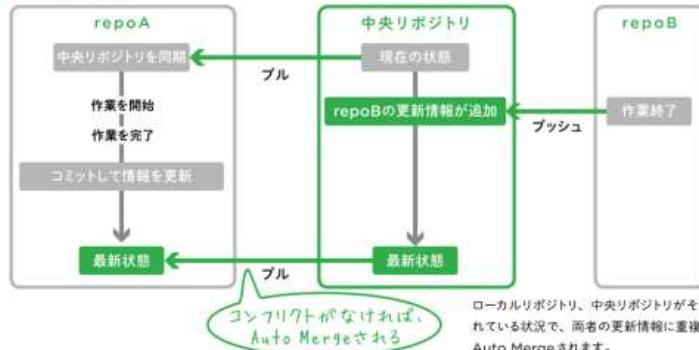
一方、平行してAさんも作業を進めており、repoAに新しいコミットを追加。そして、その新しい情報を中央リポジトリにプッシュしたとします。このとき、中央リポジトリにはrepoBには存在しない新しいコミット情報が記録されていることになります。

つまり、repoB、中央リポジトリとともに、それぞれ新しい情報が更新されているという状況です。この状況でBさんがプルすると、repoBの更新情報と

中央リポジトリの更新情報を比較してマージ処理をする必要が出てきます。

Gitは賢いプログラムのため、repoBと中央リポジトリの情報を比較して、できるだけ自動でマージしようとしてくれます。比較した結果、互いの更新情報で重なり合う部分がなければ、自動的にマージ処理を済ませてしまうのです。これがAuto Mergeです。

図7 Auto Merge



実のところ、Gitはファイルの変更内容をファイルとして管理するのではなく、ファイル内の変更箇所単位で管理しています。この管理単位を「Hunk」(ハンク)と呼びます。したがって、このHunkが重複していなければ、それ

より具体的にいえば、たとえばAさん、Bさんともに同じファイルを編集することがまったくなければ、重なり合う部分は存在しないため、問題なくAuto Mergeされます。

また、仮にAさん、Bさんが同じファイルを編集していたとしても、ファイル内の同じ箇所(行)を編集していなければ、Gitは両者の編集内容を統合してAuto Mergeしてくれます(図7)。

また、Auto Mergeの場合は、ほかの人の更新情報も取り込むことになり、自分だけのコミットではなくなるため、Auto Mergeされた時点で統合作業を行った作業履歴が

コミットされます。このコミットを「Merge Commit」といい、マージしたファイルに関するコメントも自動で作成されます(図9)。

図9 Auto Mergeされた場合のMerge Commit

```

Xcode Web サイト A Xcode Web サイト B | 1 1 1 x + 
Pulling
詳細な出力を表示
git -c diff.mnemonicprefix=false -c core.quotepath=false fetch origin
git -c diff.mnemonicprefix=false -c core.quotepath=false pull origin master
* branch master -> FETCH_HEAD
Auto-merging index.html
Merge made by the 'recursive' strategy.
index.html | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
完了しました。
  
```

▶ Auto Merge後にMerge Commitを作りたくない場合

GitでAuto Mergeした場合、通常はすぐにMerge Commitが作成されます。しかし、場合によってはMerge Commitを作成たくない場合もあります。その場合、ブル画面では「マージした変更を即座にコミット」(Macでは「すぐにマージした変更をコミットする!」)というオプション、マージ画面では「マージ後そのままコミット」(Macでは「マージ結果を直ちにコミット!」)というオプションのチェックを外してから実行します。

統合作業を確認したり、必要に応じて何か調整を行ったりする場合に利用する機能として覚えておいてください。マージ結果として残り1つが「コンフリクト」(conflict)です。コンフリクトが発生すると、Gitはマージ処理はしてくれません。問題はユーザーが解決しなければならないのです。このコンフリクトについては、次節で詳しく見ていきましょう。

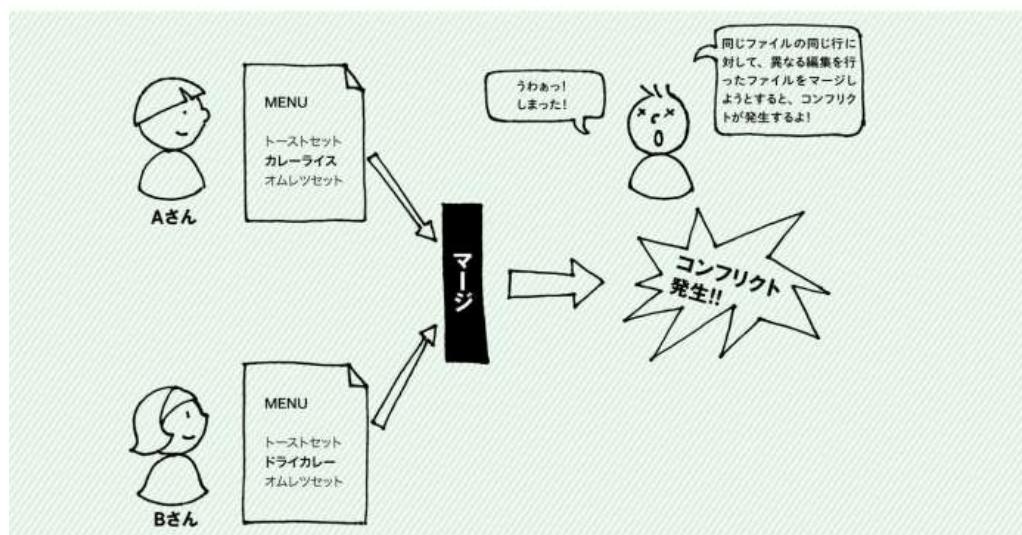
図8 GitではHunkという数行単位で更新情報を管理



コンフリクトはなぜ起こる?

マージした際に発生する可能性があるコンフリクト。

Gitを使いはじめの頃は、慌ててしまうかもしれません、コンフリクトの原因をしっかりと理解することで落ち着いて対応することができるようになります。



- コンフリクトはどのような状況で発生するのか
- マージによって発生するコンフリクトがどのような状況であるか

使用するコマンド

\$ git merge --abort

使用するSourcetreeの機能



マージ

▶ コンフリクトはどんなときに起こる?

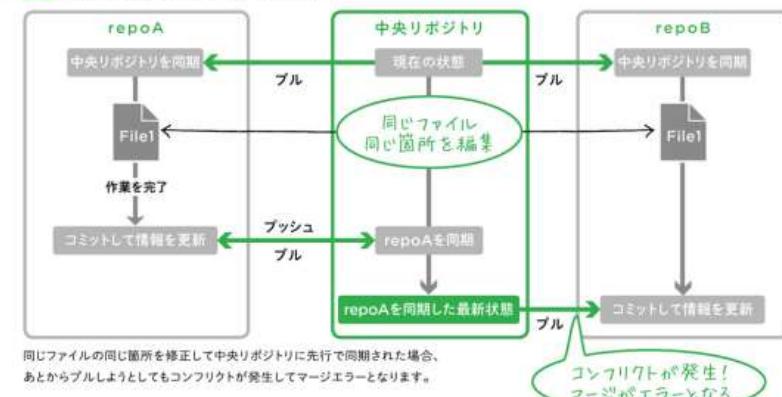
Gitを利用した共同作業を行う場合、各メンバーがそれぞれ個別のファイル編集を担当していれば、コンフリクトが発生することはないといつていでしよう。コンフリクトは同じ1つのファイルの同じ箇所に、複数の修正が同時に行われた場合に発生します。

Gitでは、更新情報をHunkという行単位で管理しています。したがって、たとえ複数のメンバーで同じファイルを編集していたとしても、それぞれがまったく異なる行を修正した場合、GitはうまくAuto Mergeしてくれます。

しかし、複数メンバーが同じ箇所を修正してしまった場合、Gitはそれらを勝手に統合したりしません。Gitにはどちらの修正が正しいかは判断できないため、その判断をユーザーに委ねるのです。

ここでもXホテルのWeb制作現場を例に考えてみましょう。たとえばディレクターからロゴ画像を小さい画像から大きい画像に変更するような指示があり、Aさん、Bさんともに、それが自分の担当だと思い込んでそれぞれに修正作業を進めてしまったとしましょう(図1)。

図1 同じ箇所を編集することによるコンフリクト



同じファイルの同じ箇所を修正して中央リポジトリに先行で同期された場合、あとからプルしようとしてもコンフリクトが発生してマージエラーとなります。

Aさんは、該当箇所を次のように修正しました

図2 図3。Aさんが先に修正作業が終り、通常通りコミット→プル→プッシュを行い、中央リポジトリへ最新コミットを同期します。

一方、BさんはAさんとほぼ同時に作業に取り掛かりましたが、修正作業は少し遅れてしまいました。Bさんは該当箇所を次のように修正しました(図4)。

Bさんは修正が終わったので、やはり通常通りコミット→プルを行います。しかし、Sourcetreeでプルしたとたん「マージできない」といった内容のメッセージが表示されてしまいます。

2人ともに同じファイルの同じ行を修正しているため、どちらが正しい情報なのかをGitは判断できません。これがコンフリクトです。

図2 修正前

```

```



図3 Aさんの修正

```

```

図4 Bさんの修正

```

```

▶ 不慮の全行コンフリクトに注意

複 数メンバーで作業しているとき、Windows、Macなど作業環境の違いで改行コードが書き換わってしまうといったトラブルも想定されます。改行コードが書き換わった状態でマージを行うと、ファイル内のすべての行がコンフリクトの対象となってしまします。

エディタの設定によっては、改行コードだけでなく、文字コードも変わってしまう可能性もあります。さらに、行末の空白やファイル末尾の改行を自動的に削除するといった設定のあるエディタもあるので注意が必要です。エディタを含めた編集ツールの設定はあらかじめ作業メンバー間で協議し、プロジェクトに対して適切に設定して統一しておく

ことが重要です。

全行コンフリクトなど、多くの箇所がコンフリクトしてしまうといったトラブルも想定されます。改行コードが書き換わった状態でマージを行うと、ファイル内のすべての行がコンフリクトの対象となってしまします。

もしくは、改行コードだけが変わったことが確かなら、一括置換を行って改めてコミットするという方法も考えられます。

いずれにしろ、全行に渡るようなコンフリクトは、共同作業においてはほかのメンバーに多大な迷惑をかけてしまうので十分に注意するようにしましょう。

Column

○ Gitの改行の自動処理設定

Gitには、WindowsとMacの改行コードの違いによる問題を回避できる機能が搭載されています。これは自動的に改行コードを変換して、Windowsのローカル環境で作業しているときは改行コードは「CR+LF」、中央リポジトリなどリモート環境にプッシュするときには「LF」に自動的に変更してくれる機能です。

WindowsでSourcetreeをインストールする際、「改行の自動処理を設定する(推奨)」というオプションがあります。これを選んでインストールすると、Gitで改行コードの自動処理を行うように設定してくれます。

また、次のようなCUIコマンドで改行の自動処理機

能をオンにすることもできます。

```
$ git config core.autocrlf true
```

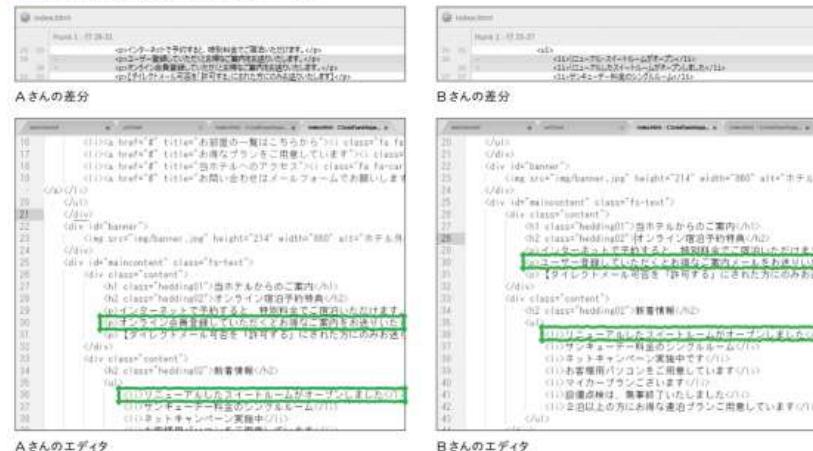
これにより、中央リポジトリなどのリモートリポジトリでのファイルの改行コードを「LF」に保つことができ、Macでブル・プッシュする際は「LF」で統一され、Windowsでブルした際は自動的に「CR+LF」に変換、プッシュする際は「LF」に変換してくれます。

▶ コンフリクトが発生したファイルの確認

Web制作現場の事例で紹介したように、コンフリクトはマージを行ったときに発生することがあります。Sourcetreeでは、ブルしたときに発生し、「Automatic merge failed」というエラーメッセージを出力して、実際のマージは行わず終了します。このとき、コンフリクトが発生していないファイルは問題なくマージされ、コンフリクトが発生しているファイルだけが「マージされません」。

コンフリクトが発生した場合、Sourcetreeではコンフ

図5 同じ箇所を編集することによるコンフリクト



Aさんのエディタ Bさんのエディタ



あとからブルしたBさんの環境でコンフリクトが発生した様子

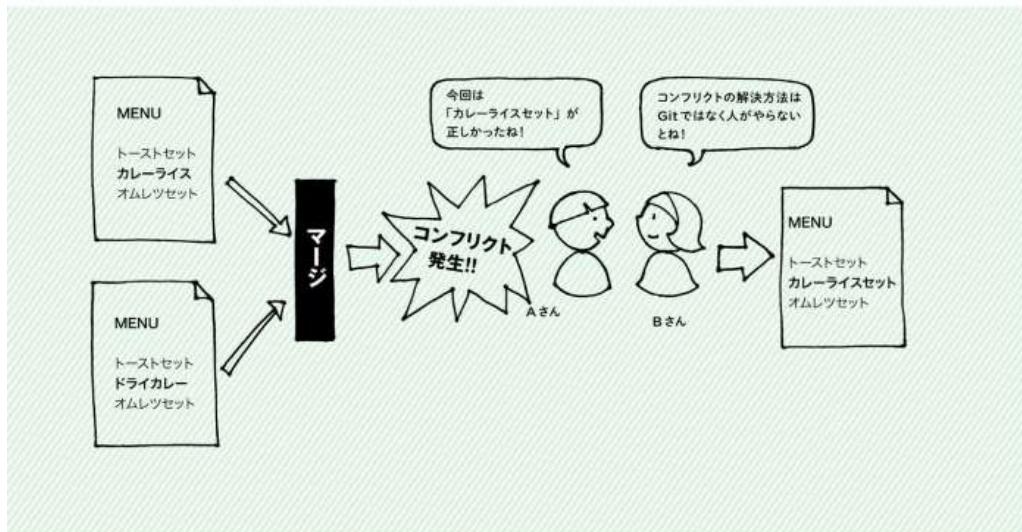
3-05

Chapter 3
複数メンバーでの運用

05 コンフリクトを解決する

コンフリクトを解決する

Gitはコンフリクトを解決してくれるわけではありません。
ユーザー自身がコンフリクトの内容を確認して、解決を図ります。
ここでは、具体的な解決手順を紹介します。



- コンフリクトしたファイルの読み方を理解する
- コンフリクトを解決してマージコミットを作成する手順

使用するコマンド

```
$ git add [コンフリクトを解決したファイル名]  
$ git checkout --theirs [マージ先ブランチの変更を正したいファイル名]  
$ git checkout --ours [チェックアウト中のブランチの変更を正したいファイル名]
```

使用するSourcetreeの機能



▶ コンフリクトが発生したファイルの内容

コンフリクトが発生したファイルは、**コンフリクトしている箇所がマークされた新しいファイルとして作業ツリーに保存され、コミットされていない状態**となります。

Sourcetreeではブランチの最上部に「コミットされていない変更があります」と表示され、これを選ぶことで作業ツリーリストに該当するファイルが△マークつきでリストアップされます。また、コンフリクト箇所を画面上で確認することもできます(図1)。

実際にコンフリクトが発生したファイルをエディタで開いてみると、コンフリクト発生箇所が次のようになっています(図2)。

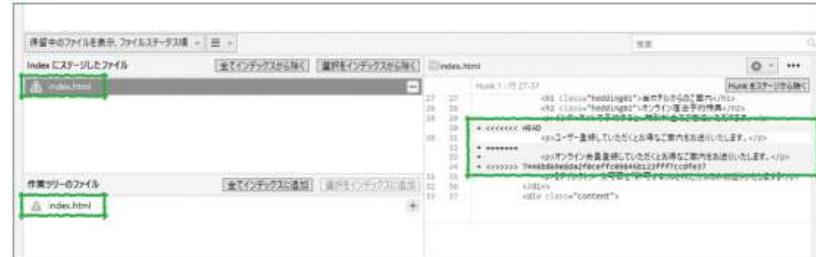
「<<<<< HEAD」という行から「>>>>> [コミットID]」という行で囲まれた部分がコンフリクトしている箇所です。「<<<<< HEAD」の次行にあるpタグは、自分で修正した情報。その次にあるpタグは、マージしよ

うとした情報。つまり、コンフリクトした相手の修正内容です。どちらが正しいのかをGitは判断することができます、人が判断する必要があります。

実際にブルを行って、このコンフリクトが発生したのはBさんですので、Bさんがその内容を確認して、何らかの判断をくださなければなりません。どうすべきかは、ケースバイケースです。Bさんが個人で判断してよい状況であれば、ファイルを修正するなどしてコンフリクトを解消します。Bさんに判断できない場合は、コンフリクト対象であるコミットを行った人、すなわちAさんに連絡を取り、協議して判断する必要があります。いずれにしろ、**コンフリクトの解決方法は人が考えなければなりません**。

共同作業においては、どういう状況では誰が判断するといった基本ルールを共有しておけば、コンフリクトも効率的に解決できます。

図1 Sourcetreeでコンフリクトしたファイルを確認



作業ツリーにコンフリクトを起こしたファイルがリストアップされ、その内容も確認することができます。

図2 コンフリクトが発生したファイルをテキストエディタで確認

```
25 <div id="maincontent" class="ta-text">  
26   <div class="content">  
27     <h1 class="hedding01">当ホテルからのご案内</h1>  
28     <h2 class="hedding02">オンライン宿泊予約特典</h2>  
29     <p>インターネットで予約すると、特別料金でご宿泊いただけます。</p>  
30   <<<<< HEAD  
31   <p>ユーザー登録していただくとお得なご案内メールをお送りいたします。</p>  
32   *****  
33   <p>オンライン会員登録していただくとお得なご案内をもう送りいたします。</p>  
34 >>>>> 17481388682ed8057534a7e07ac42311e127099  
35   </div>  
36   <div class="content">  
37     <h2 class="hedding02">新着情報</h2>  
38     <ul>  
39       <li>リニューアルしたスイートルームがオープンしました!</li>  
40   </ul>
```

「<<<<< HEAD」という行から「>>>>> [コミットID]」という行で囲まれた部分がコンフリクトしている箇所。

▶ ファイルを修正し、改めてプル、プッシュする

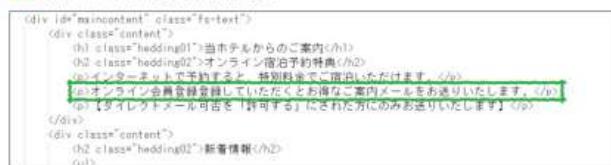
BさんはAさんと相談してコンフリクトが発生したファイルに対して適正な修正を行います^{図3}。

コンフリクトが発生していたファイルを修正して適正な状態にしたあと、BさんはSourcetreeで該当ファイルをステージに追加(add)します^{図4}。これで、Bさんは制作しているWebサイトデータとしてマージが完了したことになります。このとき、先にプルした際に、コンフリクトを

起こしたファイル以外にオートマージされたファイルがあれば同時に確認することができます。

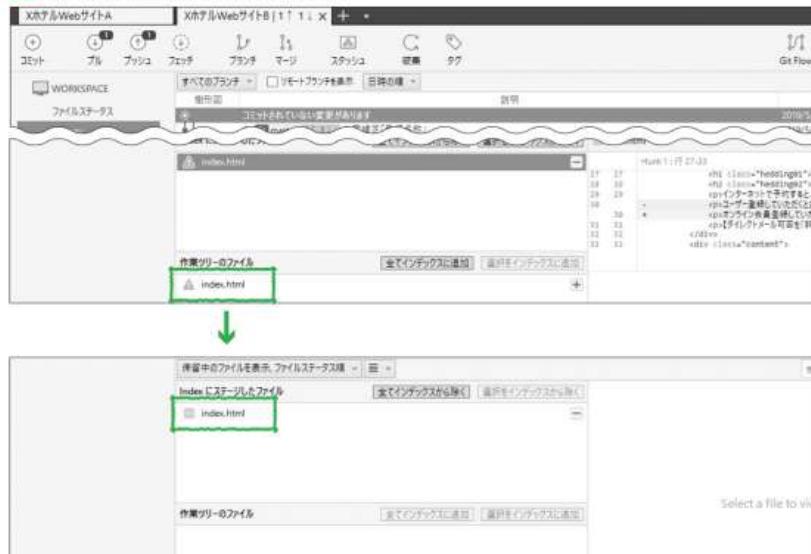
次に、Bさんは追加したファイルをコミットして、Bさんのローカルリポジトリ(repoB)を最新の状態にします^{図5}。Bさんはマージ・コミットが作成できたため、再度プルし、その後プッシュすることができるようになります^{図6}。

図3 テキストエディタで適正な情報に修正



```
<div id="maincontent" class="fs-text">
  <div class="content">
    <h1>当社からのお案内</h1>
    <h2>オンライン宿泊料金</h2>
    <p>オンライン宿泊料金</p>
    <p>オンライン会員登録していただくとお得なご案内メールをお送りいたします。</p>
    <p>1.オンライン会員登録していただくとお得なご案内メールをお送りいたします。</p>
  </div>
  <div class="content">
    <h2>新着情報</h2>
  </div>
```

図4 コンフリクトを解消したファイルをステージへ追加



コンフリクトファイルをテキストエディタで適正な情報に修正した状態では、まだSourcetree上では△マークがついています。これをステージに追加すると△マークは消え、コンフリクトを解決したことになります。

図5 コンフリクトを解消したファイルをコミット



図6 マージ・コミットが完了後のプル、プッシュ



▶ コンフリクトが発生したファイルの直し方

ここで再度、コンフリクトが発生したファイルの中身を詳しく見てみましょう。

先に挙げたHTMLファイルをテキストエディタで開くと、ところどころに「>>>」「====」「<<<」のような記号が挿入されているのがわかります。HTMLデータとしては、タグの対応づけが崩れるなどの問題が発生するため、このままの状態では使用することができません。これらの記号を削除するなどの編集を行って、HTMLデータとして適正な状態にしなければなりません。

では、この記号はどういう意味なのでしょうか。まず最初に見る「<<<<<< HEAD」は、元のファイルの情報です。次に「=====」を挟んで「>>>>>> sub」までの間がプルしたファイルの情報です。したがって、コンフリクトしたファイルを適正な状態に戻すためには、「<<<<<< HEAD」、「=====」、「>>>>>> sub」というGitの追記箇所を削除すると同時に、双方の差分を取り入れて新たなコミットを作成することになります（どちらか一方が、完全に誤りである場合のみ、いずれかを削除することとなります）。

たとえば、下記のように解決を行います。

```
<<<<<< HEAD

=====

>>>>>> sub

```

下記のように解決

```

```

どちらか一方が正しい場合、Sourcetreeのメニューを使って簡単に修正することができます。ファイル名を右クリックして表示されるコンテキストメニューで「競合を解決」→「自分（Mine）の変更内容で解決」もしくは「相手（Theirs）の変更内容で解決」を選べば「<<<<<< HEAD」などの記号も含めて、不要な情報がすべて削除されます（次ページ図7）。

ただし、元の情報もプルした情報も、いずれも間違っている場合もあります。その場合は、該当ファイルをテキストエディタで開き、手作業で修正する必要があります。その際、「<<<<<< HEAD」などの記号を漏れなく削除するように注意しましょう。

図7 コンテキストメニューでコンフリクトを解決



コンフリクトが発生したファイルを右クリックして表示されるコンフリクトメニューで「競合を解決」を選べば簡単に解決できます。

▶ Gitにコンフリクトが解決したことを知らせる

□ ンフリクトしたファイルを修正したとしても、そのファイルが作業ツリーにいる限りは、コンフリクトしている状態であるとGitは認識しています。そのため、SourceTree上でも△マークがついたままなのです。コンフリクトが解決したことをGitに知らせるためには、作業ツリーからステージへ追加する必要があります。

逆にいえば、追加してしまうとコンフリクトが解決したと認識されてしまうため、追加したファイルの修正にミスがあった場合は元に戻す必要があります。これも右クリックで表示されるコンテキストメニューの「競合を解決」→「未解決とマーク」(Macは「未解決としてマーク」)を選ぶだけで簡単に戻すことができます。

▶ コンフリクトする前の状態に戻す

プロルでマージした際、複数の箇所、複数のファイルがコンフリクトを起こしてしまうような場合、一つひとつ丁寧にコンフリクトを解決していく必要がありますが、作業をいったん停止したあとで落ち着いて解決したいといった場合は、いったんマージをキャンセルして以前の状態に戻すことができます。

SourceTreeでは、「破棄」ボタン(Macは「操作」メニュー→「リセット」)をクリックして表示される「変更を破棄」ウィンドウの「全てリセット」タブ画面で「全てリセット」ボタンをクリックします。これにより、最後に行ったコミットまで戻り、コンフリクトが発生する直前の状態まで戻ることができます(図8)。

※「リセット」ボタンもありますが、デフォルトのツールバーには表示されません。ツールバーを右クリックすると表示される「ツールバーのカスタマイズ」から追加できます。

図8 変更を破棄



「破棄」ボタンをクリックして表示される「変更を破棄」ウィンドウの「全てリセット」タブ画面で「全てリセット」ボタンをクリックすると、直前のコミットまで戻ることができます。

Column

● コミットしたユーザーを確認

コンフリクトが発生した場合、コンフリクトの解決を自分で判断できない場合は、コンフリクト相手と協議して解決策を考えなければなりません。2人で共同作業しているのであれば相手が誰かは明白ですが、3人以上で作業している場合は、相手が誰なのかを確認しなければなりません。

実際、コンフリクトを起こしているのは、自分自身のコミットと誰かのコミットです。したがってコミットしたユーザーがわかれれば、協議する相手がわかります。

SourceTreeではプランチリストの「作者」という列にコミットしたユーザー名が表示されます。ここを確認して、解決策を協議する相手を特定しましょう。

Column

● コンフリクトを修正せずにコミット

コンフリクトが発生したとき、それを解決せずにコンフリクトしたままの状態でステージに追加し、コミットしてしまうこともできます。ただし、その場合はGitによって記述された文字列がそのまま反映されてしまうため、問題となる場合があります。

本節の事例では、対象はHTMLファイルです。コンフリクトしたままの状態だと、右の記述のままHTMLファイルが保存されてしまい、ブラウザでの表示がおかしくなってしまいます。

基本的にコンフリクトは必ず解決したあと、コミットするようにしましょう。

```
<<<<< HEAD

=====

>>>> 618fe14d68a24f3099abf0eee05bd8bb12a4
8438
```

3-06

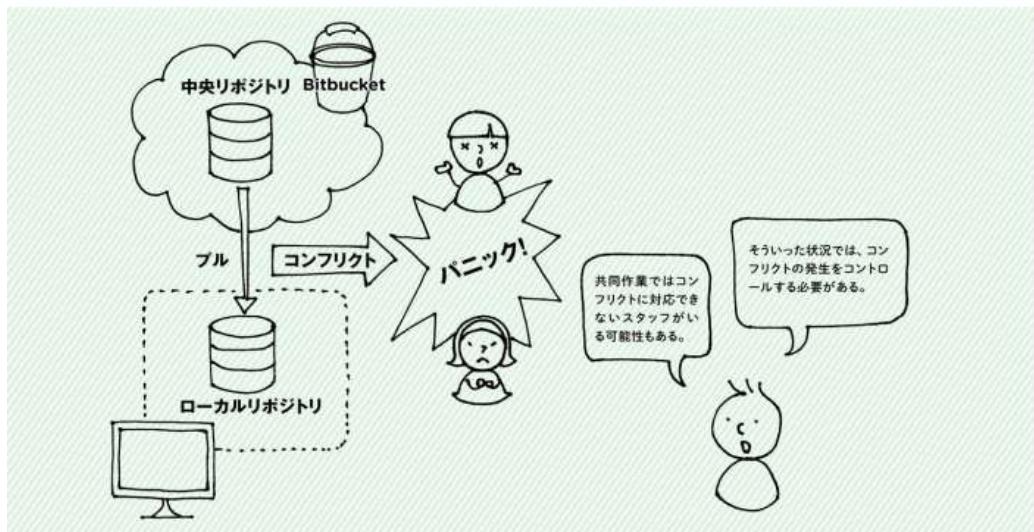
Chapter 3
複数メンバーでの運用

コンフリクトしない状態を保つ

コンフリクトが完全に起こらないようにすることは困難です。

特に共同作業では、コンフリクトは発生するものと考えましょう。

ただ、できるだけコンフリクトを抑えるような運用上のポイントもあります。



- ✓ コンフリクトの発生をできるだけコントロールする
- ✓ Fast Forwardな状態と、Fast Forwardではない状態の違いを理解する

▶ Fast Forwardな状態ではコンフリクトが発生しない

Gitによるマージ作業では、「Fast Forward」（ファストフォワード）や「Non Fast Forward」といった状態（関係性）を表す用語が出てきます。これらは、マージ先のブランチとマージ元のブランチの関係性を表すもので、「マージ元」となるブランチに記録されているすべてのコミットが、「マージ先」となるブランチにすでに記録されているかどうかを示します。

複数スタッフによる共同作業の場合で考えてみましょう。Aさん、Bさんはそれぞれのローカルリポジトリで制作作業をしています。そして、適時作業状態を中央リポジトリとプル・プッシュすることで、互いのローカルリポジトリを同期する、というのが共同作業の基本です（図1）。

Aさん、Bさんとともに、前日の作業完了時に最終コミットを中央リポジトリにプッシュしていることを前提とします。そして、次日の作業を開始する時点では、まずは中央リポジトリからプルして、ローカルリポジトリを最新の状態にすることから始めます。なぜならば、Aさんが作業している間に、Bさんが作業を進めて中央リポジトリにBさんのコミットをプッシュしている可能性があるからです。

Aさんは前日作業終了時の最終コミットを中央リポジトリ

にプッシュしてあるため、この日の最初のプルでは、AさんのローカルリポジトリrepoAにあるコミットはすべて中央リポジトリにすでに存在しています。このとき、前日のAさんの最終プッシュ後に、Bさんが中央リポジトリにプッシュしていたとしても、Aさんの最終コミットまでの履歴はrepoAと中央リポジトリの関係（正確にはブランチの関係）をFast Forwardと呼びます。Fast Forwardの関係にあるときにプルすると、実際にはマージは発生せず、repoAに存在しないより新しいコミットが追加され、repoAの最新状態がその新しいコミットに移動するだけとなります（図2）。

図2 マージが行われないということは、コンフリクトが発生することはありません。

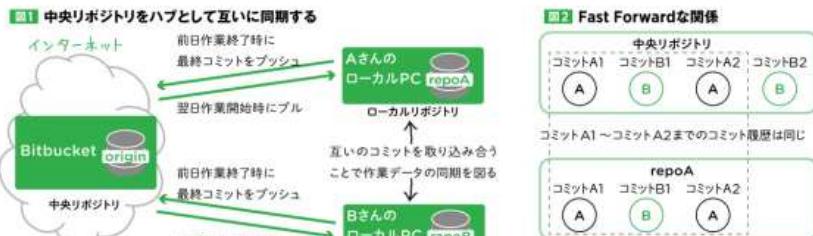
ちなみに、同様の状況において、もしAさんの最終プッシュ以降、Bさんが何もしていない場合、中央リポジトリとrepoAはまったく同じ状態のはずです。まったく同じ状態でプルした場合は、当たり前ののですが、何も同期されることはありません。このときのSourcetreeの出力はFast Forwardではなく、「Already up-to-date.」（すでに最新になっているという意味）となります。

図1 中央リポジトリをハブとして互いに同期する



共同作業の基本として、常に正しい状態に保たれた中央リポジトリをプル・プッシュして、互いのローカルリポジトリにほかの作業者のコミットを取り込み、同期された状態を保ちます。

図2 Fast Forwardな関係



Fast Forwardの関係でプルすると、マージは行われず、repoAにコミットB2が取り込まれる

図3 Fast Forward状態でプルした場合

```
Pulling
 標準的なエラーを表示
git < gitversion> pull -v --no-rebase -t core.autosetupfetch --fetch=origin
* branch    master      -> FETCH_HEAD
Updating 3d7f5e3..9472318
 从 .../repoA
 * [fast-forward] master: 9472318...[REDACTED]
```

Fast Forward状態でプルした場合、Sourcetreeでは「Fast-forward」という出力が確認できます。

repoAのコミット履歴はすべて中央リポジトリに存在し、かつrepoAには存在しないより新しいコミットが中央リポジトリにあるとき、Fast Forwardな関係となります。

▶ Git初心者のリポジトリをFast Forwardに保つ

Fast Forwardであれば、マージは行われず、結果的にコンフリクトは発生しません。つまり、コンフリクトを発生させたくない場合は、Fast Forwardな状態に保つことが重要となります。では、Fast Forwardを保つためにはどうすればいいでしょうか。

実際には、完全にFast Forwardな状態を保つのは困難です。ただし、運用ルールを決めて、それにしたがってGitを利用することで、ある程度Fast Forwardな状態を保つことも可能です。

それには、まずFast Forwardな状態に保たなければならぬスタッフがいるかどうかがポイントになります。共同作業においてコンフリクトが発生したとき、どのスタッフも適時対応可能という状況であれば、緩やかな運用ルールでかいませんが、Git初心者でコンフリクトが発生したときに混乱してしまい、リポジトリを壊してしまう可能性のあるスタッフがいる場合は、そのスタッフのリポジトリをできる限りFast Forwardに保つような運用ルールを設けることが重要です。**図4**。

図4 Git初心者はコンフリクトを解決できないかもしれません



▶ Fast Forwardを保つための運用ルール

リポジトリをできる限りFast Forwardに保ち、コンフリクトに遭遇させたくないスタッフがいる場合は、ある程度の運用ルールを定めて、それに従って作業を進めることこそが大切です。

今回の事例のBさんはGit初心者としましょう。そして、Gitリポジトリを最初に作成したAさんが中央リポジトリを含めた全体の管理者として運用ルールを決めることとします。Aさんがやらなければならないことは、Bさんがコンフリクトに遭遇しない、すなわちBさんのリポジトリをFast Forwardに保つことです。同時に、コンフリクトの解決は、基本的にAさんが行うという取り決めをしておくことも大切です。

そのための運用ルールの一例が、右のようなものです。

このルールによってBさんが中央リポジトリからブランチする際のコンフリクトはほぼ完全に発生しないようにできます。

- (1) Aさんが中央リポジトリにプッシュする前に、「必ず」Bさんが先にプッシュする
- (2) Bさんがプッシュしたあとで、Aさんが中央リポジトリにプッシュする
- (3) Bさんは自分がプッシュしたあと、次の作業を開始する前に「必ず」Aさんがプッシュした内容をブランチする

ただし、現実的にはこのルールを守れないこともあります。「今すぐWebコンテンツを修正しなければならない」など状況によつては、AさんがBさんのプッシュを待たずに、Aさん独自のコミットを中央リポジトリにプッシュしなければならないケースはレアケースではありません。

したがって、Aさんはさらに右のこと注意する必要があります。

- (1) Bさんがプッシュする前にAさんがプッシュしなければならないときは、Bさん側でコンフリクトが発生しないことを確認する。
- (2) Aさんのプッシュによって、Bさんにコンフリクトが発生することが予想される、もしくはコンフリクトが発生するかしないかが判断できない場合は、プッシュと同時にBさんに「コンフリクトが発生するかもしれない」と連絡する。

▶ コンフリクト発生に備える

あ る程度の運用ルールを決めたとして、コンフリクトを完全に封じ込めることはできません。実際、コンフリクトは頻繁に発生すると想定したほうがいいでしょう。

今回の事例でいえば、Git初心者であるBさん側でコンフリクトが発生した場合に問題となることが予想されます。したがって、Bさん側でコンフリクトが発生した備え、事前にコンフリクトの解決はAさんが行うと取り決めておくことが重要です。また、これを踏まえ、Bさんは右のような運用ルールを意識して作業する必要があります。

Gitを利用した共同作業においては、コンフリクトの発生を完全に防ぐことはほぼ不可能と考えたほうがいいでしょう。実際、コンフリクトは結構発生します。より重要なことは、コンフリクトが発生した際に、どのように解決するかをスタッフ間で共有しておくことです。

Column

● ベア(bare)リポジトリとFast Forward

BitbucketやGitHubのようなリポジトリホスティングサービスが提供するリポジトリは、**ベアリポジトリ**として作成されています。ベアリポジトリとは、「作業コピー」ファイルが存在せず、どのブランチもチェックアウト(checkout)されていない(チェックアウトできない)リポジトリです。また、ベアリポジトリ上ではマージを行うこともできないため、手元のリポジトリからプッシュす

る際は常にFast Forwardでなければなりません。ベアリポジトリと手元のリポジトリでコミット履歴に差がある場合にプッシュしてもエラーとなってしまいます。この場合いったんブランチしてマージ(その際、コンフリクトが発生したら解決)して、Fast Forwardな状態にすることでプッシュできるようになります。“プッシュの前にブランチする”癖をつけておくといいでしょう。

- (1) 作業を開始する前に、まずはSourceTreeで中央リポジトリがブランチする

- (2) 手元のローカルリポジトリrepoBに作業内容をコミットする前に、いったん中央リポジトリからブランチして、同期を保つ

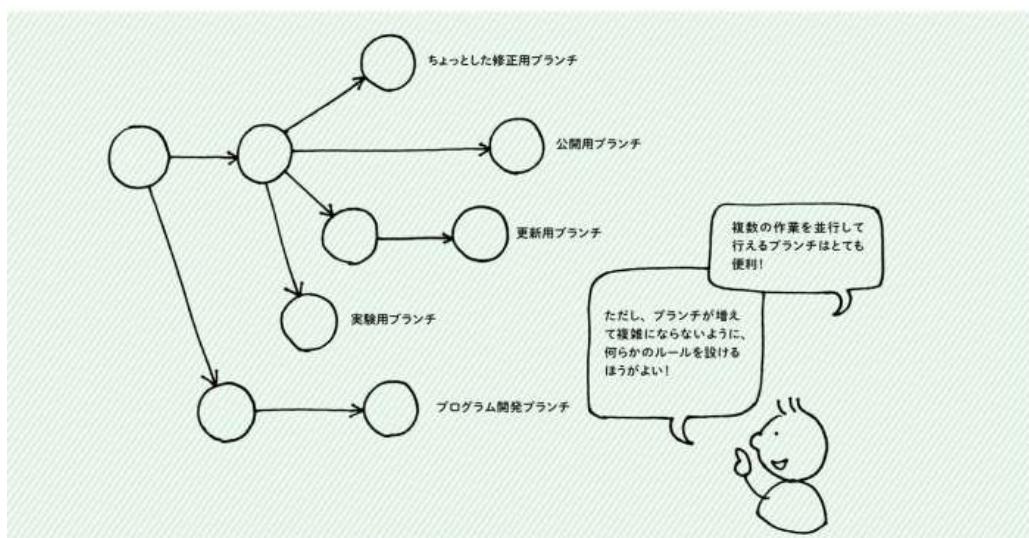
- (3) 中央リポジトリからブランチした際にコンフリクトが発生したら、勝手に判断せず、リポジトリ管理者であるAさんに連絡して、指示を仰ぐ

ブランチでフォルダを管理しよう

Gitを利用する大きなメリットのひとつが「ブランチ」です。

ひとつのフォルダで管理する作業環境で、複数の作業を並行して行うことができます。

ここでは、ブランチの使い方と運用ルールについて紹介します。



- ブランチ機能の使い方（作成、削除、切り替え、マージ）をマスターして複数の作業を同時進行できるようにする
- ブランチ機能を利用したときのコミットツリーの変化を理解する

使用するコマンド

```
$ git branch [作成するブランチ]
$ git branch -D [削除するブランチ]
$ git checkout [切り替え先ブランチ]
$ git checkout -b [作成するブランチ] [既存ブランチ]
$ git merge [ブランチ]
```

使用するSourcetreeの機能



▶ 複数の作業コピーの状態を1つのフォルダで管理できる

Gitを利用する大きなメリットのひとつが「ブランチ」(branch)です。ブランチは「枝」という意味ですが、まさに作業内容を枝分かれさせて、複数の作業を並行して進めることができます。

たとえば、トップページなど主要なページのデザイン・リニューアルが予定されているWebサイトで、制作作業中に現在公開しているサイトの更新作業も同時に併行なければならぬような場合、2つの作業を並行して進めるためには、開発環境を2つ用意するといった対応が必要となります。現在公開中のサイトに対する更新作業のための環境と、リニューアル作業を行うための環境です。このようなケースでは、更新作業内容をリニューアル作業環境にも反映しなければなりません。

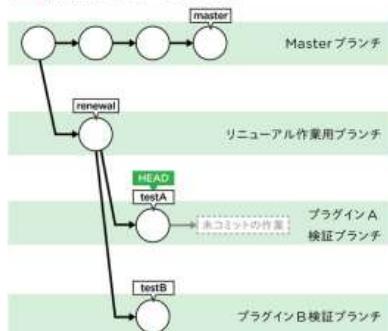
一般的に、こういった2系統の作業環境で進めた場合は、互いの進捗管理が煩雑になり、“先祖返り”といった不具合が発生するなどのリスクをはらんでしまいます。

また、WebサイトにjQueryプラグインなどを利用して

新しい機能を組み込む際、いくつかのプラグインを比較検討したいこともあるでしょう。Gitがない環境であれば、それぞれのプラグインを組み込んだ複数のHTMLファイルを作成して個別にテスト・管理することになると思います。たとえばindex1.htmlとindex2.htmlなど、別々のファイルとして管理することになるでしょう。もちろん、そういう進路でもよいのですが、ブラウザでの表示確認のたびにファイル名を変えるのは面倒です。GitがあればHTMLファイルとしてはindex.htmlに一本化したうえで複数のプラグインを比較検討も簡単にできます。もちろん、現在公開中のバージョンをひとつのブランチとして管理しておけば、安定したバージョンを破壊することなく、新しい機能追加をいろいろと試すことができるわけです。証言

ここでブランチのメリットを簡単にまとめてみます。Web制作作業現場で煩雑になりがちいくつかの作業が、ブランチ機能によって安全化・効率化できることが見えてくるのではないかでしょうか。

図1 複数作業状態をブランチで管理



Gitのブランチを活用することで、複数の並行作業を効率的に行なうことができます。

④ ブランチのメリット

- ・同時に複数の作業の状態を独立して保持して、作業を行うことができる
- ・一旦現在の作業を保留して、横から入った作業を行える
- ・同時に複数の作業を並行したい場合に、リポジトリを複数用意する必要はない、ひとつのリポジトリの中に複数のブランチを作成することで解決できる
- ・過去のある時点の作業中の状態に戻ったり、最新状態に戻すことも自由自在

▶ ブランチの作成と切り替え

では、実際にSourcetreeによるブランチ操作を見てみましょう。

Gitリポジトリを作成した時点では、「master」というブランチがひとつだけ作成されます（図2）。まさに最初に作成されるマスターとなるブランチです。

ブランチは任意のタイミングで作成することができます。Sourcetreeでは、画面上部の「**ブランチ**」ボタンをクリックすると、図3の画面が表示され、「新規ブランチ」タブ画面にて新しいブランチを作成することができます。このとき、現在作業対象となっているブランチ名が画面に表示され、新規に作成するブランチには任意の名前をつけることができます。

実際には、**ブランチ**は任意のコミットを基点として作成することになります。現在作業しているブランチを元に新しいブランチを作成する場合は「**作業コピーの親**」を選択します。また、新規ブランチを作成したあと、すぐに新しいブランチで作業をするのであれば「**新規ブランチを作成してチェックアウト**」を選びましょう。それぞれを設定して「**ブランチを作成**」ボタンをクリックすればブランチが作成されます。

ここで「**チェックアウト**（checkout）」という用語が出てきました。一般的にはホテルなどで勘定を済ませるといった意味ですが、ここでいうチェックアウトとは「図書館での貸し出し手続き」を行なうイメージです。つまり、読んでいた本を戻して、別の本を書架から取り出すようなイメージです。Gitでは、**作業対象となるブランチ（コミット）を切り替える**ときに「**チェックアウト**」といいます（図4）。

▶ ブランチのマージ

ブランチでの作業が完了した場合、そのブランチでの作業を継続することはあまり望ましくありません。本来作業すべき主流のブランチに戻る必要があります。そこで、一時的な作業用として作成したブランチは作業が完了した時点で「**派生元**」となったブランチへマージすることになります。

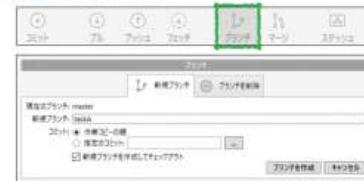
前述の例では、「taskA」という作業用ブランチを作成したため、このリポジトリにはmasterとtaskAという2つのブランチが存在することになります。そしてtaskAで開

図2 リポジトリ作成時点でのブランチ



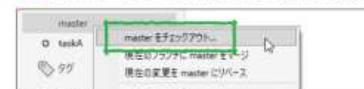
リポジトリを作成すると「master」というブランチが自動的に作成されます。

図3 ブランチを作成する



Sourcetreeの「**ブランチ**」ボタンをクリックして、新しいブランチを作成できます。

図4 チェックアウトして作業対象ブランチを切り替える



新たに作成した「taskA」というブランチから、再び「master」ブランチへ作業対象を切り替えるには、ブランチ一覧に表示されているブランチ名（ここでは master）上で右クリックして表示されるコンテキストメニューから「masterをチェックアウト」を選びます。

示されるので「OK」ボタンをクリックしてマージします（図7）。その結果、taskAで先行していたコミットがmasterに取り込まれ、masterはtaskAと同じコミット段階まで進むことになります（図8）。

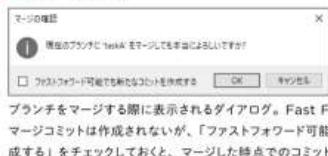
ところで、このとき行われるマージとは、前節までで解説したブルを行った際に発生するマージと同じです。したが

図5 新たなブランチによる作業と元ブランチの状態



taskAというブランチを作成して作業を進めると、masterよりもtaskAが先行した状態となります。

図7 マージの確認



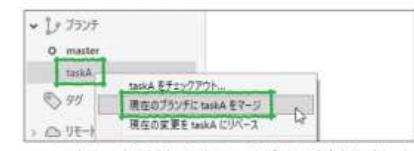
ブランチをマージする際に表示されるダイアログ。Fast Forwardの場合、基本的にマージコミットは作成されないが、「Fast Forward可能な場合も新たなコミットを作成する」をチェックしておくと、マージした時点でのコミットが自動的に作成されます。

図8 ブランチのマージを行った結果



って、仮にtaskAでの作業と並行してmasterでも作業を行っていた場合、マージによるコンフリクトが発生する可能性があります。また、taskAで作業している間、masterで何も作業していない場合は、masterにあるコミットはすべてtaskAに存在しているためFast Forwardとなります。

図6 ブランチのマージ



masterをチェックアウトしてから、taskAブランチを右クリックしてコンテキストメニューから「現在のブランチにtaskAをマージ」（Macは「taskAをmasterにマージ」）を選んでマージ作業を行います。



Macの場合は文言が少し異なる

ブランチのマージを行った結果、taskAのコミットがmasterにも取り込まれ、マージ時点でのコミットが描かれています。

▶ ブランチの削除と復元

作業が完了して元のブランチにマージしたブランチは、そのまま置いておくのはあまり意味がありません。むしろ作業のために派生させたブランチを置いたままにしておくと、ブランチがどんどん増えてしまい、作業環境全体の見通しが悪くなる可能性もあります。

特にSourcetreeのようにグラフィカルに表示されるインターフェイスではブランチの数が増えることで、ブランチごとに作業が完了しているのかどうか、見た目での判断がつきにくくなってしまいます。そこで、一旦作業が完了したブランチは削除するようにしましょう。

Sourcetreeでブランチを削除する場合、ブランチの作成と同様に画面上部の「**ブランチ**」をクリックし、表示されるダイアログで「**ブランチを削除**」タブ画面に切り替えます（図9）。「**ブランチを削除**」タブ画面には、現在リポジトリに存在しているブランチがリストアップされます。削除したいブランチをチェックして「**ブランチを削除**」ボタンをクリックすれば削除できます。

じりに存在しているブランチがリストアップされます。削除したいブランチをチェックして「**ブランチを削除**」ボタンをクリックすればブランチを削除できます。

図9 ブランチを削除



「**ブランチを削除**」タブ画面には、現在リポジトリに存在しているブランチがリストアップされます。削除したいブランチをチェックして「**ブランチを削除**」ボタンをクリックすれば削除できます。

ブランチを削除したとしても、そのブランチで行ったコミットが削除されるわけではありません。マージすることによってtaskAで行ったコミットはmasterブランチに取り込まれているため、taskAで行った任意のコミットに戻すことができます。

taskAブランチですべき作業が残っていたにも関わらずtaskAを削除してしまったなど、誤ってブランチを削除したとしても、元のブランチ（ここではmaster）へマージを行っていれば、任意コミットを選んでブランチを作成することで、taskAでの作業を再開することもできます。Sourcetreeでは、新規ブランチの作成画面で「指定のコミット」の右にある「...」ボタンをクリックするとコミット一覧が表示されるので、作業を再開したいコミットを選んで「ブランチを作成」をクリックします図10。

▶ ブランチの運用にはルールが必要

ブランチは大変便利な機能ですが、複数人で作業しているときに、各スタッフがさまざまにブランチを作成してしまったり、中央リポジトリの管理状態が煩雑化してしまい、やがて整理がつかない状況に陥ります。つまり、ブランチの運用に際しては一定のルールを設けることが重要になります。

ブランチの運用ルールは、実際には作業チームごとに独自のルールを策定することになると思いますが、基本となるセオリーが考えられます。特にGit初心者であれば、右に挙げるようなセオリーを意識してブランチを運用するといいでしょう。

実際のところ、ブランチの運用はGitを利用するユーザによってさまざまです。Gitを使いながら、自分に最適な運用ルールを考えていく必要があります。ただし、Gitにはより高度なブランチ運用を行える「Git Flow」や「GitHub Flow」といった体系化された方法論も構築されています。本書でGitの基本を学んだあと、より効率的なブランチ運用を考える際、これらの体系化されたワークフローが参考になるでしょう。

図10 ブランチの復元



新規ブランチ作成画面で、作業を再開したいコミットIDを入力して作成すると、そのコミットから派生するブランチを作成できます。なお、指定できるコミットIDは過去のコミットだけでなく、参照されなくなったコミットも含まれます。

④ ブランチ運用のセオリー

・幹となるブランチを決める

慣習としては、リポジトリを作成した際に生成されるmasterを幹としています。

・ブランチは「幹となるブランチ」から派生させる

masterブランチを幹として、ブランチを作成する場合はmasterを元として作成するようにします。

・ブランチ作業が完了したら元ブランチに統合(マージ)する

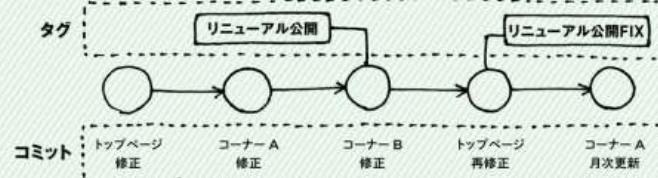
ブランチでの作業が完了した際は、必ず元ブランチ（ここではmaster）に統合（マージ）するようにします。

3-08

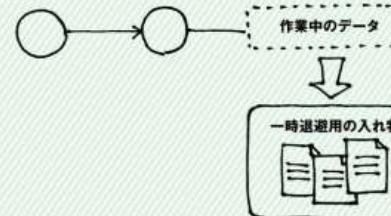
タグやスタッッシュを使いこなそう

Gitを利用した開発作業で使いこなしたいのが「タグ」と「スタッッシュ」です。復元ポイントを指定したり、現時点での作業内容を一時的に退避するなど、実際の運用上ではとても便利な機能です。

タグ



スタッッシュ



- ✓ 中央リポジトリ上のブランチの操作方法
- ✓ タグ機能の使い方とブランチ機能との使い分けを覚える
- ✓ スタッシュ機能を使って未コミットの作業を退避させる

使用するコマンド

```
$ git push [リモートリポジトリ] [ローカルリポジトリのブランチ]:[リモートリポジトリに作成するブランチ]
$ git push [リモートリポジトリ] :[削除するブランチ]
$ git tag [タグ]
$ git stash
$ git stash apply
```

使用するSourcetreeの機能



▶ 中央リポジトリに新たなブランチを作成

複 数人で作業を行う際、各スタッフの作業状況を共有するためのハブとなるのが中央リポジトリです。Bitbucketなどに作成する中央リポジトリではブランチを作成することはできません。ブランチはあくまでもローカルにあるGit上で作成することができます。ただ、複数人の作業では、任意の作業用ブランチを共有したいケースもあります。

たとえばAさん、BさんのふたりでWebサイト制作を行う際、現時点での公開サイトの更新作業と並行して、リニューアル作業も進めるといった場合です。つまり、更新作業用のブランチとリニューアル用のブランチの2つを用意して、それぞれを各作業スタッフの「幹」として運用できるわけです。

Sourcetreeを使って中央リポジトリに新たなブランチを作成したい場合は、Aさんがプッシュする際に中央リポジトリ（リモートリポジトリ）にmasterブランチの複製となるブランチ名を指定することで新規ブランチを作成することができます【図1】。

図1 中央リポジトリに新規ブランチを作成



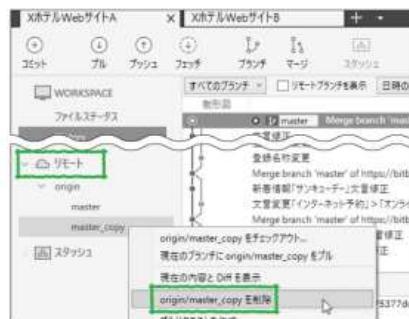
中央リポジトリに新たなブランチを作成したい場合は、Sourcetreeのブッシュ画面で「リモートブランチ」に新しいブランチ名を指定してプッシュします。

▶ 中央リポジトリのブランチを削除

中 央リポジトリに新たなブランチを作成して作業を進め、その作業が完了した場合は作業用ブランチは削除してリポジトリの状態を整理したくなるでしょう。

Sourcetreeでは、中央リポジトリのブランチ状態がブランチリストの「リモート」欄に表示されます。この欄で削除したいブランチを右クリックして表示されるコンテキストメニューで「～（ブランチ名）を削除」（Macは「削除」）を選択します。すると「ブランチ削除の確認」ダイアログが表示され、「OK」ボタンをクリックすれば中央リポジトリ上のブランチを削除することができます【図2】。

図2 中央リポジトリのブランチを削除



ブランチリストで削除したいブランチを右クリックして表示されるコンテキストメニューで「～（ブランチ名）を削除」（Macは「削除」）を選んで「OK」ボタンをクリックすれば、中央リポジトリ上のブランチを削除できます。「ブランチ削除の確認」ダイアログを見ると「リモートのブランチも削除されます（削除がプッシュされます）」とあるように、ブランチの削除からプッシュまでの一連の作業を自動的に行ってくれます。

▶ 中央リポジトリのブランチを作業対象にする

中 央リポジトリに新規ブランチを作成した時点では、実際にはローカルリポジトリ上の作業用リポジトリとして作成されているわけではありません。あくまでも中央リポジトリ上にブランチが作成されただけです。

新たに作成した中央リポジトリ上のブランチを、ローカルリポジトリ上でも作業対象としたい場合は、そのブランチをチェックアウトすることで、ローカル用のブランチを生成して作業用のブランチを作成する必要があります。

Sourcetreeでは、ブランチリストにあるリモートブランチ名を右クリックして表示されるコンテキストメニューで「origin/（ブランチ名）をチェックアウト」（Macは「チェックアウト」）を選びます【図3】。すると「チェックアウト」ダイアログの「新規ブランチを作成してチェックアウト」タブ画面が表示されます【図4】。ローカルで作業するブランチ名を指定し、「ローカルブランチでリモートブランチを追跡する」がチェックされていることを確認して「OK」ボタンをクリックすればブランチが作成されます。

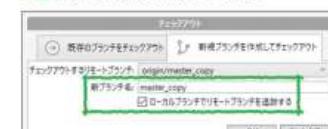
リモートのブランチをそのままローカルに持ってきてたい場合は、ブルではなくチェックアウトであることに注意しましょう。ローカルに存在しないリモートブランチをマージしたい場合はブル、そのまま切り替えるときはチェックアウトするということです。ブルした場合は、現在ローカルリポジトリでチェックアウトしているブランチに対してマージされますので、意図的であるならば別として不要なマージ作業を行ってしまう可能性があるので要注意です。

図3 ローカルに存在しないリモートリポジトリをチェックアウト



ローカルに存在しないリモートリポジトリをチェックアウトする場合は、ブランチリストにあるリモートブランチ名を右クリックして表示されるコンテキストメニューで「origin/（ブランチ名）をチェックアウト」を選びます。

図4 中央リポジトリのブランチを追跡



「チェックアウト」ダイアログの「新規ブランチを作成してチェックアウト」タブ画面で、「ローカルブランチでリモートブランチを追跡する」がチェックされていることを確認して「OK」ボタンをクリックします。

▶ コミットに目印をつけておきたい場合

ブ ランチはさまざまな応用が考えられます。たとえば一定の作業を完了して安定バージョンとして完成了ものを別のブランチとして取っておく……というような使い方はあまり適切ではありません。復元ポイントとしてバックアップを作成したい、という目的であれば、ブランチではなく「タグ」のほうが適切です。ブランチにはコミットを追加できてしまいますが、タグはコミットを許可しないため、それ以上派生できないという特徴があります。単にコミットに目印をつけたいという目的であれば、その段階から独自に派生して状態が変わることは望ましくありません。

ブランチを作成して取っておくということもできないわけではありませんが、ブランチが氾濫することになるので、使用する機能は適切に選ぶようにしましょう。

では、実際にタグの機能を利用して復元ポイントを作成して、ブランチとして復元する手順を紹介しましょう。

コミットツリーで復元ポイントにしたいコミットを右クリックして表示されるコンテキストメニューから「タグ」を選ぶと、「タグ」ダイアログの「タグを追加」タブ画面が表示されます（次ページ 図5 図6）。ここでタグ名を指定して「タグを追加」ボタンをクリックすればタグが作成できます。

図5 コンテキストメニューで「タグ」を選択



図6 「タグを追加」タブ画面でタグに名前をつける



タグに名前を付ける際、「プッシュするタグ」をチェックしておくと中央リポジトリにもタグを共有することができます。

作成したタグは、コミットツリーにタグマークとタグ名が表示されるようになります(図7)。これによって、復元ポイントが適確にわかるようになります。では、このタグを復元ポイントとして新たに作業するブランチを作成してみましょう。

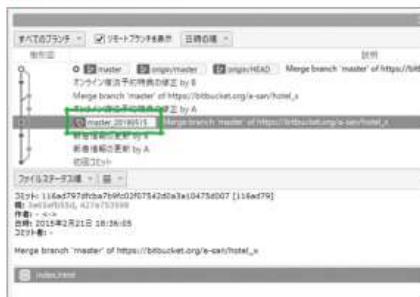
先述と同様に「ブランチ」ボタンをクリックして「新規ブランチ」画面を表示します(図8)。先ほどとは違い、この画面で「指定のコミット」をチェックして「...」ボタンをクリックします。すると「コミットの選択」画面が表示されるので、復元ポイントにしたいタグを選びます(図9)。

図7 コミットツリーでタグを確認



作成したタグは、コミットツリーにタグマークとタグ名が表示されます。

図9 「コミットを選択」画面



「コミットを選択」画面でタグを選択します。

コミットツリーの左側に表示されているタグの一覧で右クリックして表示されるコンテキストメニューにも「～をチェックアウト」というメニューがありますが、ブランチは作成できないので注意してください(図10)。なお、印目として不要になったタグは、このコンテキストメニューにある「～を削除」で削除することができます(図11)。

タグは特別な操作がそれほどありません。コミットIDに特別な印目をつけるための機能であることを覚えておきましょう。

図8 「ブランチ」の「新規ブランチ」タブ画面



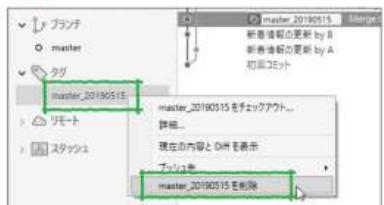
「新規ブランチ」タブ画面で「指定のコミット」を選び、「...」ボタンをクリックします。

図10 タグ一覧のコンテキストメニュー



タグ一覧で右クリックして表示されるコンテキストメニューでタグをチェックアウトしても新たなブランチは作成できません。

図11 タグの削除



タグ一覧で右クリックして表示されるコンテキストメニューからタグを削除することができます。

▶ 作業着手前の状態から、ほかの作業を行いたい場合

何

らかの理由で作業を一時中断し、作業着手前の状態に戻し、ほかの割り込み作業を行いたい、といった場合があります。このようなケースでは、ブランチではなく「スタッッシュ」(stash)を使います。

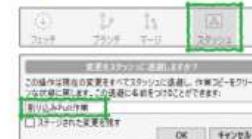
スタッッシュは、作業がまだ中途半端な状態でコミットは行いたくないようなケースで、一時的に作業状態を保管しておくための機能です。

割り込み作業を行うために、現在中途半端な状態のブランチでコミットして中断するというのはあまりよくありません。中途半端な状態のコミット、言ってみれば不要なコミットを作成してしまうと混乱を生じるもとになります。かといって、作業途中のままコミットを行わざにいると、ブルやプッシュなどの操作も行いにくくなってしまいます。そこで、スタッッシュを利用して現在の作業途中の状態(差分)を一時的に退避させて、割り込み作業を開始するわけです。

Sourcetreeでは、画面上部の「スタッッシュ」ボタンをクリックしてスタッッシュを作成します(図12)。このとき、名前をつけて名前をつけてスタッッシュを作成することができますが、あとで戻す際に内容がわかるような名前をつけておくほうがよいでしょう。スタッッシュを作成することで、作業を開始する前のクリーンな状態に戻りますので、ブルやチェックアウトといった作業も問題なく行えるようになります。

割り込み作業が完了したら、再び中断する前の状態に戻すこともできます。作成したスタッッシュはSourcetreeのコミットツリー左側にある「スタッッシュ」リストに表示されています。これを右クリックして表示されるコンテキストメニューから「～を適用」(Macは「退避した変更を適用」)を選びましょう(図13)。続けて表示される確認ダイアログで「OK」ボタンをクリックすれば、中断前の状態に戻ります(図14)。

図12 スタッッシュの作成



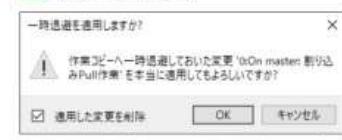
「スタッッシュ」ボタンをクリックして、中断内容がわかるような名前をつけてスタッッシュを作成します。

図13 スタッッシュを適用



スタッッシュリストでスタッッシュ名を右クリックして表示されるコンテキストメニューから「～を適用」(Macは「待避した変更を適用」)を選びます。

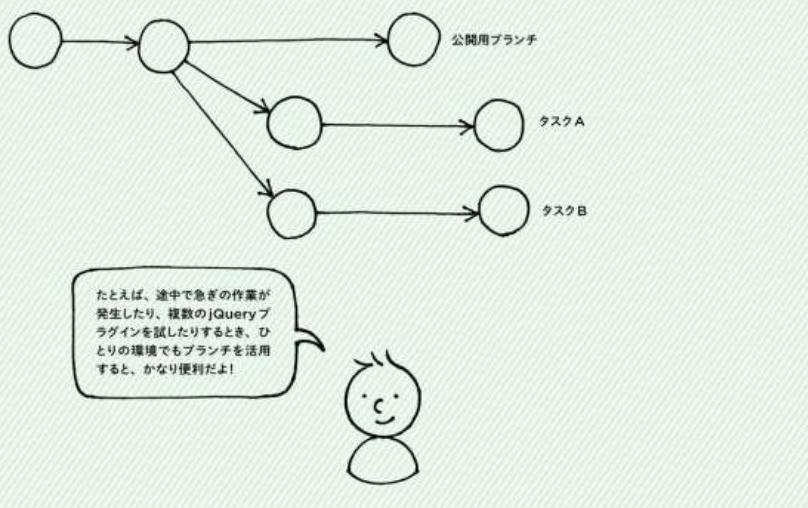
図14 スタッッシュ適用の確認



確認ダイアログで「適用した変更を削除」(Macは「適用後に削除」)をチェックしておくと、適用と同時にスタッッシュを削除できます。スタッッシュを残しておくと、同じ変更を二重に加えてしまう恐れがあるので不要なスタッッシュは削除するようにしましょう。

ひとりでブランチを活用してみる

Gitにおいてはブランチをうまく使えるようになることはかなり重要です。ここでは、あえてひとりでブランチ運用をしてみることで、複数スタッフでの共同作業に向けて、ブランチを使って運用する練習をしてみましょう。



- ブランチ機能を利用して2つの作業を同時進行するワークフロー
- スタッシュ機能を利用して、作業中の差分をコミットせずにマージする手順を学ぶ
- ブランチとHEADの関係を理解する

使用するコマンド	使用するSourcetreeの機能
<pre>\$ git branch [作成するブランチ] \$ git checkout [切り替えるブランチ] \$ git stash \$ git stash apply \$ git merge [ブランチ]</pre>	



▶ 2つの作業を同時進行する

前節でブランチの基本的な考え方が理解できたでしょうか。ここでは、実際の制作現場においてブランチをどのように活用するのか、事例を示しながら紹介します。

まずは、ひとりでブランチを活用する事例です。再びWeb制作を担当しているAさん、Bさんに登場してもらいます。Bさんは、今日中に終わらせなければならない作業(taskA)を抱えているとします。しかし、その作業を開始するにあたって必要なデータを夕方まで待たなければならなくなってしまいました。つまりAさんからの連絡待ちという状況です。また、Bさんには、今日中ではないけれど、もうひとつ別の作業(taskB)もあります。そこでBさんは、Aさんの連絡を待っている間にtaskBを進めておくことにしました。

こういった状況では、ブランチを作成することで複数の作業をスムーズに同時進行させることができます。実際に見てみましょう。

現在、Bさんのローカルリポジトリは、前回の作業が完了してコミットした状態でmasterブランチのみがあり、最終コミットがチェックアウトされています(図1)。

この状態で、まずはtaskAとtaskB、それぞれの作業用ブランチを作成します。Sourcetree画面上部の「ブランチ」をクリックして新規ブランチを作成しましょう。いずれも「作業コピーの親」つまりmasterをもとにした作業用ブランチを作成します(図2)。

これでSourcetreeのコミットツリー画面にmaster、taskA、taskBの3つのブランチが並びます。

Bさんは、まずtaskBの作業をするため、taskBブランチをチェックアウトして作業対象とします。taskBを右クリックして表示されるコンテキストメニューで「taskBをチェックアウト...」を選ぶと、コミットツリー上でtaskBにヘッドが移行します(次ページ図3 図4)。この状態でBさんはtaskBの作業を開始します(図5)。

図1 BさんのローカルリポジトリrepoBの状態

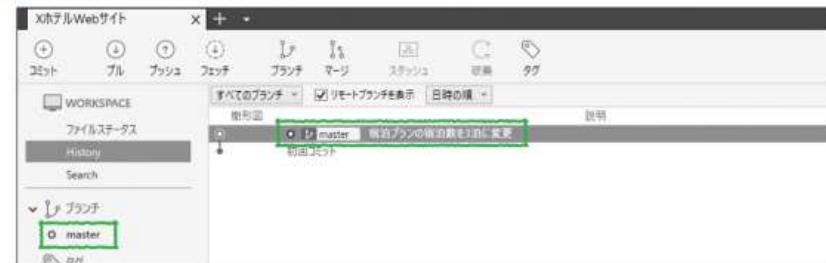


図2 taskA、taskBの新規ブランチ作成

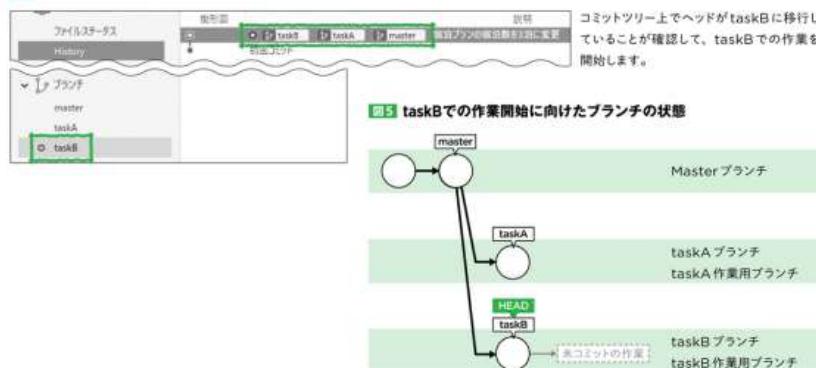


「ブランチ」ボタンをクリックして、taskA、taskB、それぞれの作業用ブランチを作成します。

図3 taskB用プランチをチェックアウト



図4 taskB用プランチでの作業準備完了



Column

○ HEADとは

プランチの解説では「HEAD」(ヘッド)という用語が登場します。HEADは、音楽プレイヤーの再生ヘッドをイメージしてください。CDなどに記録された音源データ（レコード）のどこを現在再生中ののかを指示するものです。つまりHEADがある部分が現在再生中（作業対象）ということになります。HEADは作業対象となっているプランチを指示するポイントで、チェックアウトす

ることでHEADをほかのプランチに移動することができるので、したがって、HEADが指示するプランチの状態が、実際の作業ファイルの状態となります。複数プランチがある状態でも、作業対象となるのはたったひとつのプランチです。現在作業対象となっているプランチを意識してGitを活用するようにしましょう。

▶ 2つの作業を切り替える

BさんがtaskBの作業を行っている途中で、AさんからtaskA用のデータが届き、taskAの作業に着手するよう指示が来ました。Bさんは、taskBの作業がまだ中途半端な状態ですが、taskAの作業に切り替えなければなりません。taskBはまだ作業途中でコミットできる状態ではありません。そこで前節で説明したように、BさんはtaskBの作業内容をスタッッシュして一時退避することになります（図6）。

現在、HEADはtaskBのままで。スタッッシュする前は、作業中のデータが作業ツリーのファイルに残っています。

り、コミットツリーには「コミットされていない変更があります」と表示されていました。スタッッシュすることで、作業途中のデータが一時的に隠され、コミットツリーも作業開始前のクリーンな状態に戻ります（図7）。

これで問題なく、taskAの作業に切り替えることができます。プランチリストでtaskAを右クリックして表示されるコンテキストメニューから「taskAをチェックアウト」を選択します（次ページ図8）。これでtaskAでの作業を開始する準備が整いました（図9、図10）。

図6 taskBの作業内容をスタッッシュする

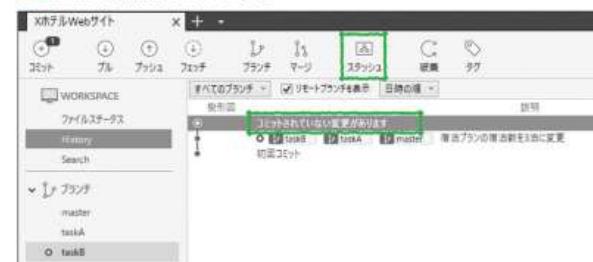


図7 taskBの作業内容をスタッッシュした結果



↓

スタッッシュする前はコミットツリーには「コミットされていない変更があります」と表示されていますが、スタッッシュすることで作業中のデータが一時的に隠され、コミットツリーは作業開始前のクリーンな状態に戻ります。

図8 taskAをチェックアウトして作業対象にする

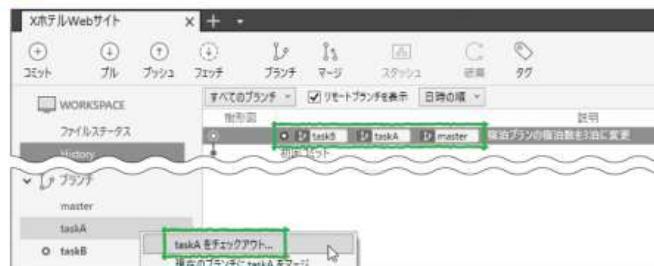
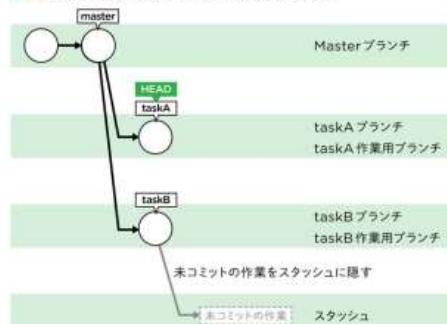


図9 taskAにHEADが移動していることを確認



図10 taskAでの作業開始に向けたプランチの状態



Column

○ コミットもスタッッシュもしないでプランチを切り替えると?

作業途中のデータが残っている状態でプランチを切り替える必要が発生した際、コミットもスタッッシュもせずに、プランチを切り替えることもできます。ただし、その場合は、作業途中のデータ（作業コピーの状態）がそのままHEADが移行したプランチに引き継がれてしま

います。そのまま意識せずにコミットしてしまうと、本来コミットすべきプランチではないプランチに対して作業内容をコミットしてしまうことになります。プランチを切り替える際は、作業途中のデータをコミットもしくはスタッッシュしてから切り替えるように注意しましょう。

▶ taskAの作業を完了して、masterにマージする

BさんはtaskAでの作業を完了したら、taskBの作業に戻ろうとします。このとき、taskAの作業状態をそのまま放置するわけにはいきません図11。taskAの作業内容をコミットして、taskAプランチでの作業をGit内で完了する必要があります。

まずは、taskAでの作業内容をtaskAプランチのままコミットします図12。これにより、taskAプランチは、master、taskBプランチに対して1段階先行した状態になります。

さらに、taskAをmasterに統合（マージ）してtaskAで行ったコミットをmasterに反映するようにします。

そのためには、まずmasterをチェックアウトします。ブ

ランチリストでmasterを右クリックして表示されるコンテキストメニューで「masterをチェックアウト」です図13。

プランチがmasterに切り替わったことを確認したら、SourceTree画面上部の「マージ」を選び図14、マージする相手としてtaskAを選んでマージします図15。このとき、taskAにはmasterに存在するすべてのコミットが含まれているためFast Forwardとなり、問題なくプランチの統合が完了します（次ページ図16）。このあと、taskAでの作業内容がレビュー・承認を受け、これ以上作業する必要がない場合は、taskAプランチは削除して整理してしまいましょう図17。

図11 taskAでの作業が完了した状態



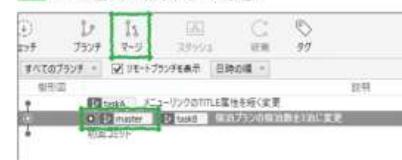
図12 taskAプランチでコミット



図13 masterをチェックアウト



図14 マージ前のコミットツリーの状態



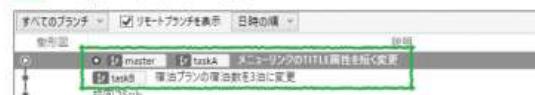
masterがチェックアウトされている状態で「マージ」ボタンをクリック。

図15 masterにtaskAをマージ



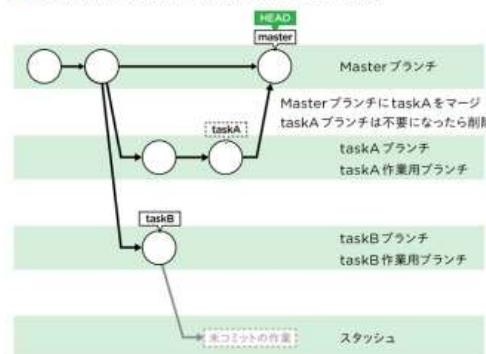
表示される選択画面でtaskAプランチを指定します。

図16 マージ後のコミットツリーの状態



マージ前はtaskAが1段階先行していましたが、マージすることでmasterがtaskAに追いつきました。一方、taskBはまだ1段階手前になります。

図17 taskBからtaskAに切り替えるときのプランチの状態



▶ taskBの作業に戻る

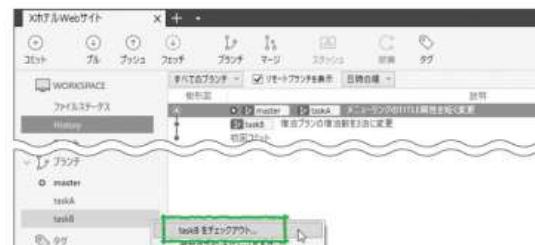
taskAでの作業が終わりmasterへの統合まで完了したBさんは、再びtaskBの作業に戻ります。まずはtaskBをチェックアウトして、再び作業対象プランチにします（図18）。

ただし、taskBをチェックアウトしても、先に作業してい

た内容はスタッッシュされているため、taskBの状態は作業前の何もしていない状態のままでです。

そこで、スタッッシュに保存した作業内容を再びtaskBに適用して作業状態を復活しなければなりません。

図18 taskBをチェックアウト



そのためにはスタッッシュリストから、先ほど保存したスタッッシュを右クリックして表示されるコンテキストメニューで「～を適用」（Macは「待避した変更を適用」）を選びます（図19）。確認ダイアログが表示されるので、「OK」ボタンを

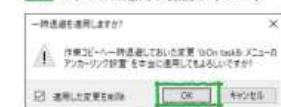
クリックしましょう（図20）。「作業ツリーのファイル」（Macは「ステージングに未登録のファイル」）を見ると、再び作業途中のデータが存在していることがわかります（図21）。これでBさんはtaskBでの作業を再開することができます（図22）。

図19 taskBプランチにスタッッシュを適用



スタッッシュリストでスタッッシュを右クリックして表示されるコンテキストメニューで「～を適用」を選択します。

図20 スタッッシュ適用の確認ダイアログ



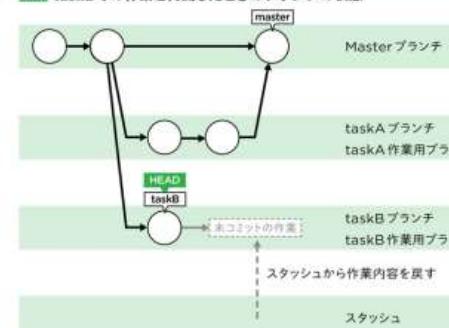
スタッッシュ名を確認して「OK」ボタンをクリックします。

図21 スタッッシュ適用後のコミットツリーの状態



taskBプランチが最初に作業していた状態に戻っていることが確認できます。

図22 taskBでの作業を再開したときのプランチの状態



▶ taskBでのコミットとmasterへの統合(マージ)

のあと、BさんはtaskBでの作業を継続します。taskBでの作業が完了した場合、まずはtaskBプランチでコミットを作成することになります。そのあと、taskBをmasterに統合（マージ）することで、Bさんのリポジトリでの作業を完了することができます。

ただし、taskBをmasterへ統合する場合、Fast Forwardにはなりません。masterへすでにtaskAのコミットがマージされており、taskBよりも先行した状態にあるからです。この状態でtaskBをmasterへ統合し

た場合、必ずマージが発生します。

また、マージが発生するということは、コンフリクトが発生する可能性もあります。たとえば、CSSファイルの同じ行を編集していたような場合です。ただ、ここでの作業はBさんひとりでの作業です。したがって、コンフリクトが発生したとしても、Bさんが独自に判断することができます。

最終的には、taskBの作業内容もmasterへ統合し、その後中央リポジトリへプッシュすることで、Bさんの作業が完了することになります。

3-10

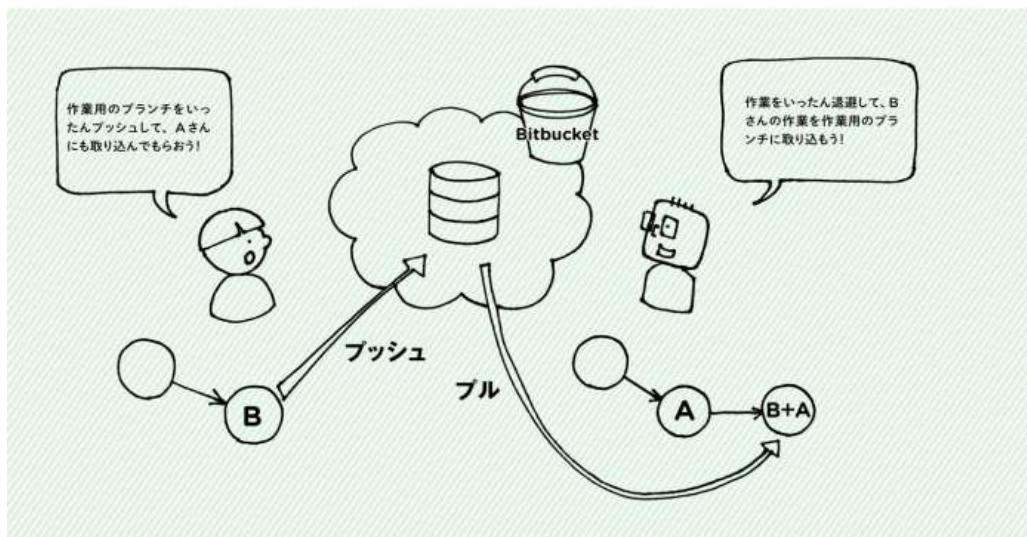
Chapter 3
複数メンバーでの運用

複数人でブランチを運用する

共同作業におけるブランチ運用には、ある程度の運用ルールが必要です。

本章の終わりに、複数スタッフによる共同作業における

ブランチ運用の一例を紹介しておきます。



- ✓ 実用的なリモートブランチの利用例を紹介
- ✓ 複数のメンバーで複数作業を同時に進行するワークフローを理解する
- ✓ 中央リポジトリのブランチをmasterにマージせずに相互の編集内容を共有する

使うコマンド

```
$ git branch [作成するブランチ]  
$ git merge [ブランチ]  
$ git push [リモートリポジトリ] [ローカルリポジトリのブランチ]:[リモートリポジトリに作成するブランチ]  
$ git stash  
$ git stash apply
```

使うSourcetreeの機能



▶ 2つの作業を同時進行する

前節では、ひとりで複数作業を並行して行う場合のブランチ運用事例を紹介しました。ここでは、複数人で作業する場合のブランチ運用事例を紹介します。

実際の制作現場では、複数のスタッフがそれぞれ個別の作業をしている場合があります。たとえば、AさんはWebサイト全体にかかる修正作業(taskA)を担当。一方、Bさんは日々発生する軽微な更新・修正作業を行っており、その作業はその日中にに対応しなければならないような緊急性のある業務を担当しているといったケースです。お互いに作業内容のリリース時期が異なっているわけです。

ただし、Bさんの日々の修正作業においては、Aさんの全般的な修正作業への影響もありえるため、Aさん、Bさんのふたりで連携をとる必要があります。

ひとりでの作業よりも多少複雑にはなりますが、このようなケースでもGitのブランチ機能を活用することで連携作業をスムーズに進行することが可能です。

まずは、ふたりで作業を開始する前段階でのコミットツリーの状態を考えてみましょう。

ふたりともにローカルリポジトリ上で個別に作業はしますが、基本的にはハブとなる中央リポジトリを介してmasterの状態は同期することが前提となります。つまり、ローカルでの作業が一段落した時点でローカルの作業ブランチをmasterに統合(マージ)し、中央リポジトリへプッシュしておく。同時に何か作業する際は、作業開始前に中央リポジトリからプルして同期を図るというのが基本ルールです。

ふたりともに新たに作業を開始する時点での中央リポジトリを反映したローカルリポジトリは同等であると想定します(図1)。この状態で、Aさん、Bさんそれぞれが、ローカルリポジトリ上に作業用ブランチを作成し、チェックアウトして作業を開始します(図2、次ページ図3)。

図1 Aさん、Bさんの現状のリポジトリ状態



中央リポジトリを介して同期しているAさん、Bさんの現状のリポジトリ状態。

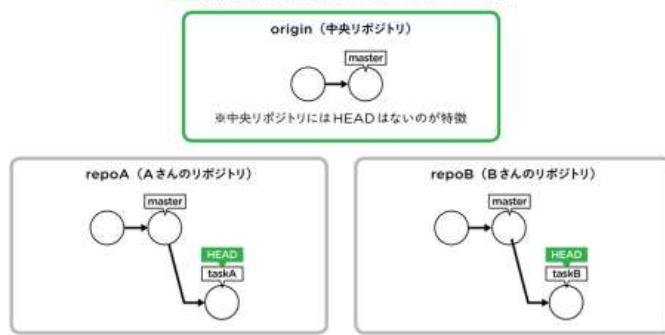
図2 Aさん、Bさんはそれぞれのローカルリポジトリに作業用ブランチを作成



Aさんは自分のローカルリポジトリでtaskAブランチを作成。

Bさんは自分のローカルリポジトリでtaskBブランチを作成。

図3 作業開始に向けた各リポジトリのブランチの状態



▶ 作業途中の段階でのプッシュ作業

Bさんは、大方の作業が一段落したので、いったん作業状態をコミットしてプッシュします (図4 図5)。

ただし、まだ完成したわけではないので、taskBと いう作業用ブランチでプッシュします (図6 図7)。同時に、Aさんの手元の repoA にも作業内容を反映してもらうように通知します。

図4 repoBでindex.htmlを追加(add)



図5 repoBでコミット

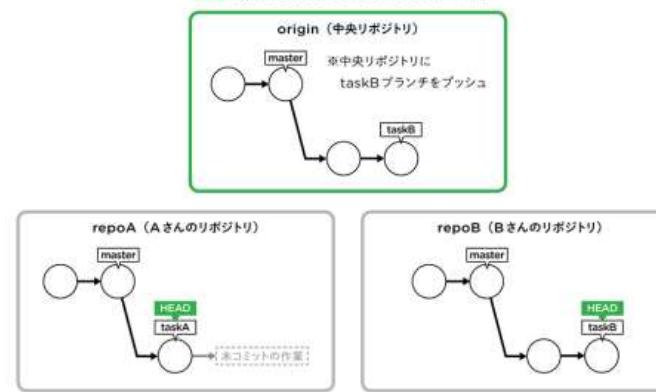


図6 repoBのtaskBブランチを中央リポジトリへプッシュ



BさんはtaskBブランチを中央リポジトリへプッシュします。その際、リモートブランチ側には同じ「taskB」というブランチ名を指定し、「追跡中」をチェックしてプッシュすることで、中央リポジトリに「origin/taskB」という新たなブランチが作成されます。

図7 repoBと中央リポジトリのブランチの状態

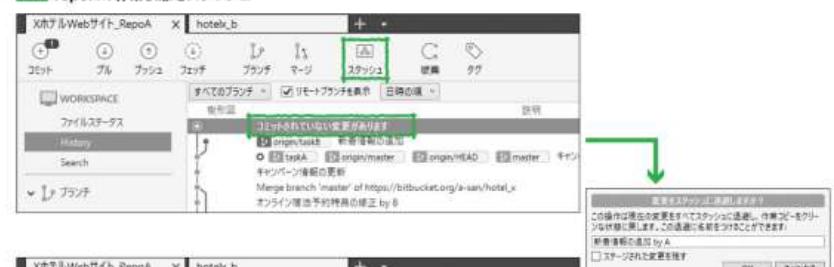


▶ 作業途中の段階でのプル作業

AさんはBさんからの通知を受けて、中央リポジトリから手元の repoA にプルをしようとします。ただし、現時点でのAさんの作業は中途半端で、まだ手元の repoA 上でコミットする状況にはありません。

そこで、Aさんは作業内容をいったんスタッッシュして一時退避し (図8)、taskBをプルしてから作業を再開します (次ページ 図9)。

図8 repoAの作業状態をスタッッシュ



Aさんは現在の作業状態をスタッッシュして、repoAをブレーンな状態に戻します。

図9 taskBをプル



▶ 作業再開のためのスタッッシュ適用

Aさんは問題なくプルできたtaskAプランチで作業を再開するため、一時退避していたスタッッシュを適用します。実際にスタッッシュを適用するとエラーが表示されますが、スタッッシュの適用は正常に行われ、作業コピーに先ほどまで作業していたファイルが復活します。

index.htmlというファイルの同じ行を編集していたために「コンフリクトが発生した」という警告です。エラーは発生しますが、スタッッシュの適用は正常に行われ、作業コピーに先ほどまで作業していたファイルが復活します。

図10 taskAプランチにスタッッシュを適用



図11 コンフリクトエラーが発生



Aさんは（場合によってはBさんと協議して）index.htmlの記述の違いを確認してコンフリクトを解決したうえで、作業を再開します図12。

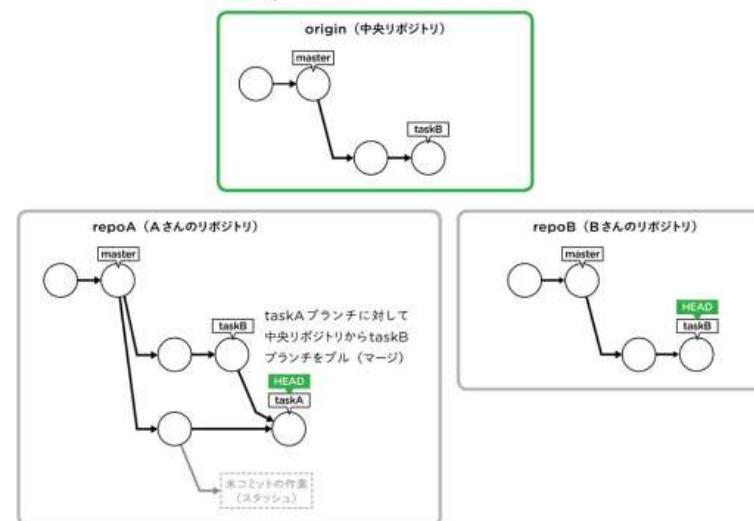
ここまででの作業で、AさんのローカルリポジトリrepoA

では、Bさんの作業内容を取り込むことなく、作業用のtaskAプランチに取り込んだ状態で作業を進めることができます図13。

図12 作業再開時のプランチツリー



図13 repoAと中央リポジトリのプランチの状態



▶ 作業完了時のブランチ削除

Bさんは、必要な作業をすべてやり終えたため、taskB ブランチの作業内容を最終コミットしたあと、master ブランチに統合（マージ）し、最後に中央リポジトリにプッシュします。

まずは、taskB で最終コミットを行います図14。次に master ブランチをチェックアウトして、master ブランチに taskB ブランチを統合（マージ）します図15。このときのマージは、taskB での作業開始後、master には何の変更もありませんので Fast Forward となります。

続けて、Bさんは中央リポジトリへ master ブランチをプッシュして本日の作業を終了します図16。

実際にはプッシュの前にブルをして、中央リポジトリの master ブランチが先行していないかどうかを確認する作業が必要です。Bさんが作業している間に、Aさんがプッシュしている可能性があるからです。いずれにしろ中央リポジトリの変更内容がローカルリポジトリに取り込まれている状態、すなわちFast Forward で中央リポジトリへプッシュが完了します。

図14 taskBでの最終コミット



図15 taskB ブランチを master ブランチに統合（マージ）

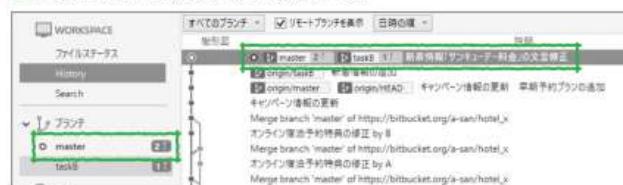


図16 repoBのmaster ブランチを中央リポジトリへプッシュ



さて、taskB での作業はすべて完了したので、もう taskB ブランチは必要ありません。役目終わったブランチは削除して後片づけをしておきましょう。

Sourcetree 画面上部の「ブランチ」をクリックして、「ブランチを削除」タブ画面に切り替えます。この画面で「taskB」と同時に中央リポジトリにひも付けられた「origin/taskB」の両方をチェックして「ブランチを削除」ボタンをクリックします図17。

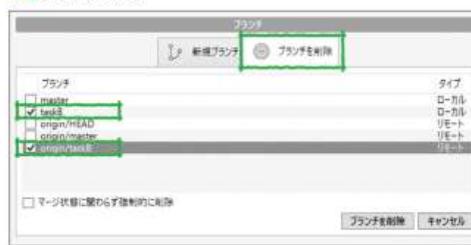
確認ダイアログが表示されるので、さらに「OK」ボタンをクリックすると、repoB 上の作業用ブランチ（taskB）が削除されると同時に中央リポジトリ上の作業用ブランチ

(origin/taskB) も削除されます図18。

このあと、さらに Aさんの作業が残ってはいますが、基本的に Bさんと同様に作業を完了したら、手元のローカルリポジトリでコミットし、masterへ統合したあと、中央リポジトリへプッシュする流れとなります。

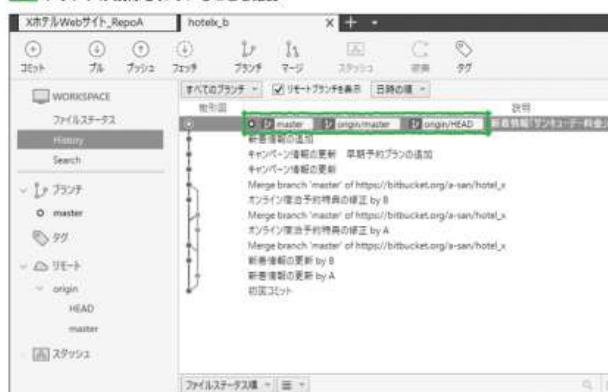
その際、中央リポジトリには Bさんの修正内容が反映されており、Fast Forward にはならないため、そのままではプッシュできません。まずはブルして、Bさんの作業内容を取り込み、その際コンフリクトが発生したら解決をしてから、コミット→masterへ統合→プッシュという流れで作業を終了します。

図17 ブランチの削除



「ブランチを削除」タブ画面で「taskB」と同時に中央リポジトリにひも付けられた「origin/taskB」の両方をチェックして「ブランチを削除」ボタンをクリックします。

図18 ブランチが削除されていることを確認



削除作業後、Sourcetree のブランチツリーを見ると、「master」と「origin/master」のみが残っており、作業用ブランチが削除されていることが確認できます。

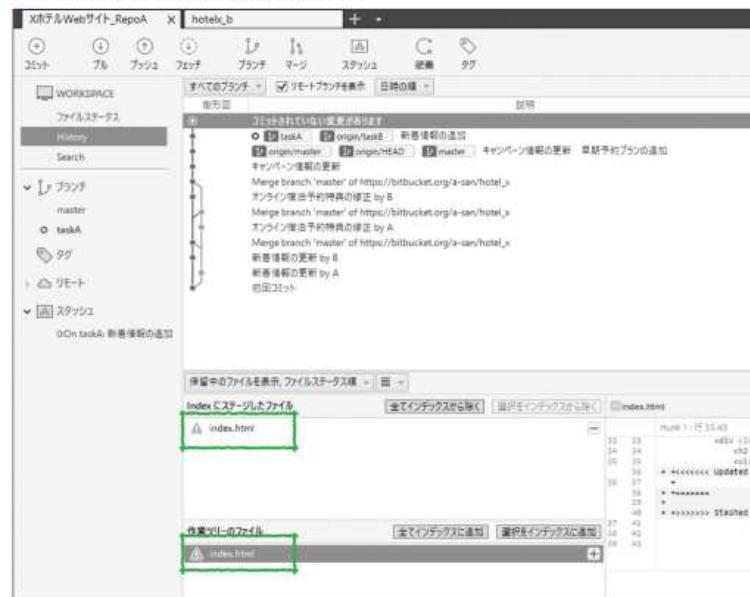
Column

● スタッシュを利用した場合のコンフリクト

本節で紹介したように、コンフリクトはマージ作業だけでなく、スタッッシュの適用を行った際にも発生する可能性があります。ただし、コンフリクト解決のルールはマージ時に発生した場合とまったく同じです。コンフリクトが発生しているファイルの内容を確認して、適切な状態に修

正するしかありません（図1）。複数人での共同作業においては、コンフリクトが多く発生することもあるので、可能な限りコンフリクトが発生しないような進め方を事前に協議しておくことができるよりよいでしょう。

図1 スタッシュ適用でコンフリクトが発生したファイル



スタッッシュ適用でコンフリクトが発生した場合、ファイルリスト上で該当ファイルに△マークがつきます。
これはマージ時にコンフリクトが発生した場合と同じです。

Chapter 4

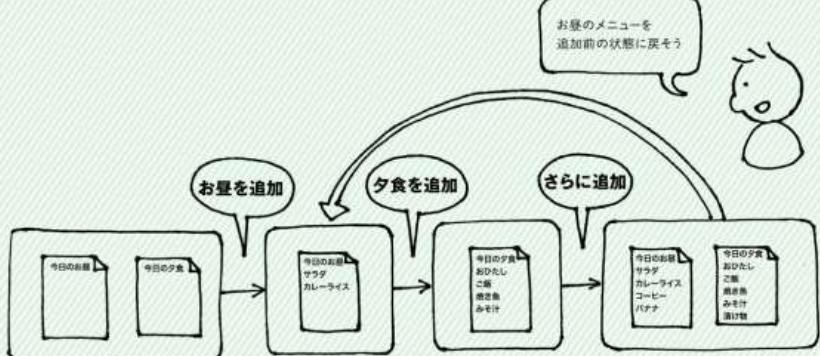
Gitを使った実践開発

「ファイルを過去の状態に戻す」など、実際にWebサイトなどを制作・開発する過程でよく起こる事例をもとに、実践的な使用方法を解説します。
ここでは、やや高度な使い方にもチャレンジしてみましょう。

作業内容を過去の状態に戻す

作業データをGitで管理する大きなメリットのひとつが、過去の状態に戻ることができるというもの。

ロールバックデータ（過去の状態に巻き戻したデータ）を得るために、
単に過去の状態に戻ったり、過去の状態から作業をやり直したりすることができます。



- コミット履歴をさかのぼって、過去のファイルの状態を確認する
- 過去のある時点のファイルの状態に戻って作業を再開する
- チェックアウト、リセット、打ち消し

使用するコマンド

```
$ git checkout -b [作成するブランチ] [過去のコミットID]
$ git reset [過去のコミットID]
$ git reset --hard [過去のコミットID]
```

使用するSourcetreeの機能



チェックアウト ブランチ

▶ Gitを使えば過去の状態に戻ることができる

Gitを利用する大きなメリットのひとつが、過去の状態に簡単に戻ることができるという点です。Gitでは、コミットという一定の指標にしたがって、作業内容を段階的に保管しています。また、ブランチ（branch）という派生したバージョンを管理できる機能があることで、並行した複数の作業も保管できます。

こういったさまざまな状態を適正に管理しているため、Gitを利用すれば過去のさまざまな状態に簡単に戻ること

ができるというメリットを享受できるのです。

たとえば、1週間前に行ったWebサイトの更新作業のうち、トップページの修正のみ再検討課題となり、「とりあえず元の状態に戻したい」といったオーダーにも応えることが容易になります。

Gitで過去の状態に戻るには、いくつかの方法があります。ここでは、チェックアウト（checkout）、打ち消し（revert）、リセット（reset）について紹介しましょう。

▶ チェックアウトで過去の状態に戻す

もともと基本かつ簡単に過去に戻る方法がチェックアウトです。チェックアウトは、現時点での作業対象となるコミットにHEADを移動するというもので、HEADは、音楽プレイヤーの再生ヘッドをイメージしてください。

チェックアウトすることで、現在作業しているファイルの状態をチェックアウトしたコミットの時点まで戻してくれます。もちろん、戻った時点より新しいコミットが削除されるわけではありません。単に再生箇所を過去に移動しているだけです。

過去のコミットをチェックアウトすることで、実際の作業ファイルもその時点のデータに戻るので、そのデータを使ってロールバックデータを揃えるといったこともできます。

たとえば、リニューアル公開したが、全体的に問題があり、いったんリニューアル前の状態に戻したいといったニーズに対応するには、Git管理していない場合はリニューアル前のWebサイトデータ全体を「2015年3月最終版」などのフォルダ名で取っておく必要があります。毎月、毎週のバックアップと考えれば、それこそ無数のフォルダを保管し続けなければなりません。これに対して、Gitで管理していればフォルダはたったひとつで済みます。Gitを使って過去のコミットへ移動するだけで、作業対象のデータがすべて過去の状態に戻ります（図1）。

コミットツリーでチェックアウト

Sourcetreeでは、コミットツリー上の任意のコミットを右クリックして表示されるコンテキストメニューで「チェックアウト」を選びます（図2）。



Gitを利用すればひとつのフォルダで済みます。過去に戻りたいときは、過去のコミットをチェックアウトしてHEADを移動するだけ。

コミットツリー上でチェックアウト



ブランチが参照していないコミットを指定してチェックアウトしようとした場合、図3のような警告メッセージが表示されます。これは、指定したコミットを指す無名のブランチが存在することにして、作業ファイルの状態だけを変更しているので、どのブランチも指していないため、そのままの状態でファイルを編集するなどの作業をした場合、コミットツリーに属しないコミットが発生してしまうという警告です。

ここで「OK」ボタンをクリックすれば、選んだコミット

時点に戻すことができ、選択したコミットには「HEAD」というタグが表示されます。

ただし、現在の作業コピーに編集中のファイルが存在している場合は、過去のコミットをチェックアウトすることはできずエラーとなり、図4のようなエラーメッセージが出力されます。この場合は、エラーメッセージの後半にあるように、作業中のファイルをコミットするか、またはスタッッシュを行って、作業中ファイルを一時退避すればチェックアウトできるようになります。

図3 警告メッセージ

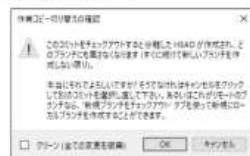


図4 エラーメッセージ

```
error: Your local changes to the following files
would be overwritten by checkout:
  test.txt

Please, commit your changes or stash them before
you can switch branches.

Aborting
```

▶ 過去に戻って作業をやり直す

あ 改めて作業し直したい場合があります。たとえば、何か新しい機能を追加していくが、まったく違う方法のほうが効率的なため、すべてやり直したいといった場合です。このようなケースでは、その作業を開始する直前のコミット状態まで戻り、改めて別の新規ブランチを作成して作業し直したほうがよいでしょう。

過去の状態を元に作業をはじめたい場合は、過去のコミットを指定してブランチを作成しましょう。SourceTreeでは、画面上部の「ブランチ」ボタンをクリックして表示される「新規ブランチ」タブ画面で「指定のコミット」右横

の「...」ボタン（Macは「指定したコミット」の「選択」ボタン）をクリックして、戻りたいコミットを選択します図5～図7。「OK」ボタンをクリックすれば、過去の任意のコミット時点を基点とした新しいブランチが作成されると同時にチェックアウトされます図8。

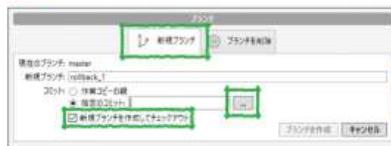
これで、やり直し作業を開始できるようになります。放棄したブランチは削除してもよいですが、そのブランチで作業した内容で再利用できるようなデータが残っているのであれば、しばらくは残しておいてもよいでしょう。最終的にやりたかった作業が完了したあと、完全に不要であることを確認してからブランチを削除しましょう。

図5 ブランチを作成してチェックアウト



SourceTreeでは、画面上部の「ブランチ」ボタンをクリックします。

図6 新規ブランチを作成



「指定のコミット」右横の「...」ボタンをクリックして図6、ブランチを生成させたい過去のコミットを選択して、「OK」ボタンをクリックします図7。コミット指定後、図6のダイアログに戻るのです。

図7 過去のコミットを選択



「新規ブランチを作成してチェックアウト」をチェックして「ブランチを作成」ボタンをクリックします。

「新規ブランチを作成したときのコミットツリー



過去のコミットから派生した新しいブランチが作成され、チェックアウトされた状態となります。

▶ リセットでブランチの状態を変更する

単 純に作業ファイルの状態を過去の状態にしたいだけであれば、過去のコミットをチェックアウトすればよいのですが、共同作業の都合などで、作業中のブランチを変えずに、ブランチ自体を過去の状態に戻したくなった場合にはリセット機能を利用します。

たとえば、masterブランチに対してコミットを行ってしまったものの、本来ほかのブランチに行なうべきだったので、masterブランチの状態は元に戻したいといった場合は、リセット（reset）を使います。まずは、その手順を見てみましょう。

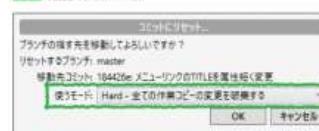
SourceTreeのコミットツリーで、戻したい時点の過去のコミットを右クリックして表示されるコンテキストメニューで「現在のブランチをこのコミットまでリセット」（Macは「○○○（ブランチ名）をこのコミットまで戻す」）を選びます図9。表示される「コミットにリセット」ダイアログで、リセットのモードをプルダウンメニューから選びます。ここでは、「Hard - すべての作業コピーの変更を破棄する」を選んでいます図10。このモードで「OK」ボタンをクリックすると、そのブランチの最新状態が選択した時点まで戻ります（次ページ図11）。

図9 ブランチの過去コミット時点でリセット



コンテキストメニューで「現在のブランチをこのコミットまでリセット」（Macでは「このコミットまで～を元に戻す」）を選びます。

図10 確認ダイアログ



リセットのモードを選んで「OK」ボタンをクリックします。

4-02

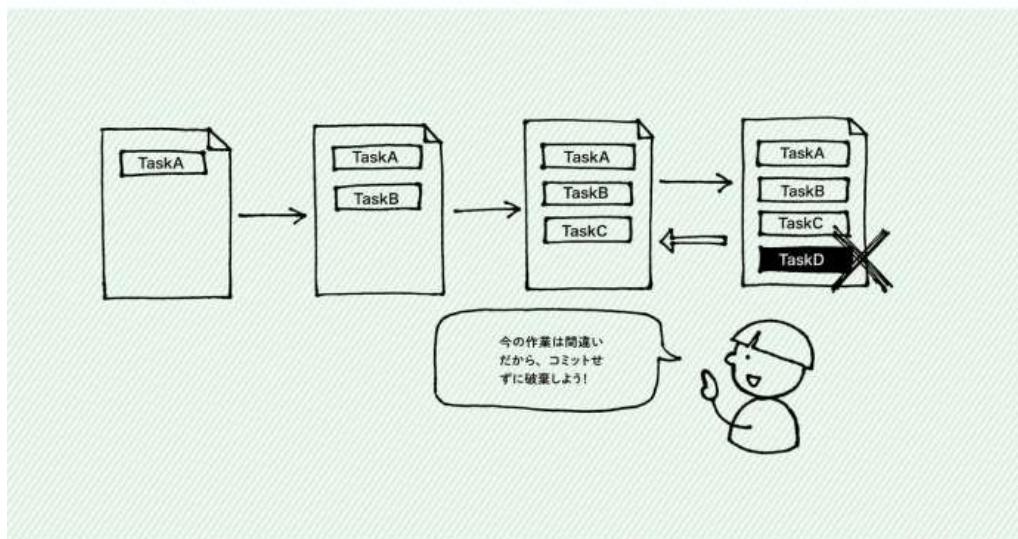
Chapter 4
Gitを使った実践開発

未コミットの作業を取り消す(破棄)

Gitで管理している場合、作業状態をコミットすることで

作業している内容を保管するだけでなく、

現在作業中でまだコミットしていないデータの編集内容をやり直すといったこともできます。



- ✓ コミットしていない作業は簡単に削除することができます
- ✓ 作業の一部分を選択して元に戻すこともできます

使用するコマンド

```
$ git reset --hard HEAD  
$ git checkout HEAD -- [元に戻すファイル名]
```

使用するSourcetreeの機能

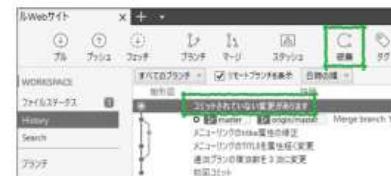


▶ 作業内容を破棄する

作業しているブランチでコミットする前であれば、その時点で作業している内容をコミットせずに破棄することもGitの機能で簡単にできます。単一のファイルを編集しているのであれば、エディタでファイルを修正すればよいことですが、複数のファイルを編集して、ある程度作業が進行した時点で作業内容全体を破棄するならば、漏れなく破棄したいところです。そこでGitを利用します。

作業コピーの内容を破棄するには、Sourcetreeの画面上部にある「破棄」(reset)ボタン(Macは「操作」メニューの「リセット」)をクリックします図1。すると「変更を破棄」ダイアログが表示されるので、「全てリセット」タブ画面に切り替えます図2。ここで「全てリセット」ボタンをクリックすると確認ダイアログが表示され、「OK」ボタン(Macは「リセット」ボタン)をクリックすると、作業ツリーに存在していた未コミットの作業内容がすべて破棄されます図3図4。

図1 「破棄」ボタンをクリック



コミットしていない作業内容がある場合、コミットツリーに「コミットされていない変更があります」と表示されています。このとき、画面上部の「破棄」ボタンをクリックすることができます。

図2 未コミットの作業を取り消す(破棄)

図2 「全てリセット」タブ画面

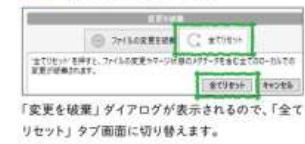


図3 確認ダイアログ



作業内容を破棄して元の状態に戻してよいか確認されます。

図4 作業内容が破棄される



作業内容が破棄され、コミットツリーから「コミットされていない変更があります」という表示が消えます。

▶ 特定のファイルに行われた変更だけを破棄する

特定のファイルに対して行われた変更内容をすべて破棄する場合は、Sourcetreeでファイルステータス画面に切り替え、該当ファイルを右クリックして表示されるコンテキストメニューで「破棄」(Macでは「リセット」)を選びます図5。確認ダイアログが表示されるので、「OK」ボタンをクリックすれば、そのファイルに加えた変更内容はすべて破棄されます図6(ファイルが削除されるわけではありません)。

図5 ファイルの変更箇所すべてを破棄



コンテキストメニューで「破棄」を選びます。

図6 確認ダイアログ



ファイルに加えた変更内容はすべて破棄されます。ファイル自身が削除されるわけではありません。

4-03

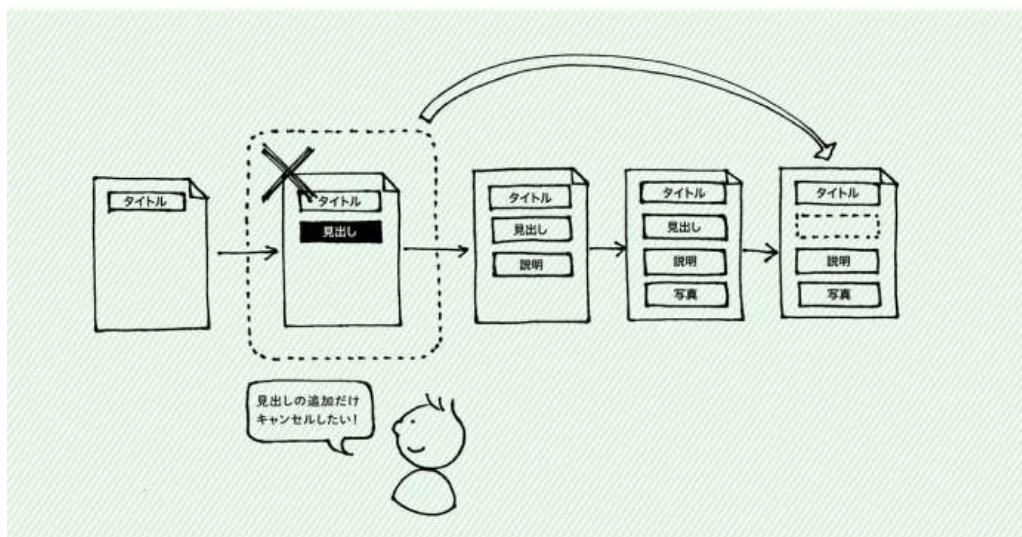
Chapter 4
Gitを使った実践開発

過去の修正を元に戻す(打ち消し)

コミットにより保管されている過去の作業内容を取り消したいといったニーズにも応えてくれるのがGitです。

過去に行なった変更を残しつつ、その変更を打ち消すことができます。

その方法や打ち消しの履歴を残す意味を解説します。



- 過去に行なった作業を打ち消す手順を確認する
- 作業をキャンセルした履歴をコミットとして残すことの意味を知る
- リセットやチェックアウトとの違いを理解する

使用するコマンド

```
$ git revert [打ち消すコミット]
```

03 過去の修正を元に戻す(打ち消し)

▶ 過去に行なった作業を打ち消したい

Gitでの作業は、主要な修正が完了した時点でコミットを重ねていきます。作業した内容はすべて記録されているため、任意の過去の段階に戻ることが可能ですが。ただし、ある一部の修正だけを元に戻したい、より厳密にいえば「なかったことにしたい」という場合、単にコミットを過去にさかのぼってしまうと、その過程で行ったコミットもなかったことになってしまいます。「ある特定のコミットだけをなかったことにしたい」、そんなニーズにも応えられるのがGitです。

たとえば、「1週間に実施したWebサイトの更新作業のうち、トップページの修正だけ元に戻したい」といったオーダーがあった場合、トップページの修正のみを記録したコミットがあれば、それを「打ち消す」(Revert)ことでオーダーに応えることが可能になります。

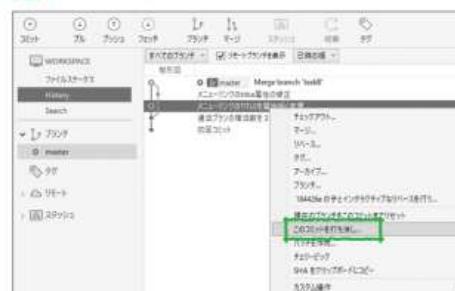
Sourcetreeでは、コミットツリー上で打ち消したいコミ

ットを右クリックして表示されるコンテキストメニューで「このコミットを打ち消し...」(Macは「コミット適用前に戻す」)を選びます^{図1}。確認ダイアログが表示されるので、「はい」ボタンをクリックします^{図2}。すると、そのブランチ上で新たな最新コミットが作成されます^{図3}。この最新コミットの内容は、打ち消しで選んでいた過去コミットの作業内容を元に戻す作業を行ったコミットです。

要するに、「打ち消し(Revert)」という機能は、過去の時点で行なった作業の「逆」を行う作業を新たに実行することです。

したがって、打ち消しを行った対象のコミットはそのままに、打ち消し対象のコミットで行なっていた編集作業の「逆」の作業を行なうことにして、その作業を新たにコミットしたことになります。^{図4}。

図1 コミットを打ち消す

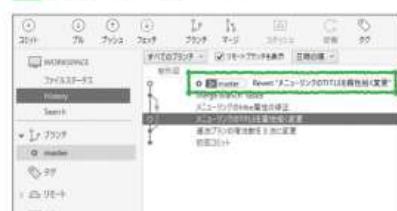


コンテキストメニューで「このコミットを打ち消し...」を選びます

図2 確認ダイアログ

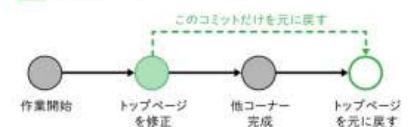


図3 打ち消しのための新しいコミット



打ち消しを行うと、過去のコミットでの作業を元に戻す作業を行った新しいコミットが作成されます。

図4 打ち消しによるブランチの状態



4-04

▶ マージコミットを打ち消す場合

ブランチを統合（マージ）してきたコミットに対して「このコミットを打ち消し...」を行うと、エラーとなります。これは、Sourcetreeの打ち消しでは、「逆」の操作を決める際に、2つの元コミットがあるマージコミットにおいて、どちらの元コミットを基準にするか指定するオプションを利用できないためです。

マージコミットを打ち消す場合は、Sourcetreeの画面上部にある「ターミナル」をクリックしてターミナル画面を起動し、

```
git revert -m 1 コミットID
```

というコマンドを実行することで可能となります。-mが元のコミットを指定するオプションで、「1」「2」のどちらかを指定することでマージ元、マージ先どちらのブランチのコミットを元に「逆」の作業を組み立てるか決定されます。

図5 Sourcetreeでマージコミットを打ち消した場合



「このコミットを打ち消し...」でマージコミットを打ち消そうとするエラーが発生します。

Column

○ リセット、打ち消し、リベースについて

リセットと打ち消し

リセットはブランチの状態を過去のある時点に巻き戻すという操作です。巻き戻したつもりでいても、中央リポジトリやほかの作業スタッフのリポジトリにコミットが残っていることがあります。その場合、中央リポジトリのほうがリセット前に行っていたコミット分先行している状態となるので、ブルすればリセットで巻き戻したコミットが復活してしまうことになります。

これに対して、「打ち消し（Revert）」操作では、「間違ったコミットがあって、それを打ち消した」という経緯が新たにコミットとして記録されるというものです。ブランチには過去のコミットもすべて残った状態となり、他ブランチとも正しくマージでき、中央リポジトリへプッシュする際もFast Forwardで正しく行えます。

したがって、過去の作業内容に間違いがあり、訂正したい場合は「打ち消し」を利用するが適切です。リセットは、masterブランチや中央リポジトリに影響を与えないようなローカルリポジトリ専用の作業用ブランチなどで、ブランチの状態をどうしても書き換える必要がある場合に使用するようにしましょう。

リベース

リセットがブランチの状態としてコミットの位置を変えるのに対して、過去のコミット履歴を書き換える操作として「リベース」があります（本書では詳細な解説はいたしません）。

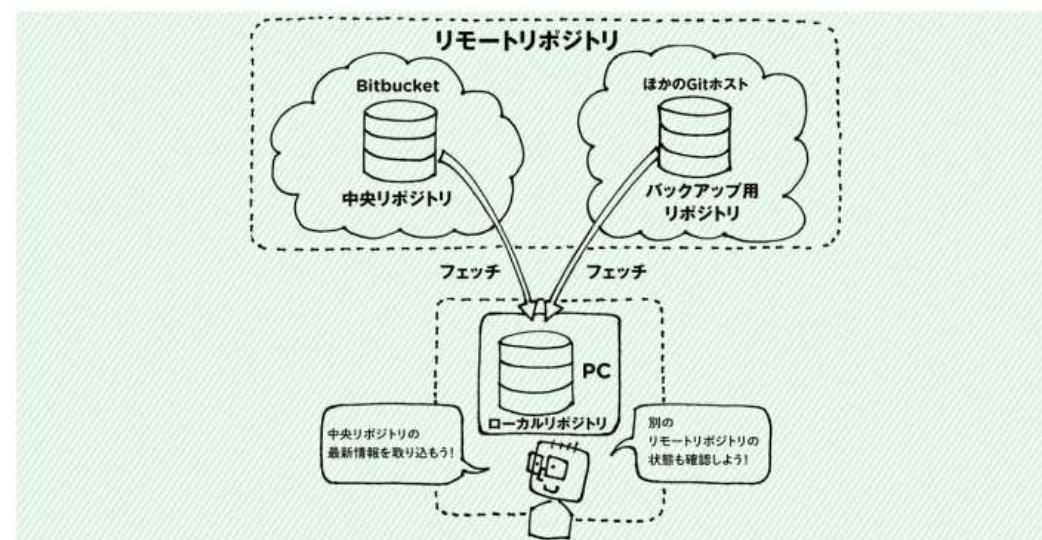
リベースは、リポジトリ全体の状態を変えてしまう可能性があり、中央リポジトリを介した他スタッフとの連携作業においては、十分に理解した上で行わないトラブルが生じる危険を伴う操作となります。中央リポジトリにPush済みのコミットに対してこの操作を行った場合、あるはずのコミットがなくなったり、あるべきではないコミットが現れるなど、混乱が生じます。Git操作やリポジトリに関して高度な知識を有していない場合は、問題が発生しても解決できず、リポジトリ 자체を放棄せざるを得ない状況に陥ることも考えられます。

したがって、原則としてはプッシュしていないコミットについてのみ、リベースによる書き換えが許されると考えるべきでしょう。また、どうしてもリベースしなければならない場合、それがローカルリポジトリのみで行うとしても、事前にリポジトリを含むフォルダをバックアップ（コピー）してから操作することを強く推奨します。

リモートリポジトリの最新状態を取得

Gitの初歩ともいえる「フェッチ」機能は、リモートリポジトリの最新状態を取得する機能です。

マージは行わずに、とりあえずリモートリポジトリの状態を把握する作業は意外と重要なものです。ブルとの違いを理解するようしましょう。



- ✓ リモートリポジトリ、リモートブランチの操作の仕方に慣れる
- ✓ アレせずにリモートブランチをフェッチ（同期）して確認する
- ✓ アルとフェッチの関係を理解する

使用するコマンド

```
$ git fetch [リモートリポジトリ]
```

使用するSourcetreeの機能



▶ リモートリポジトリとは

リモートリポジトリとは、手元にあるローカルリポジトリに対して“クローン元のリポジトリ”もしくは“同じリポジトリからクローンされた兄弟リポジトリ”的こと（→P.015）。もっともわかりやすいリモートリポジトリが Bitbucket や GitHub などでクローン元としている「中央リポジトリ」でしょう。また、中央リポジトリをハブとして共同作業している他のスタッフのローカルリポジトリもリモートリポジトリとなります。

リモートリポジトリは、Sourcetree のタブ画面の左側（Mac では左サイドバー）にある「リモート」にリストアップされます（図1）。

複数人で共同作業している場合、リモートリポジトリには、自分がブッシュしたコミット以外にも、ほかの作業スタッフがブッシュしたコミットやほかの作業者が作成したブランチが存在している可能性があります。

▶ リモートブランチの同期（フェッチ）

Sourcetree に登録したリモートリポジトリは、そこに記録されているブランチ情報も Sourcetree 上に表示されるようになります（図2）。

ここに表示される情報は、リモートリポジトリに最後に接続したときの状態です。リモートリポジトリ上で状態が変わってもリアルタイムには反映されないため、最新の情報を手元に反映する必要があります。そのための操作が「フェッチ」（fetch）です。

フェッチという操作は、リモートリポジトリの状態を手元のリポジトリに反映してくれるものです。ただし、マージは行いません。一方、ブルという操作は、リモートリポジトリからフェッチすると同時にマージまで処理しようとします。

図2 リモートブランチの表示



コミットツリーに「リポジトリ名/ブランチ名」という表記でリモートブランチが表示されます。この図では「origin/master」というのがリモートブランチです。

図1 Sourcetree 上のリモートリポジトリ



図4 フェッチ後のコミットツリー



コミットツリーに最新のリモートブランチ情報が反映されます。

図6 master に origin/master をマージ



master ブランチをチェックアウトした状態で origin/master ブランチをマージします。

図5 フェッチ後の Sourcetree 画面



フェッチにより origin/master の最新ブランチが取り込まれ先行しています。また、ローカルブランチである master には「2」というメッセージが表示されると共に、画面上部の「ブル」ボタンに「2」という数字が表示されています。

図7 マージ後の Sourcetree 画面



master ブランチをチェックアウトした状態で origin/master ブランチをマージします。

▶ リモートリポジトリの追加と削除

一般的に最初に作成してクローン元となるリモートリポジトリには origin というリポジトリ名を付けます。多くの場合、これが中央リポジトリとなります。 origin 以外の名前についてもできますが、慣例に習って origin としておくことをおすすめします。

複数人作業では、基本的に中央リポジトリさえあればハブとして機能しますが、それ以外のリポジトリを作成してバ

ックアップ用として利用することもあります。ほかのリポジトリをリモートリポジトリとして追加する場合は、Sourcetree の「リモート」の附近で右クリックして表示されるコンテキストメニューで「新規リモート」（Mac は「リモートを追加」）を選びます（図8）。

「リポジトリ設定」ダイアログの「リモート」画面が表示されるので、「追加」ボタンをクリックします（図9）。

図8 リモートリポジトリを追加



「リモート」の付近で右クリックして表示されるコンテキストメニューで「新規リモート」を選びます。

図9 リポジトリ設定



リポジトリ設定画面には、クローン元の中央リポジトリ origin がリストアップされています。「追加」ボタンをクリックして、そのほかのリポジトリを追加します。

「リモートの詳細設定」画面に切り替わるので、リモート名とリポジトリのURLを指定します^{図10}（Macでは自動的にこの画面が表示されます）。リモート名は任意の名称でかまいません。たとえばバックアップ用なら「backup」でもよいでしょう。また、BitbucketやGitHubに登録したアカウントで作成したリポジトリの場合、そのアカウント情報をSourcetreeに登録していれば、リポジトリのURLから自動的に判断して「外部拡張サービスとの拡張統合オプション」がセットされます。あとは「OK」ボタンをクリックすれば、リモートリポジトリに追加されます^{図11}。

逆に、不要になったりモートリポジトリは削除することもできます。削除したいリモートリポジトリ名のコンテキストメニューで「～（リモートリポジトリ名）を削除」を選びます^{図12}。確認ダイアログで「OK」ボタンをクリックすればリモートリポジトリが削除されます^{図13}^{図14}。

図10 リモートの詳細設定



リモート名とリポジトリのURLを指定します。

図11 新しいリポジトリが追加される



新しいリポジトリがリモートリポジトリに追加されます。

図12 リモートリポジトリを削除



コンテキストメニューで「(リモートリポジトリ名)を削除」を選びます。

図13 確認ダイアログ



図14 リモートリポジトリを削除



リモートリポジトリが削除されます。

リモートブランチのリセット

前節で解説したリセットですが、中央リポジトリのブランチの状態をリセットすることも可能です。ただし、リモートのブランチを対象とする場合は単純にリセット操作ができるわけではなく、一度ブランチを削除して、同じ名前で作り直すことになります。

対象とするリモートリポジトリによっては、ブランチの削除に制約がある場合があります。ここではBitbucket上のブランチを削除する手順を紹介します。

この作業は、中央リポジトリのブランチを削除して、履歴を書き換えた状態で、同名で作り直す操作を行うことになります。

中央リポジトリのメインブランチの削除

まずは、Sourcetreeのリモートに表示されている「origin/master」をコンテキストメニューから削除します^{図15}。確認ダイアログで「OK」ボタンをクリックします^{図16}。確認ダイアログには「リモートのブランチも削除されます（削除がブッシュされます）」と表示されますが、「OK」ボタンをクリックするとエラーとなります^{図17}。これはBitbucketではメインブランチに設定しているブランチを削除しようとするとエラーが起こるようになっているからです。結果的に、Sourcetree上のリモートブランチからは削除されますが、中央リポジトリには依然としてmaster

ブランチは存在することになります。この状態で再度フェッチすると、Sourcetree上でもorigin/masterが復活するので、実際にはリモートのブランチが消えていたことがわかります。

いずれにしろ、Bitbucketでは初期状態でmasterがメインブランチに設定されているため削除することができず、リセットさせようとしてもうまくいきません。そこで、Bitbucket上でメインブランチを変更するようにします。

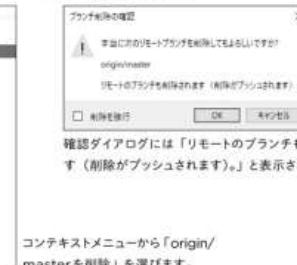
中央リポジトリのmasterブランチの削除

Bitbucketでの操作に入る前に、Sourcetreeで手元のリモートリポジトリ master のコピーを使って、中央リポジトリに master のコピーを作成しておきます。方法としては、ローカルブランチである master を「master_backup」として中央リポジトリにプッシュします^{図18}。これにより手元のリモートブランチにも master_backup が追加されます^{図19}。

図15 リモートブランチを削除



図16 確認ダイアログ



確認ダイアログには「リモートのブランチも削除されます（削除がブッシュされます）」と表示されています。

コンテキストメニューから「origin/masterを削除」を選択します。

図17 削除したときのエラーメッセージ



Bitbucketではメインブランチに設定しているブランチを削除しようとするとエラーになります。

図18 masterブランチのコピーを中央リポジトリにプッシュ



ローカルブランチである master を「master_backup」として中央リポジトリにプッシュします。

図19 リモートブランチが追加される



手元のリモートブランチにも master_backup が追加されました。

ブラウザでBitbucketの管理画面にログインします。メインブランチであるmasterを逃避するために「master_backup」などのブランチを新規作成します。その後、「Setting」画面を表示して、「メインブランチ」のプルダウンリストで「master」から「master_backup」へ切り替えます図20。これでBitbucket上でメインブランチを変更できました。

この状態でSourcetreeに戻り、改めて「origin/master」の削除を行うと、中央リポジトリ上のmasterブランチも削除できます。同時にフェッチを行い、origin/master_backupブランチをリモートブランチとして取り込んでおきましょう。

masterブランチをリセット

次にローカルのmasterブランチをリセットします。リセットの方法は「4-02 未コミットの作業を取り消す（破棄）」(→P.138)を参照してください。

図20 Bitbucketでメインブランチを切り替える



図21 リセットしたmasterブランチをpush



リセットしたローカルのmasterブランチを「origin/master」として改めて中央リポジトリにプッシュします図21。これで中央リポジトリ上のmasterブランチをリセットした状態となります図22。

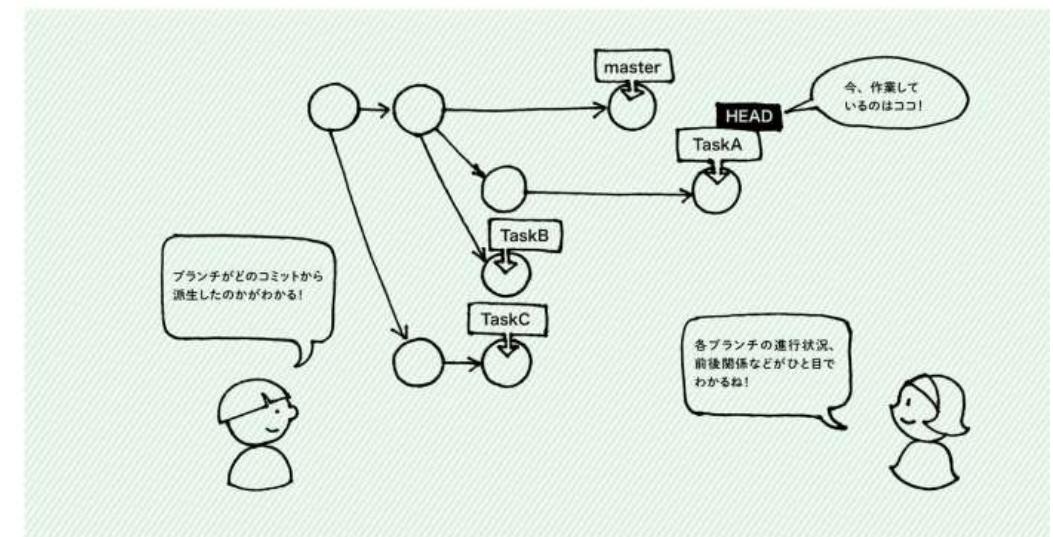
masterブランチリセットの注意点

複数人で共同作業している場合、中央ブランチをリセットすることでほかのスタッフにも影響が出てしまいます。ただし、リセットは基本的に過去に行われているコミットに戻るということであるため、中央リポジトリのリセットは、ほかのスタッフからすると各自のローカルリポジトリが単に先行した状態となり、そのまま誰かがプッシュすると、リセットを行う前の最新の状態に戻ってしまうことになります。事前に中央リポジトリのブランチをリセットすることを共同作業者に通知しておくことが重要となります。また、リセット後には、各自手元のリポジトリのブランチの状態も変更するようにしましょう。

4-05

コミットツリーから何がわかるか

ブランチを追加するにつれ、複雑になりがちなコミットツリー。しかし、複雑に見えるコミットツリーも、実は分岐と統合の組み合わせにすぎません。コミットツリーの見方を知ることで、Gitへの理解を深めましょう。



- コミットツリーの見方、操作の仕方を覚える
- コミットの前後関係をたどってみる
- 複雑なコミットツリーも分岐と統合のパターンの組み合わせ

使用するコマンド

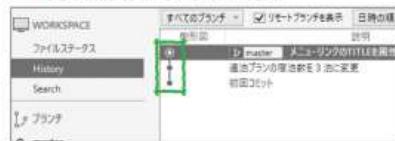
```
$ git log --graph --oneline --decorate=full
```

▶コミットツリーの枝分かれを理解しよう

次では、リポジトリを作成した初期段階では一般的に「master」というひとつのブランチだけが存在しています。ブランチがひとつしかない状態では、コミットツリーも当然枝分かれすることなく、1本の幹のように一直線のコミットツリーとなります（図1）。

では、どのようなときに枝分かれするのでしょうか。基本的に、ブランチが複数存在するときに、コミットツリーも「枝分かれする可能性がある」といえます。ただし、ブランチの数と枝の数はイコールではありません。ブランチを削除してブランチ数が減ったとしてもコミットツリーの枝分かれは残ります。

図1 ブランチがひとつのときのコミットツリー



事例で見てみましょう。masterの任意のコミット時点でtaskAというブランチを作成します。すると、コミットツリーではmasterからtaskAブランチが枝分かれし、taskAで作業を進めてコミットすると、taskAの最新コミットが先行した状態として図2のように表示されます。

taskAでの作業が完了してmasterに統合（マージ）して新たなコミット（マージコミット）を作成した場合は、図3のように枝分かれがつながり、コミットツリーは1本に戻ります。このあと、taskAブランチを削除したとしても、コミットツリー上のコミット履歴は残っているため図4のようにになります。

図2 ブランチが並行してふたつあるときのコミットツリー



masterとtaskAというふたつのブランチがあり、それぞれのブランチでコミットしたときに枝分かれします。

図3 ふたつのブランチをマージしたときのコミットツリー



masterにtaskAをマージすると枝がつながります。



taskAブランチを削除したとしても、枝分かれの履歴はコミットツリーに残ります。

▶コミットツリーにはどんな情報が含まれているのか

コミットツリーは「コミットグラフ」などとも呼ばれ、SourceTreeではログ画面の「樹形図」という部分に表示されます。この樹形図においては、1行が1コミットを表しています。また、樹形図の右横には「説明」があり、コミットメッセージが表示されるほか、ブランチ名（masterなど）やタグが表示されます。そのほか各コミットの作成日時、

作者（コミットしたユーザー名）、コミットIDも表示されます。コミットログ画面上部にはいくつかの抽出メニューがあり、「すべてのブランチ」または「現在のブランチ」を切り替えたり、リモートブランチを同時に表示するかどうかを切り替えたりもできます（図5）。また、「日時の順」「親子関係の順」で表示状態を切り替えることもできます（図6）。

④コミットツリーからわかる情報

(1) 各ブランチの最新コミット

複数のブランチがあるとき、各ブランチがどのコミットを指示しているかがひと目でわかるだけでなく、ほかのブランチと比べて進んでいるのか、遅れているのかもわかります。

(2) コミット同士の前後関係（親子関係）

各コミットごとにどちらが先か、後かという前後関係（枝分かれしていれば、各枝ごとに）やマージされた履歴をたどることができます。

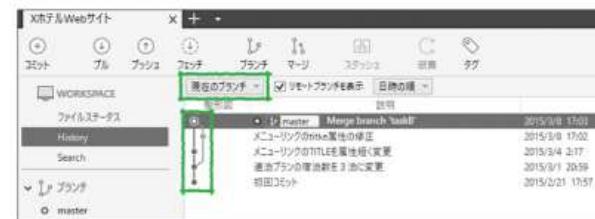
(3) リモートリポジトリの状態

追跡対象になっているリモートブランチもすべて表示することができます。たとえば、「origin/master（中央リポジトリのmasterブランチ）」がどのような状態であるか（相関するローカルリポジトリのoriginと同じなのか、進んでいたり遡っていたりするのか）がわかります。

(4) タグ情報

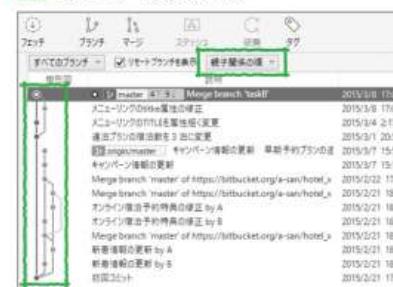
タグが過去のどのコミットを指しているのかも、コミットツリーを見るとわかります。

図5 現在のブランチのみを表示

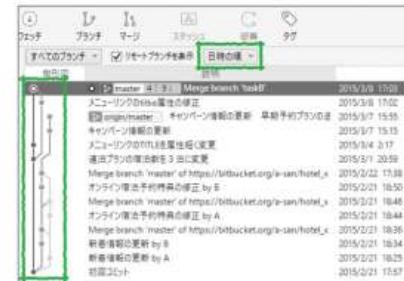


コミットツリーに現在のブランチだけを表示しています。

図6 コミットツリーの表示切り替え

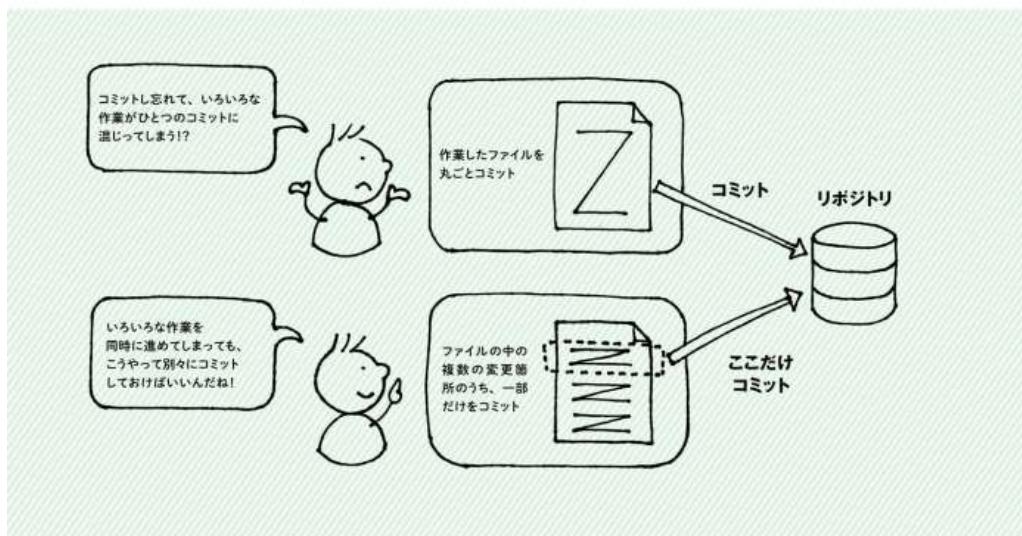


コミットツリーは「日時の順」と「親子関係の順」で表示状態を切り替えることができます。



コミットログを整理しよう

コミットにより作業経過を段階的に保管できるGitですが、作業を進めるにつれてコミットツリーは複雑になってしまう可能性があります。そこで、コミットログを整理しておくことがとても重要になります。



- コミットツリーを整えることの意味とメリットを理解する
- 作業履歴からコミットを作成するために複数の機能があることを理解する
- コミットツリーをメンテナンスする方法を知る

使用するコマンド
\$ git add -p [対象のファイル]

使用するSourcetreeの機能
追加

▶ コミットツリーを整える意味

作業が進むに連れて、コミットツリーは複雑になっていく可能性があります。また、何か問題が発生した際、コミットツリーで過去のコミットから原因を探らなければならないことがあります。そんなとき、コミットツリーをわかりやすくしておくことは、問題解決に際してかなり有効です。

過去に行ったどのコミットが原因かを特定するには、そのコミットがどのような意図で作成されたものかを知ることが重要です。たとえば、ひとつのコミットがひとつの機能追加・改修など、適正にひも付いていれば、追跡も容易になります。

ただ、コミットを極力きれいに整理しておくことが理想的ですが、コミットを整えるために作業順序を考えなくてはならないなど負担も増えてしまう可能性もあるので、無理のない範囲で検討しましょう。特に複数人で共同作業する場合は、現場の状況に応じて、各自適宜判断してコミットできるワークフローの柔軟性も重要です。

そこで、プロジェクトを開始する前にスタッフ間で協議したうえで、ある程度の決まりとしてコミット運用のポリシーを決めておくようにしましょう。

ポリシーとしては、厳しいものから緩やかなものまでいろいろ考えられますが、たとえば右のようなルールを基準に考えてみてください。

▶ 一部の変更箇所(Hunk)のみをコミット

コミットを行う際、いくつかの作業が混ざっているため、一部分だけコミットしたいという場合があります。

編集したファイル単位でコミットするかしないかを選択するには、作業ツリーにあるファイルをステージに追加するかしないかで操作できます(図1)。

Gitでは、さらにファイル内の編集箇所を部分的にコミットすることができます。

Gitは、作業によって発生した変更内容をファイル単位ではなく、変更箇所単位で管理しています。この変更箇所の単位は「Hunk」と呼ばれます。コミットのポリシーとしては、「1コミットをひとつの作業に対応づける」というものだったとしても、事情があって同時に作業してしまうこともあります。そのような場合は、ファイル内のある範囲の編集内容、つまり該当するHunkだけをコミットすることができるわけです。

②コミットポリシーの例

【厳密さ 高】

1コミットには、必ずひとつの作業単位だけを含めるようにして、作業単位の名前を必ず書くこと

【厳密さ 中】

1コミットには、そのときいっしょに行なった作業はすべて含めてもよいが、メッセージ欄にはきちんと作業内容を書くこと

【厳密さ 低】

コミットメッセージは書いても書かなくても自由だが、作業がひと段落したら必ずコミットすること

図1 一部のファイルのみステージに追加



編集したファイルのうち一部のファイルだけを追加すれば、ファイル単位での部分的コミットができます。

Sourcetree のファイルステータス画面では、作業ツリーに編集したファイルがリストアップされます。Hunk 単位でコミットしたいファイルを選択すると、画面右側にファイルの変更内容がリストアップされるので、追加したい変更箇所にある「Hunk をステージへ移動」ボタン（Mac は

「Hunk をステージングに移動」ボタン）をクリックします [図 2]。すると、その Hunk のみをコミット対象としてファイルがステージに追加されます [図 3]。この状態でコミットを行えば、Hunk 単位でのコミットが完了します。

図 2 一部の変更箇所 (Hunk) だけを選択



図 3 一部の変更箇所 (Hunk) だけをコミット対象としたファイルをステージに追加

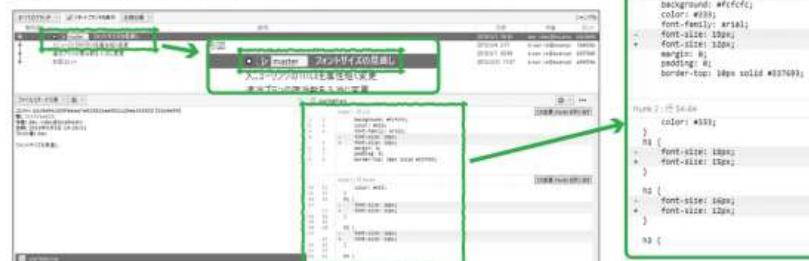


▶ コミットを分割する

直近で行ったコミットであれば一度リセットして、改めて差分を分けてコミットし直すことで、分割することができます。ふたつの作業を並行して進めていて、作業内容が混ざったままコミットしてしまった場合などに活用します。コミットの分割作業は、リセット操作により実現します。

コミットをSoftモードでリセット
図 4 は「フォントサイズの見直し」というコミットを行った直後の状態です。コミット対象となった CSS ファイルの内容を見ると、数カ所 font-size を変更していることがわかります。

図 4 複数作業混在のまま行ってしまったコミット



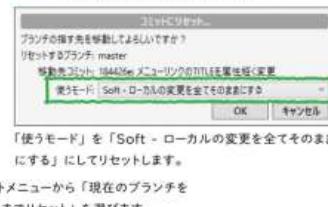
まずはこのコミットを Soft モードでリセットします。コミットツリーで直前のコミットのコンテキストメニューから「現在のブランチをこのコミットまでリセット」（Mac は「○○○○（ブランチ名）をこのコミットまで戻す」）を選びます [図 5]。「コミットにリセット…」ダイアログで「使うモード：Soft - ローカルの変更を全てそのままにする」にチェックします [図 6]。

Soft モードによるリセットが完了し、ステージに先ほどの CSS ファイルが戻っています [図 7]。

図 5 コミットをリセット



図 6 「コミットにリセット」ダイアログ



「使うモード」を「Soft - ローカルの変更を全てそのままにする」にしてリセットします。

コンテキストメニューから「現在のブランチをこのコミットまで戻す」を選択します。

図 7 リセット完了後のステージ



コミットがリセットされ、作業ファイルがステージに追加した状態まで戻ります。

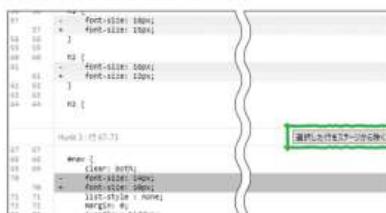
コミット対象の変更箇所を除外する

リセット後の上図 [図 7] の Hunk のリストで「Hunk をステージから除く」ボタンをクリックすれば、その Hunk をコミットから除外することができます。

さらに Git では、Hunk 内の一部をステージから除外す

ることもできます。Hunk の一覧で、除外したい Hunk 内の任意の変更箇所（行）をクリックして選択し、「選択した行をステージから除く」（Mac は「行をステージングから除く」）ボタンをクリックします [図 8]。選択した行が Hunk から取り除かれたファイルがステージに残ります [図 9]。

図 8 Hunk から変更箇所を取り除く



Hunk 内の任意の変更箇所（行）をクリックして選択し、「選択した行をステージから除く」ボタンをクリックします。

図 9 操作結果



選択した行が Hunk から取り除かれたファイルがステージに残ります。

再度コミットする

ステージからHunk自体を除外したり、Hunk内の行を除外したりするなどして、改めてコミット用のファイルをステージに追加した状態で再度コミットを行います図10。

コミットが完了すると、先ほど除外した変更内容を含む

図10 一部の変更箇所を除外してコミット



図11 さらにコミットするファイルが残る

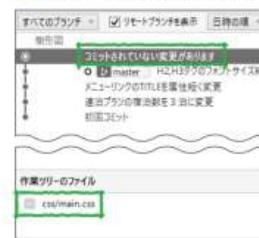
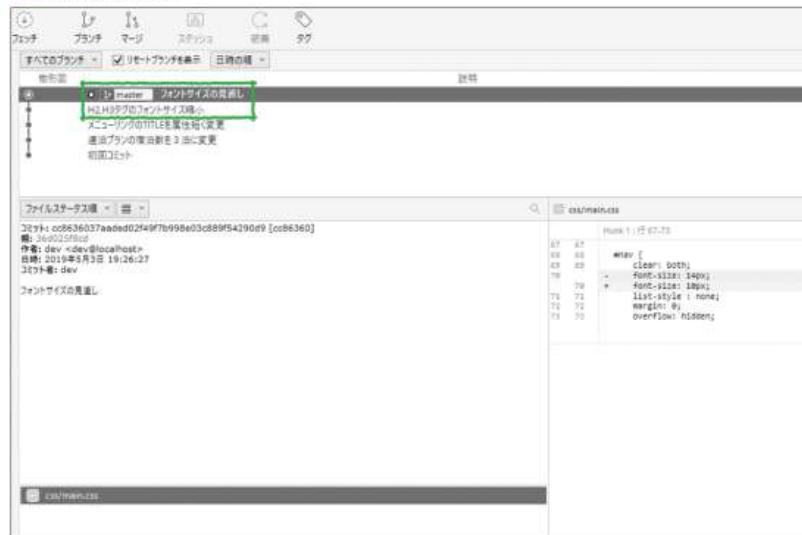


図12 残りの変更内容をコミット



除外した変更内容を含むファイルが作業コピーに残っています。

図13 コミットが分割される



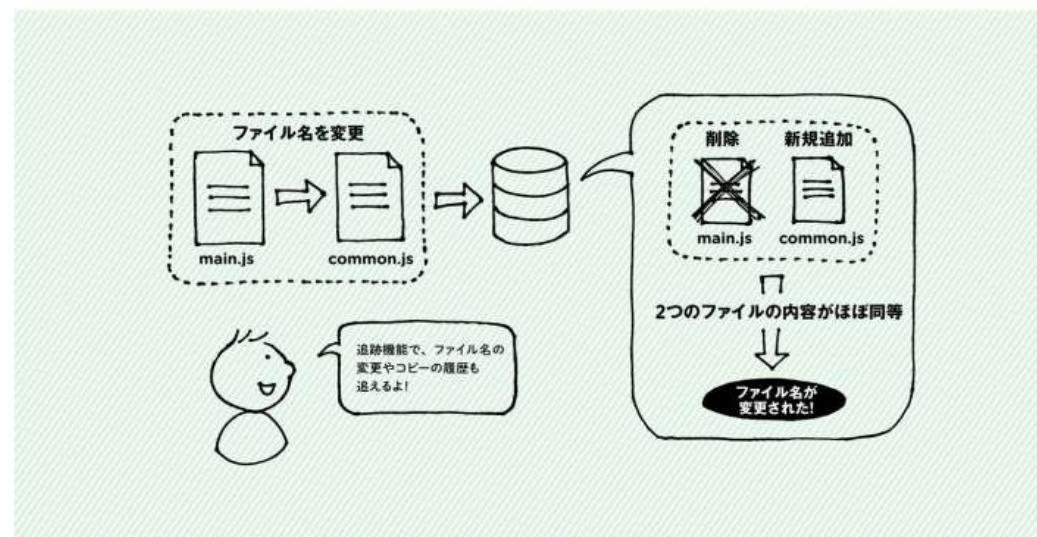
ひとつだったコミットがふたつのコミットに分割されます。

4-07

ファイルの移動、複製の追跡

Gitはさまざまな優れた機能を有しています。

この節では、ファイルの移動やコピーといった作業リソースの状態の変化をGitがどのように判断し、管理しているのかを紹介しましょう。



- ✓ ファイルの移動、複製、削除がGit上でどのように自動的に判断されるのかを知る
- ✓ ファイルの移動や複製が行われた履歴をたどってみる

使用するコマンド

```
$ git diff --find-renames [コミットID]
$ git diff --find-copies [コミットID]
```

▶ 内容が同じファイルを自動追跡

Gitは非常に高度なファイル追跡機能を有しています。ファイルの内容が一定以上同じであれば、ファイルを自動的に追跡してくれます。あるファイルが削除されて、**内容が似たファイルが同時に追加されると、それはファイルが移動したと見なされる**のです。

たとえば、main.jsというJavaScriptファイルのファイル名を、common.jsに変更したとします。このとき、このJavaScriptファイルは内容が空のファイルで、ファイル名を変更しただけです。

Sourcetreeのログ画面を見ると、作業ツリーにふたつのファイルがリストアップされています（図1）。common.jsには■アイコンがついており、リポジトリに未登録なファイルとして認識されています。main.jsには■アイコンがついており、削除されたファイルとして認識されていることがわかります。

図1 作業ツリーにふたつのファイル



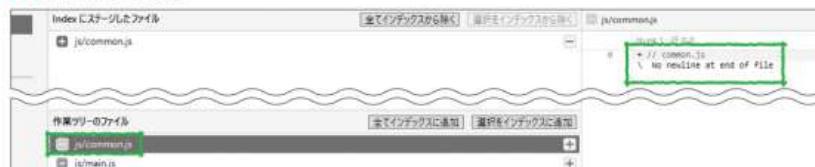
common.jsはリポジトリに未登録なファイル、main.jsは削除されたファイルとして認識されています。

図2 ステージに追加



ステージに追加するとcommon.jsのみがリストアップされ、ファイル名変更として取り扱われています。

図3 ファイルの内容も変更



common.jsの内容を変更しています。

これらをステージに追加します。するとステージにはcommon.jsのみがリストアップされ、この時点でファイル名が変更したとして処理されます（図2）。

同じ作業を、今度はファイル名と同時にファイルの内容を書き換えて行ってみます（図3）。ファイルの内容を書き換えて追加すると、common.jsは追加ファイル、main.jsが削除ファイルとして扱われています（図4）。

つまり、削除された際のmain.jsの内容と、新たに追加されたcommon.jsの内容が異なるため、ファイル名の変更としては扱われなくなり、ファイルの削除と追加という処理に変化していることがわかります。

Gitでは、**同じ内容のファイルが追加されていなければファイルの削除として記録されます。同様に、既存のファイルに一定以上内容が似たファイルが追加されると、ファイルの複製として記録されることになります。**

図4 ステージに2つのファイルが追加される



今度はcommon.jsとmain.jsのふたつがリストアップされ、ファイルの削除と追加という処理になっています。

▶ ファイル名の変更や複製の履歴を追跡

ほど見たように、Gitではファイル名単位で履歴を管理しているので、ファイル名と同時に内容も変更して追加するといった通常作業ではファイルの削除と追加が同時に行われたものと見なされて、**ファイル名が変更された記録が抜けてしまっています**。そのファイルが複数のファイルから参照されている共有ファイルの場合、ファイル名がいつ変更されたかという記録は非常に重要です。したがってファイル名変更の記録を追跡できることも重要なとなります。

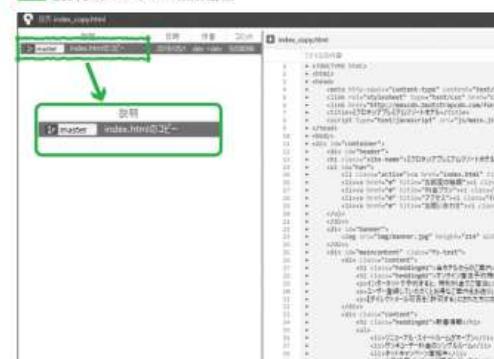
実際のところ、Gitではファイル名の変更があったことの履歴は残されていて、追跡することも可能です。

たとえば、index.htmlをコピーしてindex_copy.html

を作成してコミットしたとします。Sourcetreeでindex_copy.htmlを右クリックし、メニューから「選択のログを表示」（Macは「選択したファイルのログ」）を選択し、index_copy.htmlの編集履歴をログから確認すると、新規追加された時点までしか確認できません（図5）。

ここで、画面下部にある「**ファイル名の変更を追跡**」（Macは「名前が変更されたファイルに従ってください」）をチェックすると、コピー元のindex.htmlの編集履歴も表示されます（図6）。同様に、ファイル名が変更されたファイルの編集履歴をログで追うことで、ファイル名を変更する前のファイル編集履歴を確認することができる場合があります。

図5 複製したファイルの編集履歴



複製したファイルの編集履歴をログ画面で確認します。

図6 ファイル名の変更履歴を確認



「**ファイル名の変更を追跡**」をチェックすると、コピー元のindex.htmlの編集履歴も表示されます。

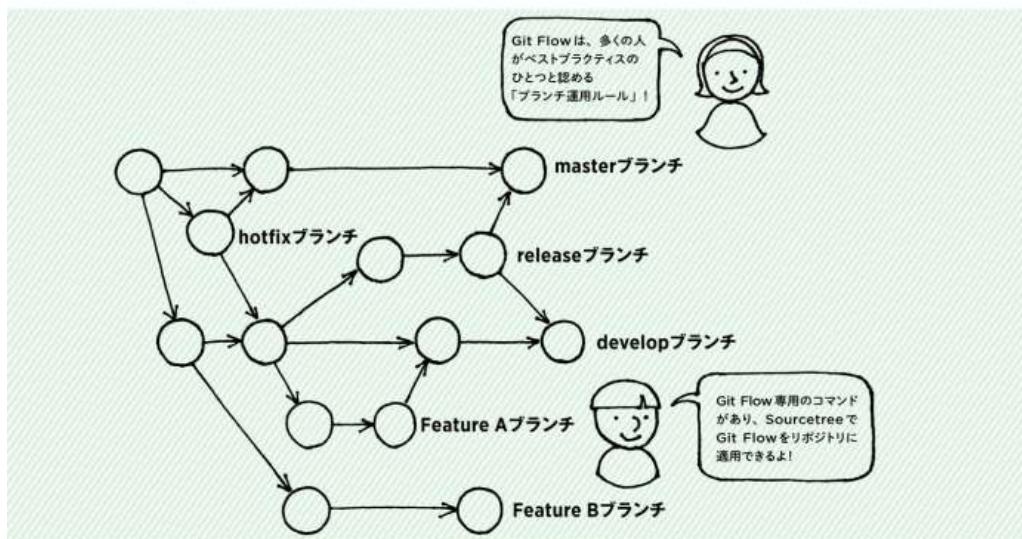
4-08

Git Flowを使った運用

Gitを利用する大きなメリットのひとつがブランチでしょう。

ただし、いたずらにブランチを増やしてしまうのは混乱のもとです。

特に複数スタッフで共同作業する場合は何らかのルールが必要になります。



- Git Flowのブランチ操作の方法を学ぶ
- Gitの代表的なブランチ運用のワークフローを学ぶ

使用するコマンド
\$ git flow [Git Flowコマンド]

使用するSourcetreeの機能



▶ Git Flowとは

Gitの使い始めはmasterブランチのみで運用し、ブランチは必要最低限で作成すると感じるかもしれません。Gitを使い慣れてくるとブランチの利便性に気づき、多用し始めるはずです。ただ、ブランチを複数作成し始めると、制作作業を進めていくうちにブランチが散乱して管理が困難になってしまふこともあるでしょう。

特に複数人で共同作業するような現場で複数のブランチを運用するときは、何らかのルールを設けることが大切です。制作現場によって、さまざまなルールが考えられますが、そのひとつの解が「**Git Flow**」です。

Git Flowはブランチ運用ルールのひとつで、多くの人にベストプラクティスのひとつであると認められたものです。現在では、Git Flow専用コマンドで運用することができるようになっており、Sourcetreeでも作成したりボジトリをGit Flowで運用できるようにするために画面上部に「Git Flow」ボタンが用意されています。

実際にGit Flowを導入するかどうかは別として、まずはGit Flowで何ができるのかを知ることは、ブランチ運用ルールを考えるにあたって大きなヒントとなります。大まかにいって、Git Flowでは次のような課題が解決できます。

◎ Git Flowで解決できる課題

(1) ブランチが煩雑になる、ブランチ作成のルールが定まらないという問題が解決できる

(2) バグフィックスのマージ漏れや不必要的機能のマージを回避できる

(3) Fast Forwardブランチ上の作業のトレーサビリティが担保される

▶ Git Flowはどういう運用ルールなのか

Git Flowは、主に以下の表1のような4つのブランチを使用したブランチ運用ルールです。

Git Flowではブランチごとに用途、目的が明確に定まっており、作業したいときは目的に応じたブランチを作成してから行うというのが基本ルールです。

また、たとえばmasterに公開したあとで、問題が発覚して修正したくなった場合は「hotfix」というブランチを作

成して作業を行うというルールも含まれます。

軽微な作業であればブランチの作成は少し面倒に感じるかもしれません。masterブランチで直接作業するのではなく、1タスクごとにブランチを作成することが重要です。Git Flowは、原則としてdevelopやmasterに直接コミットしない、というルールにより成り立っているのです。

表1 Git Flowの主な運用ルール

ブランチ	説明
developブランチ	作業を行うためのブランチで、主にこのブランチ上で作業を行う。このブランチに直接コミットするのではなく、ほかのブランチで行った作業をマージするためのブランチ。
featureブランチ	ひとつの作業単位に該当するブランチで、1タスク毎にブランチを作成する。修正作業や、機能追加など、タスクごとにfeatureブランチを作成して作業を行う。
releaseブランチ	公開の準備を行うためのブランチ。公開する前にdevelopから派生してこのブランチを作成して準備、調整を行う。releaseブランチを作成することで、リリース前の準備を行いながら次バージョン向けの作業を並行できる。リリース準備が完了したら、このブランチからリリースタグを作成して、masterブランチにマージする。
masterブランチ	公開されているバージョンを管理するためのブランチ。作業者はこのブランチへのコミットは行わない。

▶ Git Flowを使用した作業の例

ここでは、Sourcetreeを使ったGit Flowによる運用ルールを適用した作業事例を簡単に紹介します。作業内容としては、「issue1」という作業を行って、v1.0という名前のバージョンとして公開する」という事例です。

Git Flowを使用するための準備

まずは作成済みのリポジトリをSourcetreeで開き、「リ

ポジトリ」メニュー→「Git Flow」→「リポジトリを初期化」を選びます^{図1}。「Git Flow用にリポジトリを初期化」ダイアログが表示されます^{図2}。このダイアログでは、Git Flowで慣例的に使用されるプランチ名があらかじめセットされています。任意に変更してもかまいませんが、基本的にはデフォルトのまま「OK」ボタンをクリックします。Git Flow用にリポジトリが初期化されます^{図3}^{図4}。

図1 Git Flowでリポジトリを初期化



「リポジトリ」メニュー→「Git Flow」→「リポジトリを初期化」を選びます。画面下部の「Git Flow」ボタンをクリックしても同じです。

図3 初期化作業メッセージ



featureプランチで作業を行う

作業を開始するにあたって、まずはdevelopプランチからfeatureプランチを派生させます。Git Flowでは原則としてdevelopやmasterに直接コミットしない、というルールを思い出してください。

「リポジトリ」メニュー→「Git Flow」→「新規フィーチャーを開始」を選び^{図5}、「新規フィーチャーを開始」ダイアログでフィーチャー名に「issue1」と入力して「OK」ボタンをクリックします^{図6}。

図2 Git Flow用初期化ダイアログ



プランチ名はデフォルトのまま「OK」ボタンをクリックします。

図4 初期化完了



リポジトリの初期化が完了すると、developプランチが追加されます。

作業用のfeatureプランチを作成



「リポジトリ」メニュー→「Git Flow」→「新規フィーチャーを開始」を選びます。

これで、master、developに加えて、feature/issue1というプランチが新たに作成され、チェックアウトされた状態となります^{図7}。

何らかの編集作業を行い、作業内容をfeature/issue1プランチでコミットします^{図8}^{図9}。

featureプランチをdevelopプランチにマージ

feature/issue1プランチでの作業が完了したので、developプランチにマージして、feature/issue1プランチを削除します。このとき、通常のSourcetreeでの操作とは異なり、専用メニューで行います。

「リポジトリ」メニュー→「Git Flow」→「フィーチャープランチを完了」(Macは「Featureを終了」)を選びます^{図10}。

「フィーチャープランチを完了」ダイアログが表示されるので、プランチを削除がチェックされていることを確認して「OK」ボタンをクリックします^{図11}。これで、feature/issue1プランチがdevelopプランチにマージされ、同時にfeature/issue1プランチが削除されます^{図12}。

図6 初期化完了



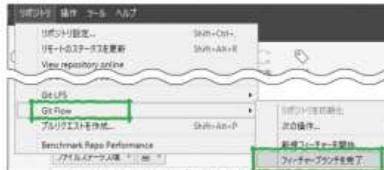
「新規フィーチャーを開始」ダイアログでフィーチャー名を入力します。

図8 featureプランチでコミット



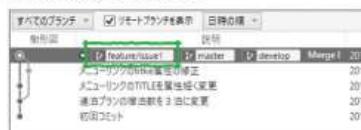
feature/issue1プランチで作業を行い、変更内容をステージに追加して、コミットします。

図10 フィーチャープランチを完了



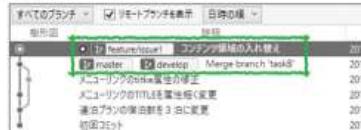
「リポジトリ」メニュー→「Git Flow」→「フィーチャープランチを完了」を選びます。

図7 featureプランチ作成完了



feature/issue1というプランチが新たに作成され、チェックアウトされました。

図9 featureプランチでコミット完了



コミットした結果、feature/issue1プランチがmaster、developプランチよりも先行します。

図11 「フィーチャープランチを完了」ダイアログ



プランチを削除がチェックされていることを確認して「OK」ボタンをクリックします。

図12 developプランチへのマージが完了



feature/issue1プランチがdevelopプランチにマージされ、同時にfeature/issue1プランチが削除されます。

releaseブランチで公開準備を行う

作業内容の公開に向けてリリース作業を行います。Git Flowでリリース作業を行うのは**release**ブランチです。

「リポジトリ」メニュー→「Git Flow」→「新規リリースを開始」を選びます^{図13}。「新規リリースを開始」ダイアログが表示されるので、リリース名を「v1.0」と入力し、「最新の開発ブランチ」がチェックされていることを確認して「OK」ボタンをクリックします^{図14}。

これで、**develop**ブランチから派生した**release/v1.0**ブランチが作成されます。**release**ブランチでは、必要に応じて経微な調整作業を行うこともありますが、基本的にそのまま公開作業へ移行します。

図13 リリース作業を開始

「リポジトリ」メニュー→「Git Flow」→「新規リリースを開始」を選びます。

「リポジトリ」メニュー→「Git Flow」
→「リリースを完了」を選びます。

図15 リリースを完了して公開作業を開始

ブランチを削除がチェックされていることを確認して「OK」ボタンをクリックします。

リリース(公開)作業を完了する

releaseブランチの準備が完了したら、公開作業へ移行します。「リポジトリ」メニュー→「Git Flow」→「リリースを完了」(Macは「Releaseを終了」)を選びます^{図15}。「リリースを完了」ダイアログで、ブランチを削除がチェックされていることを確認して「OK」ボタンをクリックします^{図16}。このダイアログでは**master**ブランチにつけるタグを指定することもできます。特に指定しなければ、リリース名がタグとして付与されます。

release/v1.0ブランチが**master**ブランチにマージ(同時に**develop**ブランチにマージ)され、**release/v1.0**ブランチは削除されます。リリースを完了した後のコミットツリーでは、**master**ブランチにリリース名のタグがつき、最新コミットとして先行した状態となります^{図17}。

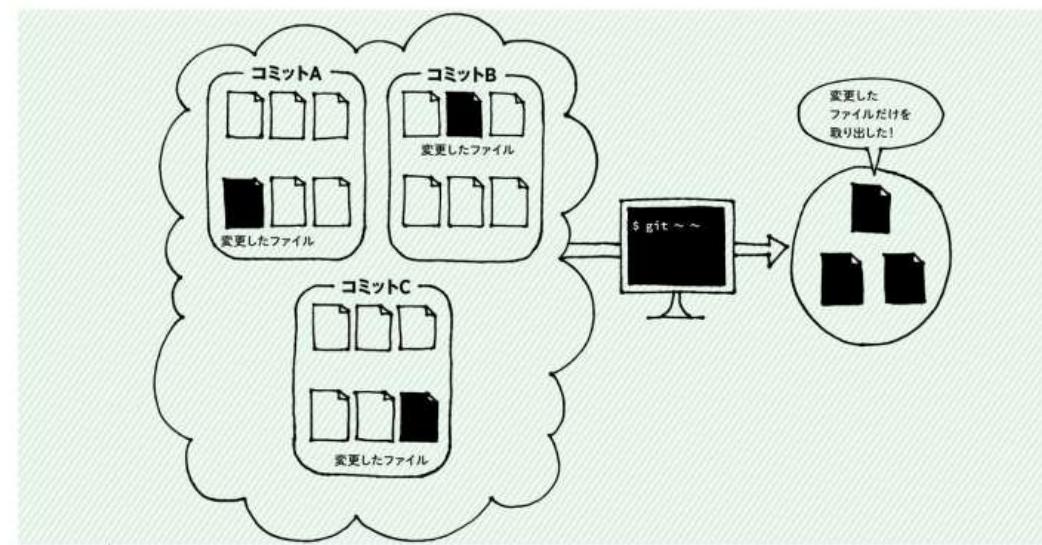
これでGit Flowによる公開作業はすべて完了しました。

4-09

◎さらに高度な応用編

便利なGitの機能を活用

本節と次節は、Sourcetreeで簡単に操作できる範囲にとどまらず、Gitのより高度で便利な機能を紹介します。ターミナルを使った操作やコマンド入力にも踏み込んで、Git本来の幅広い活用方法も紹介しますので、ぜひ一度触れてみてください。



- ✓ ターミナルでの操作にチャレンジしてみる
- ✓ Gitのより高度で便利な活用方法に触れる
- ✓ 作業履歴の掘り下げ方や差分の適用方法にもそれぞれ複数の手段があり、状況によって方法を選ぶことができる

使用するコマンド

```
$ git diff --name-only --diff-filter=ACM [部分をとるコミットID]
$ git archive --format=zip HEAD -o [出力ファイル名] [アーカイブするファイル名]
$ git blame [履歴を行ごとに確認したいファイル名]
$ git format-patch [パッチを作成するコミットID]
$ git am [適用するパッチファイル名]
```

使用するSourcetreeの機能

ターミナル
(Mac: 編集)

▶ 差分ファイル一覧を取り出す

Webサイトの更新作業などを行っている際、更新のために作業した差分ファイルをクライアントに報告しなければならないこともあります。差分ファイルを手作業でリストアップしていくは漏れが生じる可能性もあり、確実性に乏しいといえます。Gitを利用すれば、あるコミットからほかのコミットの間で作業対象となったファイル一覧を取り出すのも容易であり、また確実です。

この作業には、いわゆる「黒い画面」とよばれるターミナルでのコマンドベースの操作が必要です。ターミナルはSourcetree画面上部にある「ターミナル」(Macでは「端末」)ボタンをクリックして呼び出すことができます^{図1}。SourcetreeのターミナルはLinuxのシェルエミュレータとなっているため、GitコマンドおよびLinuxコマンドを使



画面上部にある「ターミナル」ボタンをクリックするとターミナルが起動します。

差分ファイル一覧抽出用コマンドの詳細

先ほどのコマンドを詳しく解説しましょう^{図3}。冒頭にある「git diff」は差分情報を取り出すための基本コマンドです。そのオプションとして続く「--name-only」はファイル名だけの一覧にするという指定になります。

さらに続く「--diff-filter=ACM」というオプションは差分抽出する際のフィルター指定で、「ACM」は以下のようない意味となります。

- A : Added (追加)されたファイル
- C : Copied (コピー)されたファイル
- M : Modified (変更)されたファイル

図3 差分ファイル一覧抽出用コマンド



用することができます。ターミナルが起動したら、以下のようないコマンドを入力して[Enter]キーを押します。

```
$ git diff --name-only --diff-filter=ACM 4557646  
HEAD
```

このコマンドを実行した結果、ターミナル画面にはいくつかのファイル名が表示されます^{図2}。

***注意:** ターミナルの基本的操作などについては、本書では解説しておりません。また、GitコマンドおよびLinuxコマンドの詳細についても本書では掲載しておりません。詳しく知りたい方は別の専門書籍などをご参照ください。

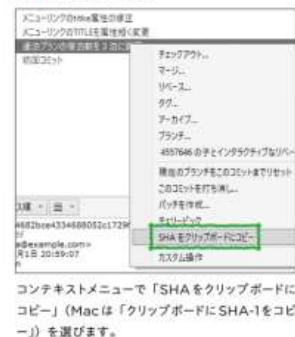
▶ コミットIDの取得方法

このコマンドでポイントとなるのは、始点、終点それぞれのコミットIDです。コミットIDとは、コミットが生成されたときにユニークな値として付与される番号で、Sourcetreeのコミットツリー右端にある「コミット」という列に表示されている文字列です^{図4}。

このコミットIDを取得するには、そのコミットを右クリックして表示されるコンテキストメニューで「SHAをクリップボードにコピー」(Macは「クリップボードにSHA-1をコピーします」)を選びます^{図5}。あとはターミナル画面でペースト(Windowsなら[ALT]+[V]、Macなら[command]+[V])するだけです。

なお、このコマンドでは終点に始点にあるような文字列ではなく「HEAD」と指定しています。先に説明したように「HEAD」は現在チェックアウト中のコミットを示す特別なIDです。終点に、現在チェックアウトしているコミットではなく、ほかのコミットを指定したい場合は、コミットツリーから目的のコミットIDを取得して入力してください。

図5 コミットIDのコピー



コンテキストメニューで「SHAをクリップボードにコピー」(Macは「クリップボードにSHA-1をコピーします」)を選びます。

▶ 図4 コミットIDが表示されている場所



コミットIDは、Sourcetreeのコミットツリー右端にある「コミット」という列に表示されています。

▶ 便利なコマンドは「カスタム操作」に登録

ターミナルで頻繁に利用するコマンドは、Sourcetreeの「カスタム操作」に登録すればより手軽に利用できるようになります。毎回、ターミナルを起動してコマンドを入力するという手間を省けるだけでなく、入力ミスをなくすというメリットもあります。

カスタム操作にコマンドを登録する

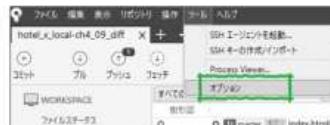
カスタム操作にコマンドを登録する場合は、まずSourcetreeの「ツール」メニュー→「オプション」(Macは「Sourcetree」メニュー→「環境設定」)を選びます(次ページ図6)。「オプション」画面が表示されたら、「カスタム操作」(Macは「カスタムアクション」)タブ画面に切り替えます^{図7}。「追加」ボタンをクリックすると「カスタム操作を編集」画面が表示されます^{図8}。この画面で、先ほど解説した「追加変更したファイル名一覧」を取得するコマンドを登録してみましょう。「メニュー表示名」

(Macは「メニュー名」)に適当な名称を入力し、「実行するスクリプト」には「git」と入力します。「パラメータ」には先ほどのコマンドのgit以降の部分を入力しますが、1カ所異なる点があります。先述のコマンドでは始点となるコミットIDを指定していますが、このままでは常にこのコミットIDが始点となってしまいます。**差分抽出用の始点は任意に変更したい**ため、この部分は「\$SHA」とします。こうすることで、コミットツリー上で選択したコミットのIDが自動的に指定されるようになります。

***注意:** Windowsの場合、システムのコマンドライン(SourcetreeではなくWindowsの「コマンドプロンプト」)からGitが利用できるようにパスを設定しておく必要があります。なお、P.019の方法でSourcetreeといっしょにGitをインストールした場合、パスは設定されません。独自にWindowsのパスを設定するか、別途Gitをインストールしてパスを設定する必要があります。

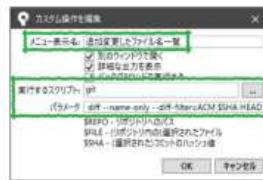
また、このコマンドで得られるファイル名リストはコピーして利用したい場合、「詳細な出力を表示」(Macは「フル出力を表示」)をチェックしておきましょう。すべての指定を終えたら「OK」ボタンをクリックします。カスタム操作

図6 オプション画面を開く



「ツール」メニュー→「オプション」(Macは「Source tree」メニュー→「環境設定」)を選択します。

図8 「カスタム操作を編集」画面でコマンド登録



カスタム操作に登録したコマンドを実行する

早速、登録したコマンドを呼び出してみましょう。

コミットツリーから差分のファイル一覧を取り出したい始点となるコミットを選び、右ボタンクリックで表示されるコンテキストメニューから「カスタム操作」→「追加変更したファイル名一覧」(Macは「カスタムアクション」→「リポジトリのアクション」→「追加変更したファイル名一覧」)

図10 登録したコマンドを呼び出す



図11 コマンドの実行結果



タブ画面に戻るので、登録したメニュー名が表示されていることを確認して「OK」ボタンをクリックしましょう(図9)。これで、カスタム操作の登録は完了です。

図7 「カスタム操作」タブ画面に切り替える



オプション画面で「カスタム操作」(Macは「カスタムアクション」)タブをクリックして切り替え、「追加」ボタンをクリックします。

図9 コマンド登録を確認する



カスタム操作タブ画面に、今登録したメニュー名が表示されています。

▶ 差分ファイルを一式zip圧縮して取り出す

先 に紹介したコマンドではファイル名を一覧で取り出していましたが、今度は対象となるファイルだけを一式zip圧縮して取り出すコマンドを紹介しましょう。

Webサイトの更新作業などでは、変更を加えたファイルのみ、つまり差分ファイルを納品しなければならないケースは多くあります。このコマンドを活用すれば、手作業によるミスをなくし、信頼性の高い納品作業を実現することができます。

コマンドは以下のようにになります。このコマンドをSourceTreeのターミナルを起動して実行します。

```
$ git archive --format=zip HEAD -o ../../diff.zip
$ git diff --name-only 4557646 HEAD
```

コマンドの内容を解説します。冒頭にある「git archive」はファイルをアーカイブする基本コマンドです。そのオプションとして続く「--format=zip」はアーカイブ形式にzipを指定しています。次の「HEAD」はファイル内容を現在チェックアウトされているコミットの状態とする指定です。次の「-o ../../diff.zip」は取り出した結果と

してのzip圧縮ファイルの保存先ファイル名を指定しています。

最後の「`git diff --name-only 4557646 HEAD`」は、先述のファイル名一覧を取得するコマンドを「`」(パッククオート)で囲んだものです。つまりアーカイブ化する対象を抽出したファイル名一覧をarchiveコマンドに渡していることになります。

MEMO 保存先の指定

このコマンドではzipファイルの保存先を「-o ../../diff.zip」としています。SourceTreeではターミナルは作業しているリポジトリ管理下の最上位ディレクトリにいる状態で起動します。したがって、このコマンドで「-o ./diff.zip」とファイル名だけを指定すると、リポジトリ管理下のディレクトリにzipファイルが作成されてしまい、Gitで未登録の追加ファイルと認識されてしまいます。そこで「./diff.zip」としてひとつ上のディレクトリ階層に保存するように指定しているわけです。

▶ 不具合発生の発端となったコミットを探る

W ebサイト制作やプログラム制作において、何らかの不具合が発生した場合、その不具合を解決することはもちろんですが、クライアントに対して障害報告を行う必要もあります。また、プログラム上の不具合の原因を見つけるだけでなく、ミスが発生した状況を確認することで、同様のミスを再び起こさないように作業環境の改善を図ることにも役立ちます。

その不具合が、ファイルの中の特定の箇所の記述が原因であることが判明した場合、その記述変更を行った履歴をたどり、そもそもどの時点で発生していたかを追跡することは、責任の範囲を確認するうえでも重要です。

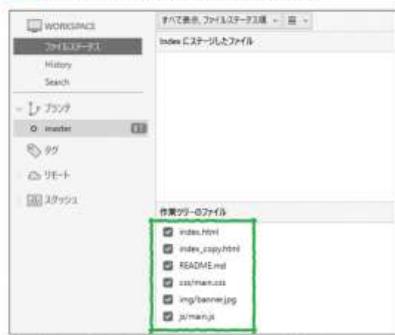
Gitでは「Blame」という機能で、任意の変更箇所を行単位の履歴を追跡することができます。具体的に見てみましょう。

SourceTreeのファイルステータス画面で、絞り込み用のブルダウンメニューで「すべて」を選び、すべてのファ

イルをリストアップします(次ページ図12)。次に、不具合の原因となっているファイルを右クリックして、コンテキストメニューから「選択項目にアノテーションを付ける」を選択します(図13)。すると、そのファイルの内容が表示され、ファイル内の各行ごとにコミットIDが表示されます(図14)。続けて、不具合の原因となっている該当行のコンテキストメニューから「ファイルログウインドウで表示」を選択します(図15)。するとログ画面が表示され、該当の行を最後に変更したコミットを特定することができます(図16)。

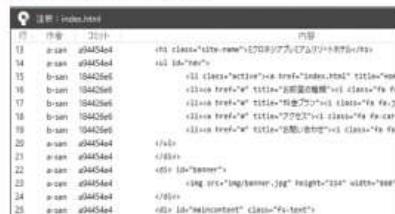
さらに、その行の過去の変更にさかのぼるには、ある行を最後に変更したコミットのひとつ前のコミットの該当ファイルから、再度「選択項目にアノテーションを付ける」を選択します(図17)。Blameには「責任を問う」と「がめる」という意味があります。その名の通り、この機能で誤りの原因になったコミットをいつ誰が行ったのか、責任の所在を明らかにできるのです。

図12 ファイルステータス画面に全ファイルを表示



絞り込み用のプルダウンメニューで「すべて」を選び、すべてのファイルをリストアップします。

図14 行ごとのコミットIDを表示



ファイル内の各行ごとにコミットIDが表示されます。

図16 その行のコミット履歴を確認



その行のコミット履歴が表示されるので、最後に変更したコミットを特定することができます。

MEMO パッチファイルの作成について

Sourcetreeのパッチ作成画面では複数のコミットを選択できます(次ページ図20)。ただし、パッチファイルは「1つのコミットに対して1つのパッチファイル」が原則です。したがって、複数コミットを選択する場合は、パッチ作成画面の下部にある「コミットごとに分割したパッチを作成」をチェックして、1コミットごとのパッチファイルを作成しましょう。また、パッチファイルを適用する際は、古いコミット用のパッチファイルから順番に適用しましょう。

図13 選んだファイルのBlameを表示させる



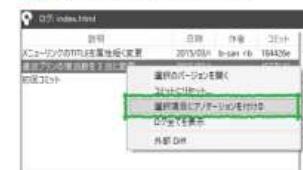
コンテキストメニューから「選択項目にアノテーション付ける」を選択します。

図15 ログウィンドウを表示



コンテキストメニューから「ファイルをログウィンドウで表示」を選択します。

図17 さらに過去の変更にさかのぼる



ひとつ前のコミットの該当ファイルから再度「選択項目にアノテーション付ける」を選択します。

▶ ブッシュ、プルができる環境に差分を適用する

Gitは変更内容を適用するパッチを作成することもできます。何らかの都合により、リポジトリ同士で直接通信ができない、つまりブッシュ、プッシュができるないような環境に対しても、修正差分を適用することができるのです。

パッチファイルは、適用したいコミットを選択して作成します。

実際に手順を見てみましょう。図18 のようなりポジトリでパッチファイルを作成してみます。まずは「操作」メニュー→「パッチを作成...」、もしくはコミットツリー上で右クリックして表示されるコンテキストメニューで「パッチを作成...」を選びます図19。「パッチを作成」画面が表示されたら、「ログ」タブ画面に切り替え(Macは「コミットからパッチを作成」タブ)、コミットを選択して「パッチを作成

ボタンをクリックします図20。この操作で、リポジトリで管理しているディレクトリにpatch.diffという名前のパッチファイルが生成されます。

次に、出力したパッチファイルを別の環境に適用してみます。USBメモリなどでpatch.diffを別のPCなどにコピーします。その環境でSourcetreeを起動し、パッチを適用したいリポジトリを開きます図21。「操作」メニュー→「パッチを適用...」を選びます。「パッチを適用」画面が表示されたら、パッチファイル(patch.diff)を指定し、モードを「完全なコミットとして取り込む」にセットして「パッチを適用」ボタンをクリックします図22。これで、その環境にパッチが適用され、プルしたと同様にリポジトリの状態が更新されます図23。

図18 パッチを作成するリポジトリの状態

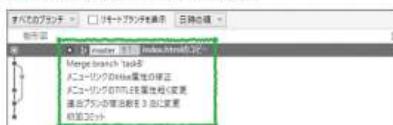


図20 パッチに含めるコミットを選択

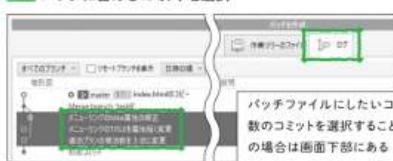


図22 パッチファイルを適用

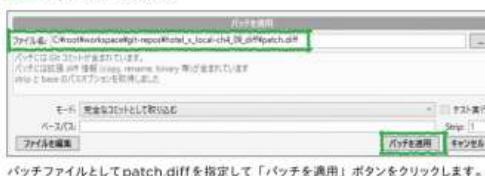


図23 パッチの適用が完了



図19 コンテキストメニューで「パッチを作成...」を選択



図21 パッチを適用する前のリポジトリの状態



MEMO Macの場合

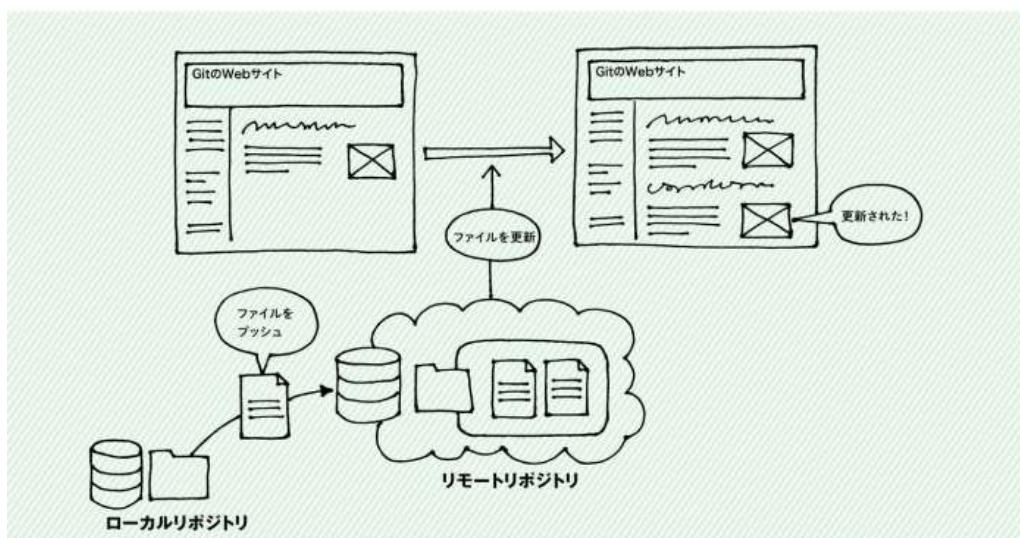
Sourcetreeのバージョンによっては、生成されたパッチを適用するためにパッチファイルの整形が必要です(そのまま適用すると「無効なパッチファイル」と表示されます)。詳しくは下記のATLASSIANのページを参照ください。

<https://community.atlassian.com/t5/Sourcetree-questions/quot-This-is-not-a-valid-patch-file-quot-for-a-patch-file-that-1/qaq-p/965040>

◎さらに高度な応用編

Webサイトの更新をGitで行う

本書の読者のみなさんが、一番知りたいのがWebサイトの制作データをGitで管理するということでしょう。データ管理自体は、今まで解説してきた通りです。ここでは、Webサイトの更新をGitでどのように行うのかを紹介します。



- Bitbucketのような中央リポジトリを独自に作って連携できることを知る
- ノンベアリポジトリにプッシュするなどのようなことが起きるのか理解する
- Webサイト上のファイルの更新をGitで行う方法を知る

使用するコマンド

```
$ git init --bare
$ git push [リモートリポジトリ] [ローカルリポジトリの送信元ブランチ]:[リモートリポジトリの送信先ブランチ]
```

使用するSourcetreeの機能



▶ 独自の中央リポジトリの活用

公 開しているWebサイトをGitで更新することもできます。たとえば、Webサーバ上でサイトデータを格納しているディレクトリをリモートリポジトリにして、ローカルPC上のリポジトリと連携させればよいことになります。ただ、その場合はサーバ上にGitがインストールされており、ローカルPCからSSHなどで接続してGitコマンドがらも操作することができます。

利用できる必要があります。サーバへのGitのインストールやSSHアクセス方法などは環境によって異なるため、本書では取り扱いません。すでに、そのような環境が準備されていることを前提に解説しますが、使用するGitコマンドはそのまま利用でき、Sourcetreeのターミナルからも操作することができます。

▶ Bitbucket以外のリポジトリにプッシュする

本 書で紹介してきたBitbucket上のリポジトリと同様に、独自に作成したリポジトリに対してもプッシュすることができますが、一部制約があります。Gitで管理するリポジトリは大きく**(bare) リポジトリ**と**(non-bare) リポジトリ**に分かれます。

ツッショするには、ペアリポジトリである必要があります。**BitbucketやGitHub**などに作成するリポジトリは、基本的にすべてペアリポジトリです。作業コピーがないため、ペアリポジトリ内ではコミットは一切できません。ローカルリポジトリなどからツッショすることでコミットを更新していくだけです。

ペアリポジトリとは

ペアリポジトリはノンベアリポジトリとは異なり、作業コピー（作業ディレクトリ）を持っていません。作業環境であるローカルPCに作成したローカルリポジトリには作業コピーがあります（つまりノンベアリポジトリです）。

ノンベアリポジトリには、実際に作業中のプランチが存在しており、プッシュによって作業内容が破壊されてしまうことを防ぐため、通常はプッシュすることができません。ブ

SSH接続のできるターミナルからWebサーバにアクセスしてペアリポジトリを作成します。ここでは、例として「webserver_repo.git」というディレクトリを作成してみます。

```
$ mkdir webserver_repo.git
$ cd webserver_repo.git
$ git init --bare
```

◎通常のリポジトリ

- ・作業コピー（HEAD）がある
- そのため、チェックアウトやコミットができる

◎ペアリポジトリ

- ・作業コピー（HEAD）がない
- そのため、すべてのプランチに対してプッシュできる。中央リポジトリとして利用するのに便利。
- ファイルの実体が存在していないので、チェックアウトやコミットができない。作業することはできない。

1行目の「mkdir webserver_repo.git」でディレクトリを作成。2行目の「cd webserver_repo.git」で、作成したディレクトリに移動しています。

そして3行目の「git init --bare」でペアリポジトリを作成します。「git init」コマンドでリポジトリを作成しますが、これに「--bare」というオプションをつけることでペアリポジトリとして作成することができます。

ペアリポジトリを登録

作成したペアリポジトリをSourcetreeでリモートリポジトリとして登録します。SourcetreeでWebサイト制作用のリポジトリを開いた状態で、画面上部左端にある「設定」ボタンをクリックして「リポジトリ設定」ダイアログを表示します（次ページ図1）。

「リモート」タブ画面で「追加」ボタンをクリックして「リモートの詳細設定」画面に切り替え、リモート名とURL/パスを設定します^{図2}。

このとき、URL/パスにはWebサーバへのSSHでの接続情報を入力します。そして「OK」ボタンをクリックすれば登録が完了し、リモート一覧に登録したリモートリポジトリが表示されます^{図3}。

図2 リモートリポジトリの情報を指定

任意のリモート名をつけて、URL/パスにはSSHでの接続情報を入力します。

独自のベアリポジトリにプッシュする

作成、登録したリモートリポジトリには、まだプランチもコミットも存在していません。そこで、ローカルの作業内容をプッシュします。

Sourcetreeのリモート一覧にあるリモートリポジトリ名を右クリックして表示されるコンテキストメニューから「web_server(リモート名)にプッシュ」を選びます^{図4}。

「プッシュ」画面が表示されるので、リモートプランチに

図4 リモートリポジトリにプッシュ

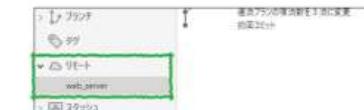
コンテキストメニューから「web_server(リモート名)にプッシュ」を選びます。

図6 SSH接続の認証画面

ユーザー名とパスワードを入力して「Login」ボタンをクリックします。

図1 リポジトリ設定

「追加」ボタンをクリックして「リポジトリ設定」ダイアログを表示します。

図3 リモートリポジトリの登録完了

リモート一覧に登録したリモートリポジトリが表示されます。

masterと指定し、「追跡中?」をチェックした状態で「OK」ボタンをクリックします^{図5}。SSH接続による認証画面が表示されるので、ユーザー名とパスワードを入力して「Login」ボタンをクリックします^{図6}。認証に成功するとプッシュが行われ、リモートリポジトリにローカルの作業内容が同期されます^{図7}。

「プッシュ」画面が表示されるので、リモートプランチに

図5 「プッシュ」画面

リモートプランチにmasterと指定し、「追跡中」をチェックして「OK」ボタンをクリックします。

図7 リモートリポジトリへの初回プッシュが完了

リモートリポジトリにローカルの作業内容が同期されます。

▶ Webサーバ上のファイルをGitのプッシュで更新する

際には、ノンベアリポジトリに対してもプッシュすることは可能です。ただし、ただプッシュしただけではWebサイトにファイルがアップされません。リモートリポジトリ側の設定として、プッシュされたら作業コピーのファイルに対して反映するようにしなければなりません。そのためには、「hook」というコマンドを使って設定を行います。

ノンベアリポジトリを用意する

まずはサーバ上にノンベアリポジトリを用意しましょう。サーバ上にある既存のリポジトリをクローンするか、手元のリポジトリをそのままアップロードしてもいいでしょう。

たとえば、「webserver_non_bare.git」というディレクトリに対してクローンする場合は、ターミナルでサーバにアクセスして図8のコマンドを実行します。

クローン元となるリポジトリがない場合は、手元のリポジトリをフォルダごとコピー（アップロード）してもよいでしょう。ただし、その場合はコミットしていない変更内容がない状態である必要があります。

いずれかの方法で、サーバ上にノンベアリポジトリを作成してSourcetreeでリモートリポジトリとして登録します^{図9}。

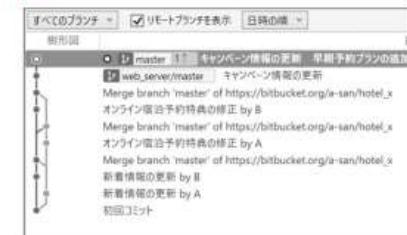
ノンベアリポジトリにプッシュしてみる

試しに、作成したノンベアリポジトリに対してプッシュしてみます。ノンベアリポジトリではチェックアウトされているプランチにはプッシュできません。この例ではmasterプランチがチェックアウトされているので、master_pushというプランチにプッシュします。

Sourcetreeのプッシュ画面で、リモートプランチに「master_push」と指定し、「追跡中」もチェックして「OK」ボタンをクリックします^{図10}。これにより、master_pushプランチは同期されますが、サーバ上のファイルはmasterプランチの状態を示しているため、プッシュする前の状態のままで^{図11}。

図8 Gitのクローンコマンド

```
git clone [リポジトリのURL] webserver_non_bare.git
```

図9 クローンもしくはコピー後のリポジトリの状態**図10** ノンベアリポジトリにプッシュ

チェックアウトされているmasterプランチにはプッシュできないので、master_pushというプランチにプッシュします。

図11 プッシュ後のコミットツリー

master_pushとローカルのmasterプランチが同期しますが、リモートの作業コピーはmasterのままで更新内容は反映されません。

Hook設定を行って作業コピーにも変更を反映する

では、Hook設定を行ってみましょう。リポジトリ内に`.git/hooks/`というフォルダがあるので、まずはその中に`post-receive`というファイルを作ります。これがHook設定用のテキストファイルとなります。内容は図12のようなテキストを入力しておきます。この設定ファイルの内容は、「プッシュを受け付けたら master_push ブランチをマージする」という意味を持ちます。

なお、このファイルをアップしただけでは実行されません。ファイルのパーミッションを変更して実行権限を追加しなければなりません。パーミッションを変更するには、ターミナルでWebサーバにアクセスして、図13のコマンドを実行^{*}します。

図12 Hook設定用のテキストファイル「post-receive」

```
#!/bin/sh
exec git merge master_push
```

図13 「post-receive」に実行権を設定

```
cd webserver_non_bare.git/
chmod +x .git/hooks/post-receive
```

図14 Hook設定後にプッシュ



図15 フェッチしてリモートブランチの状態を確認



▶ チェックアウトされているブランチに直接プッシュ

は、Gitにはチェックアウトされているブランチに対するプッシュを受け入れるような設定が存在します。しかし、その設定ではプッシュできるだけで、作業コピーが書き換わるわけではありません。つまり、Webサイトにファイルをアップできるわけではなく、期待したような結果にはなりません。

実際にどのようなことが起るのか、試してみましょう。

まずは設定を行わずに、ノンペアリポジトリにそのままプッシュするとエラーになります図16。

そこでチェックアウト中のブランチも強制的にプッシュするように設定を変更します。ターミナルでWebサーバにアクセスして、以下のコマンドを実行します。

```
$ cd webserver_non_bare.git/
$ git config receive.denyCurrentBranch ignore
```

1行目でリモートのノンペアリポジトリのディレクトリに移動し、2行目で現在チェックアウトされているブランチへのプッシュ受け入れを許可するようなコマンドを実行しています。

この状態で再度プッシュを行うと、今度はエラーとならずには完了します図17。

ここでコミットツリーを見ると、確かにプッシュは完了していますが、実際にはリモートリポジトリ上の作業コピーは書き換わっていません。つまり、コミット履歴は更新(変更が記録)されているのに、作業コピーには反映されないという異常な状態となります。したがって、このリモートリポジトリ上の作業コピーでコミットを行うと変更前の状態に戻る、つまりRevert(打ち消し)と同様の状態になってしまいます。意図しない結果となってしまうため、チェックアウト中のブランチへの強制的なプッシュは行わないことをおすすめします。ノンペアリポジトリへのプッシュは、先述したHook設定によって作業コピーへ反映するようになります。

図16 チェックアウト中のブランチにプッシュ



図17 強制的にプッシュした結果



プッシュは完了していますが、実際には作業コピーは書き換わっていません。

Column

Bitbucketへ作成した公開鍵を登録する



① 次に、Bitbucketにログインし、左下にあるアイコンのメニューから、「View profile」を選択します。次に、メニューから、「設定」→「SSH鍵」を選択します。



② P.178の手順②で作成した公開鍵の内容をコピーして、「Key」欄にペーストします。「SSHキーの作成／インポート」のダイアログを閉じてしまった場合は、再度ダイアログを表示し、「Load」ボタンをクリックして、秘密鍵「id_rsa.pub」を選択すると、「Key」欄に公開鍵の内容が表示されます。「Label」には任意の名前を入力してください。最後に「鍵を追加」をクリックします。

Bitbucketへの登録(Macの場合)



① Bitbucketにログインして上図まで進んでください。ここでターミナルを起動し、次のコマンドを入力します。(コマンドは上図の「with」以降に記載されています)。

```
$ cat ~/.ssh/id_rsa.pub | pbcopy $pbm
```

② 作成した鍵の内容がクリップボードにコピーされますので、「Key」欄にペーストして登録します。

自分のPCへの登録

自分のパソコンにおいても鍵を利用するための登録を行います（Macは不要）。



① Sourcetreeの「ツール」メニュー→「SSHエージェントを起動」を選択します。タスクトレイに、SSHエージェントのアイコンが表示されるので、これをダブルクリックします。

② 「Add Key」ボタンをクリックし、秘密鍵「id_rsa.pub」を選択して登録します。以上でBitbucketと自分のPCに公開鍵の登録が完了です。

最後にBitbucketでの確認



① リポジトリをクローンするときのポップアップに「SSH」と表示されます。クローン後は、SSHでリモート接続されます。

用語索引

記号・数字

「+」ボタン	028
<<<<< HEAD	095, 097
=====	097
>>>>> [コミットID]	095
.gitignore	070
「.git」フォルダ	024

A

Already up-to-date	086
Auto Merge	087

B

bare	173
Bitbucket	015
Blame	169
branch	013, 105

C

checkout	106, 133, 137
commit	012, 029, 049
Conflict	087

D

diff 形式	063
---------	-----

F

Fast Forward	087, 101
「Featureを終了」	163
fetch	058, 144

G

Git	017
git add	030
git archive	169
git commit	030
git config	092
git diff	166
Git Flow	161
GitHub	015
GitHub アカウントの追加	046
git init --bare	173
git reset --hard	068
git revert	142
git --version	017

「Hard」	068, 136
HEAD	116
「History」	049
Hook 設定	176
Hunk	054, 088
「Hunkをステージから除く」	155
「Hunkをステージへ移動」	055, 154
「Hunkをステージングに移動」	055, 154

I	Index にステージしたファイル	028
---	-------------------	-----

M	master	059, 106
merge	058, 085	
Merge Commit	089	
「Mixed」	068, 136	

N	non-bare	173
---	----------	-----

O	origin	059
origin/master	059	

P	pull	058, 061
push	058, 059	

R	reset	135, 137
Revert	137, 141	

S	「SHAをクリップボードにコピー」	167
「Soft」	068, 136	
Sourcetree	016	
SSHの公開鍵認証	178	
stash	113	

あ	「あなたの作業」	040
---	----------	-----

執筆者プロフィール

大串 肇(おおぐし・はじめ)

▶ <https://www.m-g-n.me>
▶ <https://themes-park.com>

株式会社mgn代表。WordPressを利用したWebサイトの制作および運用コンサルティングに従事。お客様と共に考え、行動し、「成果」を積み重ねるがモットー。クライアントの事業がより発展するよう、Webのスペシャリストとして提案や改善対応を行うことを心がけている。

主な著書に『いちばんやさしいWordPressの教本 人気講師が教える本格Webサイトの作り方』(インプレスジャパン・共著)、『現場でかならず使われているWordPressデザインのメソッド』、『プロが選ぶWordPress優良プラグイン事典』(共にMdN・共著)など。

久保靖賀(くぼ・やすか)

▶ <http://www.xpa.jp>

IT系出版社にて編集業務、ゲームメーカーにてゲームディレクション業務に携わった後、エディトリアル・Webデザイナーとして独立。現在、コンテンツ企画から制作・運用までワンストップサービスを提供する株式会社エクスパに参画し、主にWebコンテンツ開発に従事。個人事業として、IT情報関連の著作・編集業務も並行している。東京グラフィックデザイナーズクラブ“TGC”会員。

豊沢泰尚(とよさわ・やすたか)

▶ <https://www.sharingseed.co.jp>

有限会社シェアリングシード代表、Webアプリケーションエンジニア。情報理工学修士。個人事業としてWebシステム開発やサイト制作に多数関わり、その後2006年に有限会社シェアリングシードを立ち上げる。Web制作会社向けに技術協力や、開発技術の導入支援を行うほか、多くのWebシステムの開発を行っている。著書に『携帯+iPhone モバイルサイト制作術 実践的コーディング&デザイン完全ガイド』(MdN / 共著)など。

う
打ち消し 137, 141, 142

か
改行コード 092
隠しファイル(フォルダ) 024
「カスタムアクション」 167
「カスタム操作」 167
画像の変更履歴 065

き
「競合を解決」 097
「行をステージングから除く」 155

く
グラフ 064
「クリップボードにSHA-1をコピー」 167
「グローバル無視リスト」 071
クローン 035, 036, 076, 081

け
「現在のブランチに～をマージ」 106
「現在のブランチをこのコミットまでリセット」 135, 155

こ
「このコミットを打ち消し...」 141
「このリポジトリをクローンする」 076
コミット 012, 029, 049
コミットグラフ 150
コミット先行 051
コミットツリー 150
「コミット適用前に戻す」 141
「コミット」ボタン 029, 032, 050
コミットメッセージ 012, 029, 051
「コミットをただちに～プッシュする」 060
コンフリクト 061, 089, 091

さ
「作業コピーの親」 106
「作業ツリーのファイル」 028
「削除」 024, 026, 067

し
集中型 013
樹形図 064, 150
「新規フィーチャーを開始」 162
「新規リモート」 145

「新規リリースを開始」 164

す
「すぐにマージした変更をコミットする」 089
「スタッッシュ」 113
ステージ 028, 051
ステージングエリア 028
「ステージングに未登録のファイル」 031, 054
「ステージングを分割して表示」 031
「全てインデックスから除く」 067
「全てインデックスに追加」 054, 028

せ
「選択項目にアノテーションを付ける」 169
「選択した行をステージから除く」 155
「選択をインデックスから除く」 067
「選択をインデックスに追加」 028

た
「ターミナル」ボタン 166
「タグ」 111
「端末」ボタン 166

ち
チェックアウト 106, 133, 137
中央リポジトリ 075

つ
「追跡を停止する」 072
「追跡をやめる」 072

と
ドットファイル(フォルダ) 024

の
ノンペアリポジトリ 173

は
バージョン管理システム 011
「破棄」 067, 098, 139
バッチを作成... 171
バンク 088

ふ
「ファイルステータス」 028, 031
「ファイル名の変更を追跡」 159
「フィーチャーブランチを完了」 163

フェッчу 058, 144

フォーク 035
ブッシュ 058, 059, 079
「ブッシュ」ボタン 059, 079
ブランチ 013, 105
「ブランチ」ボタン 106
「ブランチを削除」 107
ブル 058, 061, 083
「ブル」ボタン 061
「ブルリクエスト」 040
ブルリクエスト 035
分散型 013

へ
ペアリポジトリ 103, 173
「変更をすぐに～にプッシュする」 060

ま
マージ 058, 083, 085
「マージ結果を直ちにコミット」 089
「マージ後そのままコミット」 089
「マージした変更を即座にコミット」 089

む
「無視」 072

ゆ
「ユーザーとグループのアクセス権」 081

り
「リセット」 067, 098, 139
リセット 135, 137, 142
リベース 142
リポジトリ 012, 015
「リポジトリ限定無視リスト」 070
「リポジトリ固有の無視リスト」 070
リポジトリブラウザ 025
「リポジトリを初期化」 162
リモートブランチ 079
リモートリポジトリ 015, 144
「リモートを追加」 145
「履歴」 049

ろ
ローカルリポジトリ 015, 023