WEB Engineer's Books

git SE SHOEISHA

エンジニアのための Gitの教科書 上級編

Git 内部の仕組みを理解する

河村 聖悟 著

本書に関するご質問、正誤表については、下記の Web サイトをご参照ください。

正誤表　　http://www.shoeisha.co.jp/book/errata/
刊行物 Q&A　　http://www.shoeisha.co.jp/book/qa/

インターネットをご利用でない場合は、FAX または郵便で、下記にお問い合わせください。

〒 160-0006　東京都新宿区舟町５
（株）翔泳社 愛読者サービスセンター
FAX 番号：03-5362-3818

電話でのご質問は、お受けしておりません。

※本書に記載された URL 等は予告なく変更される場合があります。
※本書の出版にあたっては正確な記述につとめましたが、著者や出版社などのいずれも、本書の内容に対してなんらかの保証をするものではなく、内容やサンプルに基づくいかなる運用結果に関してもいっさいの責任を負いません。
※本書に掲載されているサンプルプログラムやスクリプト、および実行結果を記した画面イメージなどは、特定の設定に基づいた環境にて再現される一例です。
※本書に記載されている会社名、製品名はそれぞれ各社の商標および登録商標です。
※本書の内容は 2015 年 12 月執筆時点のものです。

はじめに

Git を学んで、チーム運用や開発フローの設計をこなし、実際に運用をはじめてみると、思ってもみない問題に出くわす機会が増えてきます。チームメンバーが泣きついてきた問題は、大抵、解決するのにちょっとした工夫や、一捻りが必要な状況になってから相談されているケースがほとんどです。その時、最初に今の状態を把握して、その情報を基に解決するための低レベルな Git コマンドを調べたり、ブランチを駆使して履歴を基に戻せないか四苦八苦する事になります。

今の状態を把握し、対策すべきポイントを洗い出す時に必要となるのが、Git のバージョン管理の内部構造を知ることです。普段なにげなく利用しているコマンドが内部的にどう動いているのか、データ構造はどうなっているのか。「なぜ」動いているのかを理解する事で、あらゆる問題への対応への助走が格段に早くなります。また、ブランチの運用や普段の Git の運用が、内部の動きを頭に描きながら行う事で、格段に簡単に感じるようになります。

この上級編を通して、さまざまな基礎コマンドの動きが内部管理ファイルの状態をどう変更しているかを学び、ブランチの概念はどう内部管理されているかを知って履歴の変更を自由自在に頭に描けるようにしましょう、Index の仕組みや最も使う git add /git commit を自分で作りながら実際の挙動を学べば、コマンドの使いこなし・３つのエリア（ワーキングディレクトリ・ステージングエリア・リポジトリ）の行き来・ブランチの分岐統合を自由自在に行えるようになります。

目　次

01 Git のバージョン管理の仕組みを知る ～初級編～

この節では、Git によるリポジトリの管理システムの動きを、Git コマンドの実行と対比しながら、理解を深めていきます。内部的な管理の動きを知ることで、各コマンドの意味を深く理解し、さらに使いこなすことができるようになります。

Git のバージョン管理の概要

Git のリポジトリの情報は、ファイルで管理されています。つまり、ファイルで構成されるデータストアです。Git では、ワーキングディレクトリで利用し編集している（ソースコードなどの）ファイルが、そのままの形で管理情報として格納されている訳ではありません。コミットしたファイルやディレクトリの情報は、コミットの情報・ファイルのメタデータ・データ・ディレクトリやパーミッションなどに分割され、それぞれにユニークな ID を採番して、別のデータとして管理します。ユニークな ID を採番されたデータは、「オブジェクト」という単位で管理されます。

コミットによって変更された履歴は、ブランチや各オブジェクト同士で参照し合っているオブジェクトの ID を変更することによって、柔軟に管理されます。つまり、ID を参照しながら管理情報の関係をつなぐ、連想記憶ファイル・システムです。これらの ID の参照は、その名前の通り「参照」として管理されます。例えば、コミットされたファイルはオブジェクトとなり、ファイルのメタデータを格納したオブジェクトから参照され、さらにメタデータの情報はコミット情報を格納したオブジェクトから参照されます。この連想記憶ファイル・システムは、重厚長大なデータベースより管理しやすく、サイズを小さく、より効率よくリソースを活用するための工夫です。

この節では、実際に「オブジェクト格納領域」や「参照」をどのように分けて格納し、どのように変更しているかについて、各コマンドの動きを見ながら解説します。

git init と管理情報

まずは、git init コマンドを使ってリポジトリ構築の初期に生成される Git の管理情報を見ていきます。local というディレクトリを作り、そこにリポジトリを構築します。

リポジトリの作成

```
$ mkdir local
$ cd local

# リポジトリの作成
$ git init
Initialized empty Git repository in /workspace/git/local/.git/
```

Git の管理情報が格納されるディレクトリパスの設定を git rev-parse コマンドを利用し

て、確認してみます。

管理情報の格納先を確認

```
$ git rev-parse --git-dir
.git
```

リポジトリ管理情報の格納先パスには、デフォルトで .git がパスとして指定されています。リポジトリパスは、git init の --separate-git-dir オプションか、git コマンドの --git-dir オプション、環境変数 GIT_DIR などで設定可能です。本解説ではデフォルトの .git をリポジトリパスとしてそのまま利用しています。

local ディレクトリ

```
$ cd local
$ ls -a
.    ..    .git
```

local/.git ディレクトリの初期の構成は、以下のようになっています。

local/.git のディレクトリ初期構造

```
$ tree .git
.git
├── HEAD
├── branches
├── config
├── description
├── hooks
│   ├── applypatch-msg.sample
│   ├── commit-msg.sample
│   ├── post-update.sample
│   ├── pre-applypatch.sample
│   ├── pre-commit.sample
│   ├── pre-push.sample
│   ├── pre-rebase.sample
│   ├── prepare-commit-msg.sample
│   └── update.sample
├── info
│   └── exclude
├── objects
│   ├── info
│   └── pack
└── refs
    ├── heads
    └── tags
```

各管理情報の初期状態を、ざっと見ていきます。各ファイルの説明よりも、Git の各コ

マンドによって、管理情報がどう変更されいていくかを先に知りたい場合は、次項の「git add と管理情報」以降から読み始めてください。

HEAD ファイル

.git/HEAD には、現在のブランチを指すシンボリック参照（"symbolic reference"）を格納します。シンボリック参照は、ハッシュ値への直接参照ではなく、他を参照します。

.git/HEAD

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

初期状態では、参照である refs/head/master が参照されています。これが参照の参照です。.git/refs ディレクトリと master ファイルについては、「git commit と管理情報」にて後述しますが、最初のコミット時に作成される master ブランチを、ブランチ作成前から参照していることになります。この時点では、まだシンボリック参照が指している .git/refs/head/master という管理情報は存在していません。.git/HEAD は、ブランチを切り替えるたびに変更されます。後述の「git checkout と管理情報」で詳しく説明します。

config ファイル

.git/config は、リポジトリ固有の設定ファイルです。

.git/config

```
$ cat .git/config
[core]
    repositoryformatversion = 0
    filemode = true
    bare = false
    logallrefupdates = true
    ignorecase = true
    precomposeunicode = true
```

この節の解説は、Git がどのようにバージョン管理を行うかを見ていくことが目的ですので、細かい設定まで見る必要はありませんが、簡単に解説します。初期状態では、内部フォーマットのバージョンを指す repositoryformatversion、ワーキングディレクトリのファイルの実行ビットを優先するかどうかの filemode、bare リポジトリであるかを表す bare、.git/logs/ 以下に全ての参照の更新を記録するかどうかを表す logallrefupdates、ファイルシステム上の大文字小文字の区別を無視するかを表す ignorecase、MacOS 上でファイルシステムがファイル名を Unicode 変換するのを Git が元に戻し Windows とのファイル名の互換性を保つ動きをする precomposeunicode などの設定が行われます。細かく .git/config の設定内容を知りたい場合は、man コマンドを利用して、man git-config によって

詳細を確認できます。$GIT_COMMON_DIR 環境変数が設定されている場合は、このファイルの代わりに、$GIT_COMMON_DIR/config が参照されます。.git/config は、Git のコマンドによって自動変更が行われます。後述の「git remote と管理情報」と「git push と管理情報」において、変更される内容について詳しく説明します。

description ファイル

.git/description には、リポジトリ名が格納されています。

.git/description

```
$ cat .git/description
Unnamed repository; edit this file 'description' to name the repository.
```

.git/description は、Git に付属する GitWeb という CGI を利用して Git を使う際に、リポジトリ名を表示するために参照されます。初期状態では、上記のように Unnamed repository という状態でリポジトリ名がありません。.git/description を直接編集することでリポジトリ名を変更できます。GitHub のリポジトリ名は、.git/description は参照されず、GitHub 上でリポジトリ名を編集することになるので注意してください。

hooks ディレクトリ

.git/hooks ディレクトリは、以下のように *.sample ファイルを格納しています。

.git/hooks

```
$ ls .git/hooks
applypatch-msg.sample        pre-push.sample
commit-msg.sample          pre-rebase.sample
post-update.sample          prepare-commit-msg.sample
pre-applypatch.sample        update.sample
pre-commit.sample
```

.git/hooks の初期状態では、フックに利用できるサンプルスクリプトが格納されていますが、全て無効化されています。.sample の拡張子を取り除くことで有効化できます。Git コマンドによって、.git/hooks 以下に変更が行われることはありません。

exclude ファイル

.git/info/exclude には、除外ファイルのパターンを記載できます。

.git/info/exclude

```
$ cat .git/info/exclude
# git ls-files --others --exclude-from=.git/info/exclude
```

```
# Lines that start with '#' are comments.
# For a project mostly in C, the following would be a good set of
# exclude patterns (uncomment them if you want to use them):
# *.[oa]
# *~
```

初期状態では何の設定も入っておらずコメントだけ記載されています。除外ファイルのパターンといえば、.gitignore がありますが、.gitignore はディレクトリ単位の除外リストです。このファイルは、リポジトリに対しての除外ファイルを設定します。git status、git add、git rm、git clean コマンドが .git/info/exclude を参照します。

objects ディレクトリ

.git/objects ディレクトリは、リポジトリに関連付けられたオブジェクトを格納する格納領域（オブジェクトストア）です。3 種類のオブジェクトファイルとパックファイル（"packfile"）が格納されます。パックファイルについては、03「git gc と管理情報」にて詳細を解説します。objects/ ディレクトリが格納することになる「オブジェクト格納領域」は、以下の 3 つの基本オブジェクトから構成されます。各オブジェクトの詳細は、Git コマンドを触りながら、後述の「git add と管理情報」および「git commit と管理情報」にて後述します。ここでは概要だけ眺めて 3 種類あることだけを確認してください。

オブジェクト名	概要
blob（ブロブ）オブジェクト	ファイルのデータのみを格納します。ファイルのメタデータは格納しません。ファイルの内容が少しでも異なれば、新しい blob オブジェクトが作成されます。後述の「blob（ブロブ）オブジェクト」で、詳細を解説します。
tree（ツリー）オブジェクト	blob オブジェクトの ID へのリンク、1 階層分のディレクトリの情報とファイルのパス名を格納します。さらに、tree オブジェクトを参照することで階層構造を管理することができます。後述の「tree（ツリー）オブジェクト」で、詳細を解説します。
commit（コミット）オブジェクト	変更時のメタデータを格納します。 コミットの、日付、ログメッセージ、コミッターなどのメタデータと、ディレクトリ構造・ファイルの変更を関連付けるため、tree オブジェクトへの参照を格納します。後述の「commit（コミット）オブジェクト」で、詳細を解説します。

refs ディレクトリ

.git/refs ディレクトリに作成されるサブディレクトリ以下に、ローカルブランチ・リモート追跡ブランチなどの各ブランチの参照と、「tag（タグ）」オブジェクトを格納します。参照については、後述の、「リファレンス・参照（refs）」において、実例を用いて解説します。$GIT_COMMON_DIR 環境変数が設定されている場合は、このファイルの代わりに、$GIT_COMMON_DIR/refs 以下の「参照」が参照されます。

オブジェクト名	概要
tag（タグ）オブジェクト	オブジェクトを特定するシンボルの役割で、特定のコミットに付けた名前と、その名前が指す commit オブジェクトを格納します。後述の「tag（タグ）」オブジェクトで、詳細を解説します。

git init によるリポジトリ初期構築時には、いくつかのファイルとディレクトリが自動生成されます。これらの管理情報の中から、この節では主に、コマンド発行時のオブジェクト格納領域である objects/ ディレクトリと、参照格納領域である refs/ ディレクトリが変更される動きに注目していきます。以降の Git コマンドでは、初期状態からの管理情報の変更差分を抜き出して、実際の格納データを使って解説します。

git add と管理情報

新規ファイルを作成して、作成したファイルをステージし、git add コマンドによって管理情報がどのように追加・変更されるかを見ていきます。まずは、ワーキングディレクトリにファイルを追加して、ステージングします。

ファイルの作成とステージング

```
$ cd local
$ echo "good morning" > a.txt
$ ls -a
.    ..    .git    a.txt

# ステージング
$ git add a.txt
```

git add 実行後に、追加された管理情報は以下の通りです。

add コマンド時に追加された管理情報

```
.git
├── index
├── objects
│   ├── b1
│   │   └── eb87387a92aa01e2bd12ddf8a7fab28dda14e1
```

このような Git コマンドの実行時に、コマンドごとに異なる .git ディレクトリ以下の管理情報が、追加および更新されていきます。git add の後、index ファイルと、objects ディレクトリの b1 ディレクトリの下に、オブジェクトファイル eb87387a92aa01e2bd12ddf8a7fab28dda14e1 が追加されています。それぞれ、どのような管理情報を持つのか見ていきます。

index ファイル

git add を利用すると、指定したファイルはステージングされ、「追跡（tracked）」状態になります。この現在のインデックスの情報が .git/index に書かれています。.git/index はバイナリファイルなので、直接ファイルを閲覧しても、意味がわかりません。git status を使って現在のインデックスがどうなっているかを確認します。

index を確認

```
$ git status
On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

    new file:   a.txt
```

new file として a.txt がステージングされています。ステージングされたファイルを確認する場合には git ls-files --stage を使います。

ステージングされたファイル

```
$ git ls-files --stage
100644 b1eb87387a92aa01e2bd12ddf8a7fab28dda14e1 0    a.txt
```

a.txt の横に、いくつかの管理情報が表示されています。ここで、b1eb87387a92aa01e2bd12ddf8a7fab28dda14e1 という値に注目してください。

ハッシュ値

git ls-files --stage を用いて表示したステージの状態には、ステージされた a.txt の横に、b1eb87387a92aa01e2bd12ddf8a7fab28dda14e1 という値が表示されていますが、これをハッシュ値と呼びます。Git は、オブジェクトの内容からハッシュ関数 SHA-1 を利用して生成したハッシュ値を、オブジェクトの名前として付けます。生成されるハッシュ値は、40 桁の 16 進数となっており、objects/ 以下に生成されるオブジェクトは、a.txt における b1eb87387a92aa01e2bd12ddf8a7fab28dda14e1 のように、全てユニークなハッシュ値で名付けられ管理されています。

名前空間の活用

ステージされたファイルのデータ

```
├── objects
│   ├── b1
│   │   └── eb87387a92aa01e2bd12ddf8a7fab28dda14e1
```

さて、おかしなことに気付きませんか？　先ほど、git ls-files --stage で見ていたハッシュ値が名前として付けられているならば、b1eb87387a92aa01e2bd12ddf8a7fab28dda14e1 がオブジェクトのファイル名として存在していそうですが、よく見るとハッシュ値が分割され、先頭 2 文字の b1 がディレクトリ名に利用され、それ以降のハッシュ値 eb87

387a92aa01e2bd12ddf8a7fab28dda14e1 がファイル名として利用されています。これには理由があります。git は名前空間として先頭 2 文字を利用し、データ管理や検索の効率を向上させているのです。では、実際に、このオブジェクトの中身を見ていきます。

blob（ブロブ）オブジェクト

git add によって追加された objects ディレクトリの下の階層のオブジェクトファイルを見ていきます。オブジェクトファイルの中身をみるには、git cat-file コマンドを利用します。

b1eb87387a92aa01e2bd12ddf8a7fab28dda14e1 を確認

```
$ git cat-file -t b1eb87387a92aa01e2bd12ddf8a7fab28dda14e1
blob
$ git cat-file -p b1eb87387a92aa01e2bd12ddf8a7fab28dda14e1
good morning
```

git cat-file の -t オプションは、オブジェクトの種類を表示します。b1eb87387a92aa01e2bd12ddf8a7fab28dda14e1 は blob と表示され、blob オブジェクトであることがわかります。git cat-file の -p オプションは、ファイルの中身を表示します。blob オブジェクト b1eb87387a92aa01e2bd12ddf8a7fab28dda14e1 の中には、good morning という、a.txt に記載したデータの中身だけが格納されています。

blob オブジェクトでは、ファイルのメタデータとは切り離して、データの内容だけを保持します。blob オブジェクトに付けられているハッシュ値は、データ内容から計算されており、同一内容のデータは全て同じハッシュ値を持つことになります。good morning というデータを持つファイルを作成して、git add すると、常に上記の例と同じ b1eb87387a92aa01e2bd12ddf8a7fab28dda14e1 という blob オブジェクトが生成されるということになります。

実際に、別のリポジトリを作成し、z.txt というファイルに "good morning" というデータを作成して git add してみます。

z.txt として同じデータを作成し blob オブジェクトのハッシュ値を確認

```
# 新しいリポジトリの作成
$ mkdir git-hash-test
$ cd git-hash-test/
$ git init
Initialized empty Git repository in /workspace/git-hash-test/.git/

#z.txt の作成
$ echo "good morning" > z.txt
$ cat z.txt
good morning

# ステージング
$ git add z.txt
```

```
# ハッシュ値の確認
$ git ls-files --stage
100644 b1eb87387a92aa01e2bd12ddf8a7fab28dda14e1 0    z.txt
$ git cat-file -t b1eb87387a92aa01e2bd12ddf8a7fab28dda14e1
blob
$ git cat-file -p b1eb87387a92aa01e2bd12ddf8a7fab28dda14e1
good morning
```

z.txt においても、a.txt と全く同じ blob オブジェクト b1eb87387a92aa01e2bd12ddf8a7fab28dda14e1 が生成されました。以上のように、ファイル名が違っていても、データが同じであれば、同じハッシュ値が採番されます。この仕組みによって、Git のバージョン管理システムは、同じデータをバージョンごとに持つ必要がなくなり、効率の良いデータ格納を行うことができます。例えば、ファイル名が変更された場合、データの値は変わらないので、blob オブジェクト自体には何の変化もない、ということになります。

git commit と管理情報

コミット git commit によって、管理情報がどのように変化するかを見ていきます。先ほどの git add が完了した状態から、git commit でコミットを行います。

コミット

```
# コミット
$ git commit -m "first greeting"
[master (root-commit) 1162a51] first greeting
 1 file changed, 1 insertion(+)
 create mode 100644 a.txt
```

初回コミットによって、master ブランチが作成されています。

ブランチを確認

```
$ git branch
* master
```

続いて、初回コミットによって、追加・更新が行われたファイルを見ていきます。

コミットによって追加・更新が行われたファイル

```
.git
├── COMMIT_EDITMSG
├── index
├── logs
│   ├── HEAD
```

```
│   └── refs
│       └── heads
│           └── master
├── objects
│   ├── 11
│   │   └── 62a5173ee3a6b9dff9a1d0941a28e4656840f0
│   ├── 1a
│   │   └── 24d97271e9e771f2717927f9b7a0a158ecc1fe
│   ├── info
│   └── pack
└── refs
    ├── heads
    │   └── master
```


COMMIT_EDITMSG ファイル

COMMIT_EDITMSG は、直前のコミットメッセージが格納されています。

COMMIT_EDITMSG

```
$ cat .git/COMMIT_EDITMSG
first greeting
```


-m オプションを付けないコミット時や、amend でコミットが修正される場合に、このファイルを直接編集することになります。

リファレンス・参照（refs）

refs/ 以下は、「リファレンス」あるいは「参照」と呼ばれ、任意の Git オブジェクトを参照します。

.git/refs/heads/master

```
$ cat .git/refs/heads/master
1162a5173ee3a6b9dff9a1d0941a28e4656840f0
```

初回コミットによって、master ブランチが作成されましたが、master ブランチの参照先を、.git/refs/heads/master に格納しています。master ブランチの参照先を git rev-parse コマンドを利用して確認してみます。

master の参照先

```
$ git rev-parse master
1162a5173ee3a6b9dff9a1d0941a28e4656840f0
```

masterブランチは、最新のコミットとして、1162a5173ee3a6b9dff9a1d0941a28e4656840f0というオブジェクトを参照しています。このように、refs/以下には、特定のGitオブジェクトへの参照が格納されます。

logsディレクトリ

git commit後、.git/logsディレクトリが作成されました。.git/logsディレクトリは、参照への更新を記録します。.git/logs/HEAD`ファイルには、HEADの参照の変更履歴が格納されています。

.git/logs/HEAD

```
$ cat .git/logs/HEAD
0000000000000000000000000000000000000000 1162a5173ee3a6b9dff9a1d0941a28
e4656840f0 Seigo Kawamura <seigo_kawamura@example.com> 1438690167 +0900
commit (initial): first greeting
```

初回コミットであることから、直前のコミットは履歴として存在しません。0000000000000000000000000000000000000000は、直前のコミットがない状態を表していて、通常この位置には直前のコミットのハッシュ値が記録されます。1162a5173ee3a6b9dff9a1d0941a28e4656840f0が今回のハッシュ値です。これと同じ情報は、git reflogで見ることができます。

reflogを確認

```
$ git reflog
1162a51 HEAD@{0}: commit (initial): first greeting
```

短縮したハッシュ値である1162a51は、1162a5173ee3a6b9dff9a1d0941a28e4656840f0の先頭部分であることがわかります。

では、参照.git/refs/heads/masterが参照しているオブジェクト1162a5173ee3a6b9dff9a1d0941a28e4656840f0を探してみます。今回の追加・更新のあったファイルの一覧を見ると、objects/11/62a5173ee3a6b9dff9a1d0941a28e4656840f0という形で格納されているオブジェクトがあることがわかると思いますが、これが「commit（コミット）」オブジェクトです。

commit（コミット）オブジェクト

commitオブジェクト1162a5173ee3a6b9dff9a1d0941a28e4656840f0を見てみます。

1162a5173ee3a6b9dff9a1d0941a28e4656840f0を確認

```
$ git cat-file -t 1162a5173ee3a6b9dff9a1d0941a28e4656840f0
commit
```

```
$ git cat-file -p 1162a5173ee3a6b9dff9a1d0941a28e4656840f0
tree 1a24d97271e9e771f2717927f9b7a0a158ecc1fe
author Seigo Kawamura <seigo_kawamura@example.com> 1438690167 +0900
committer Seigo Kawamura <seigo_kawamura@example.com> 1438690167 +0900

first greeting
```

タイプを確認すると、commit と表示され、commit オブジェクトであることが確認できます。commit オブジェクトの内部には、コミットの author や時刻、コミットメッセージが格納されていることがわかります。コミット 1162a5173ee3a6b9dff9a1d0941a28e4656840f0 の中身を確認してみます。

コミット 1162a5173ee3a6b9dff9a1d0941a28e4656840f0 を確認

```
$ git show 1162a5173ee3a6b9dff9a1d0941a28e4656840f0
commit 1162a5173ee3a6b9dff9a1d0941a28e4656840f0
Author: Seigo Kawamura <seigo_kawamura@example.com>
Date:   Tue Aug 4 21:09:27 2015 +0900

    first greeting

diff --git a/a.txt b/a.txt
new file mode 100644
index 0000000..b1eb873
--- /dev/null
+++ b/a.txt
@@ -0,0 +1 @@
+good morning
```

このような、コミットのメタデータが commit オブジェクトとして記録されています。参照したコミット ID と、commit パラメータに記載された commit オブジェクトのハッシュ値が、同じであることに注目してください。コミットオブジェクトのハッシュ値は、そのままコミット ID として利用されます。

blob オブジェクトのときは、同一のデータであれば同一のハッシュ値が名付けられていましたが、commit オブジェクトのハッシュ値はどうでしょうか？　同一内容のコミットは、同一のコミット ID（ハッシュ値）となるでしょうか？　blob オブジェクトと、同じような実験をしてみます。新しいリポジトリを作成し、全く同じデータを持つ a.txt を作成して、全く同じコミットメッセージでコミットしたらどうなるでしょうか？

同じデータを持つ a.txt を同じコミットメッセージでコミット

```
# 新しいリポジトリの作成
$ mkdir commit-hash-test
$ cd commit-hash-test/
$ git init
Initialized empty Git repository in /workspace/commit-hash-test/.git/
```

```
#a.txtの作成とコミット
$ echo "good morning" > a.txt
$ ls -a
.    ..    .git    a.txt
$ cat a.txt
good morning
$ git add a.txt
$ git commit -m "first greeting"
[master (root-commit) 2480ba9] first greeting
 1 file changed, 1 insertion(+)
 create mode 100644 a.txt

#ハッシュ値の確認
$ git rev-parse master
2480ba9fa3530eb7be7ec27912dc94e0ca05214c

#コミット内容の確認
$ git show 2480ba9fa3530eb7be7ec27912dc94e0ca05214c
commit 2480ba9fa3530eb7be7ec27912dc94e0ca05214c
Author: Seigo Kawamura <seigo_kawamura@example.com>
Date:   Mon Aug 10 17:22:56 2015 +0900

    first greeting

diff --git a/a.txt b/a.txt
new file mode 100644
index 0000000..b1eb873
--- /dev/null
+++ b/a.txt
@@ -0,0 +1 @@
+good morning
```

この実験のコミットでは、2480ba9fa3530eb7be7ec27912dc94e0ca05214c がコミットIDとして割り当てられました。先ほどの、commitオブジェクト1162a5173ee3a6b9dff9a1d0941a28e4656840f0と同じファイル・同じコミットメッセージでコミットしましたが、commitオブジェクトのコミットIDは、異なることがわかります。コミットオブジェクトはauthorとコミット時刻を含むことからもわかる通り、commitオブジェクトのハッシュ値であるコミットIDは、任意の全てのリポジトリで一意です。開発者同士、どのリポジトリの開発であろうとも、コミットIDを伝えることで、どのコミットについての話題かを厳密に特定することができます。

少し話を戻して、commitオブジェクトの中身をもう一度見ていきます。

commitオブジェクト1162a5173ee3a6b9dff9a1d0941a28e4656840f0

```
$ git cat-file -p 1162a5173ee3a6b9dff9a1d0941a28e4656840f0
tree 1a24d97271e9e771f2717927f9b7a0a158ecc1fe
author Seigo Kawamura <seigo_kawamura@example.com> 1438690167 +0900
committer Seigo Kawamura <seigo_kawamura@example.com> 1438690167 +0900
```

```
first greeting
```

コミットで変更のあったファイルの情報 a.txt についての記述が見つかりません。その代わりに、tree というパラメータと、ハッシュ値が格納されています。これは、ディレクトリ情報を管理する tree（ツリー）オブジェクトへの参照です。このような、commit オブジェクトは、コミットのメタデータを持ち、ディレクトリ・ファイルの変更については、tree オブジェクトへの参照を持ちます。

tree（ツリー）オブジェクト

今回のコミットで objects/ 以下に追加された、tree オブジェクトの中身を見ていきます。

1a24d97271e9e771f2717927f9b7a0a158ecc1fe を確認

```
$ git cat-file -t 1a24d97271e9e771f2717927f9b7a0a158ecc1fe
tree
$ git cat-file -p 1a24d97271e9e771f2717927f9b7a0a158ecc1fe
100644 blob b1eb87387a92aa01e2bd12ddf8a7fab28dda14e1    a.txt
```

タイプを確認すると、tree と表示されています。tree オブジェクトの中身には、blob オブジェクトへの参照が格納されていることが確認できます。このような、tree オブジェクトは 1 つ以上の blob オブジェクトと tree オブジェクトを持ち、それぞれのファイル・ディレクトリの形式とパーミッションを管理しています。ここには、モード（mode）・オブジェクトタイプ・ファイル名が書かれています。blob の横に、100644 という数列が確認できます。これがモードを表しています。見た目通りの数値ではなく、bit で管理されています。先頭 4bit は、オブジェクトのタイプ、続く 3bit は利用しておらず、残り 9bit は UNIX のパーミッションを表しています。例えば、以下のようなモードがあります。

モード（mode）	ビット	意味
100644	1000 000 110100100	実行できないファイル
100755	1000 000 111101101	実行可能なファイル
040000	0100 000 000000000	ディレクトリ
120000	1010 000 000000000	シンボリックリンク
160000	1110 000 000000000	Gitlink（git submodule で利用）

Git のバージョン管理の仕組みを知る・初級編では、git init から git commit までの管理情報の変更を追いかけて、内部の作りを見てきました。基本的なオブジェクトを学び、Git がどのように動いているのかがだいぶブラックボックスではなくなってきたと思います。Git のバージョン管理の仕組みを知る・中級編では、チーム開発で必要となるコマンドの管理情報がどのように変更されるのかを見ていきます。

02 Git のバージョン管理の仕組みを知る ～中級編～

中級編では、リモートリポジトリとローカルリポジトリの間の管理と、マージやリベースなどの動きと共に管理情報がどのように変更されるか、を見ていきます。

git init --bare と管理情報

01「Git のバージョン管理の仕組みを知る　～初級編～」で操作していたリポジトリを継続して操作し、実際の動きを追っていきます。

複数人で開発するときに、全員が参照するリポジトリ上のワーキングディレクトリで、特定の個人が開発を行い、ファイルの編集やブランチの切り替えができるとしたら、どうでしょう？　もちろん、それでも運用は可能です。しかし、そこでブランチを切り替えるためにチェックアウトを繰り返ししたらどうでしょうか？　作業中のリポジトリに、プッシュが行われたら？　思いもよらないデータの不整合が起こる可能性があります。

bare リポジトリはワーキングディレクトリを持ちません。git init --bare を利用することで、全員が参照しているリポジトリで、特定の個人が直接開発を行わないようにすることができます。実際に、git init と git init --bare では、管理情報がどのように異なるのか見ていきます。

local ディレクトリの横に、remote_test.git ディレクトリを作成し、そこに bare リポジトリを作成してみます。なぜ remote_test ディレクトリではなく、.git を末尾に付加した意図は後でわかります。

remote_test.git の作成と bare リポジトリの作成

```
$ mkdir remote_test.git
$ ls
local        remote_test.git
$ cd remote_test.git/

#bare リポジトリの作成
$ git init --bare
Initialized empty Git repository in /workspace/git/remote_test.git/
```

remote_test.git のリポジトリが、bare リポジトリになっているか、確認してみます。

remote_test.git は bare リポジトリか？

```
$ git rev-parse --is-bare-repository
true
```

bare リポジトリと、通常の init が作る管理情報の構成とが、どのように異なるかを見てみます。remote_test.git ディレクトリのファイル構成は、以下のようになっています。

remote_test.git のファイル構成

```
$ cd remote_test.git
$ ls -a
.        branches    hooks        refs
..        config        info
HEAD        description    objects
```

比較のため、git init したばかりの local ディレクトリと比較してみます。

local のファイル構成

```
$ cd local
$ ls -a
.    ..    .git
```

git init による初期化状態で、local ディレクトリの場合は、.git ディレクトリのみが存在していますが、bare リポジトリ remote_test.git の場合は、refs を始めとした複数の管理情報がルートディレクトリ直下（ここでは remote_test.git/）にあることがわかります。実際のディレクトリ構成を並べて、比較してみます .

remote_test.git/ のディレクトリ構成は、以下のようになっています。

remote_test.git のディレクトリ構造

```
$ tree remote_test.git
remote_test.git
├── HEAD
├── branches
├── config
├── description
├── hooks
│   ├── #（省略）
├── info
│   └── exclude
├── objects
│   ├── info
│   └── pack
└── refs
    ├── heads
    └── tags
```

local/ のディレクトリ構成は以下のようになっています。

local のディレクトリ構造

```
$ tree local/.git
local/.git
```

```
├── HEAD
├── branches
├── config
├── description
├── hooks
│   ├── #（省略）
├── info
│   └── exclude
├── objects
│   ├── info
│   └── pack
└── refs
    ├── heads
    └── tags
```

local ディレクトリに git init で作成したばかりの管理情報は git-dir（この場合は .git）の下に作成されています。一方で、remote_test.git で作成した bare リポジトリでは、ルートディレクトリ直下に管理情報ファイルが並んでいます。上下で、見比べるとわかりますが、配置されている管理情報ファイルの種類には、差はありません。

差がないことを確認

```
# ディレクトリ構造をファイルに出力
$ tree local/.git > local_tree
$ tree remote_test.git > remote_tree

# 差を確認
$ diff local_tree remote_tree
1c1
< local/.git
---
> remote_test.git
```

diff コマンドの結果から、一行目のディレクトリ名の以外には、差がないことがわかります。つまり、local における .git と、remote_test.git が位置づけとしては同じ管理フォルダとなるため、remote_test に .git を付加した名前を付けています。また、初期状態の比較だけでは気付きませんが、bare リポジトリはワーキングディレクトリを持たないため、作業インデックスを格納する .git/index が通常作成されません。

git remote と管理情報

初級編の操作は、ローカルリポジトリに閉じていました。既存プロジェクトに、リモートを設定して、ローカルリポジトリとリモートの連携時に、管理情報がどう変更されるのかを見ていきます。これまでの手順では、git clone ではなく、git init によってリポジトリを作成しているので、リモートの設定が行われていません。リモートとの連携を試すには、

リモートの設定が必要となります。git remote を用いて、前の節で設定した remote_test.git を利用しながら、リモートを設定し、管理情報の変更を見ていきます。まずは、local ディレクトリに移動して、git remote で、リモートリポジトリとして remote_test を設定します。

remote の追加

```
$ cd local

# リモートの追加
$ git remote add origin ../remote_test
```

remote_test.git をリモートに指定するときは、remote_test として指定します。

remote コマンドによる管理情報の差分

```
$ cd local
$ tree .git
.git
├── config
```

config ファイルにだけ変更が行われました。config ファイルには、[remote "origin"] という設定が追加されています。リモートに関する設定を記載します。

.git/config の差分

```
$ cat .git/config
(略)
[remote "origin"]
    url = ../remote_test
    fetch = +refs/heads/*:refs/remotes/origin/*
```

url は、git remote で指定した ../remote_test を格納しています。fetch = +refs/heads/*:refs/remotes/origin/* の部分は、「転送元：転送先」の形式で記述されています。fetch の際は、remote の .git/refs/heads/ 以下のブランチを、ローカルの同名の追跡ブランチ .git/refs/remotes/origin/ 以下に置くことを表しています。

git push と管理情報

設定したリモートの origin への git push コマンドで、ローカルのリポジトリの管理情報がどのように変更されるかを見ていきます。git push の --set-upstream オプションで、追跡ブランチの設定を行います。

upstream の設定

```
$ cd local
```

```
#プッシュ
$ git push --set-upstream origin master
Counting objects: 3, done.
Writing objects: 100% (3/3), 233 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To ../remote_test
 * [new branch]      master -> master
Branch master set up to track remote branch master from origin.
```

master ブランチの upstream として、リモートの origin における master ブランチを設定しました。ローカルのリポジトリの管理情報の変更差分を見てみます。

local の tree の変更差分

```
$ cd local
#ローカルリポジトリの管理情報の差分
$ tree .git
.git
├── config
├── logs
│   └── refs
│       └── remotes
│           └── origin
│               └── master
└── refs
    ├── remotes
    │   └── origin
    │       └── master
```

参照 refs に、remotes/origin ディレクトリが作成されました。ここに remote である origin の master 追跡ブランチの現在の参照、.git/refs/remotes/origin/master が格納されています。

.git/refs/remotes/origin/master

```
$ cat .git/refs/remotes/origin/master
1162a5173ee3a6b9dff9a1d0941a28e4656840f0
```

ハッシュ値への参照が格納されています。参照されている 1162a5173ee3a6b9dff9a1d0941a28e4656840f0 は、以下の通り最初のコミット "first greeting" です。

1162a5173ee3a6b9dff9a1d0941a28e4656840f0 を確認

```
$ git cat-file -p 1162a5173ee3a6b9dff9a1d0941a28e4656840f0
tree 1a24d97271e9e771f2717927f9b7a0a158ecc1fe
author Seigo Kawamura <seigo_kawamura@example.com> 1438690167 +0900
committer Seigo Kawamura <seigo_kawamura@example.com> 1438690167 +0900
```

```
first greeting
```

追跡ブランチ master が、local リポジトリの参照している commit オブジェクト 1162a5173ee3a6b9dff9a1d0941a28e4656840f0 と、同じ commit オブジェクトを参照していることがわかります。

git push による master ブランチへの set-upstream によって、config にも、変更が掛かりました。.git/config の変更差分を見てみます。

.git/config

```
$ cd local
$ cat .git/config
(省略)
[branch "master"]
    remote = origin
    merge = refs/heads/master
```

[branch "master"] というブランチの設定が追加されています。remote という設定項目が追加され、git push では、リモートである origin の情報を更新する設定がされました。同様に、merge の設定項目が追加されました。git pull コマンド時に、origin からフェッチ（git fetch）して、refs/heads/master をローカルの master ブランチにマージ（git merge）するという設定を意味しています。

プッシュされたリモート側の管理情報は、どのように変更されたでしょうか？ remote_test.git ディレクトリの管理情報の変更差分を見ていきます。

リモートの tree の変更差分

```
$ cd remote_test.git
# リモートリポジトリの管理情報の差分
$ tree .
.
├── objects
│   ├── 11
│   │   └── 62a5173ee3a6b9dff9a1d0941a28e4656840f0
│   ├── 1a
│   │   └── 24d97271e9e771f2717927f9b7a0a158ecc1fe
│   ├── b1
│   │   └── eb87387a92aa01e2bd12ddf8a7fab28dda14e1
└── refs
    ├── heads
    │   └── master
```

リモートにオブジェクトが 3 つ作成されました。local リポジトリで git commit したときに追加された 3 つのオブジェクトが、リモート remote_test リポジトリにもそのまま作成されています。

リモートのオブジェクトファイルを確認

```
$ cd remote_test.git

#commit オブジェクト
$ git cat-file -t 1162a5173ee3a6b9dff9a1d0941a28e4656840f0
commit
$ git cat-file -p 1162a5173ee3a6b9dff9a1d0941a28e4656840f0
tree 1a24d97271e9e771f2717927f9b7a0a158ecc1fe
author Seigo Kawamura <seigo_kawamura@example.com> 1438690167 +0900
committer Seigo Kawamura <seigo_kawamura@example.com> 1438690167 +0900

first greeting

#tree オブジェクト
$ git cat-file -t 1a24d97271e9e771f2717927f9b7a0a158ecc1fe
tree
$ git cat-file -p 1a24d97271e9e771f2717927f9b7a0a158ecc1fe
100644 blob b1eb87387a92aa01e2bd12ddf8a7fab28dda14e1    a.txt

#blob オブジェクト
$ git cat-file -t b1eb87387a92aa01e2bd12ddf8a7fab28dda14e1
blob
$ git cat-file -p b1eb87387a92aa01e2bd12ddf8a7fab28dda14e1
good morning
```

作成された管理情報３つのオブジェクトは、local リポジトリと同じハッシュ値の、commit オブジェクト、commit オブジェクトから参照される tree オブジェクト、tree オブジェクトから参照される blob オブジェクトでした。ローカルリポジトリ local と、リモートリポジトリ remote_test は、git push 後、全く同じオブジェクトファイルを管理していることがわかりました。リモートリポジトリの master ブランチの参照はどうなっているでしょうか？

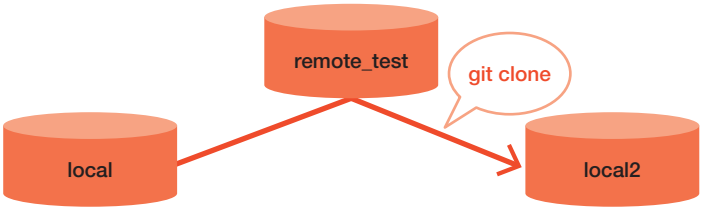
リモートの master ブランチの参照

```
$ cd remote_test.git
$ cat refs/heads/master
1162a5173ee3a6b9dff9a1d0941a28e4656840f0
```

リモートリポジトリの master ブランチは、commit オブジェクト 1162a5173ee3a6b9dff9a1d0941a28e4656840f0("first greeting") を指しています。git push 後に、ローカルリポジトリと、リモートリポジトリの参照が一致したことがわかります。

git clone と管理情報

local とは異なるディレクトリに、リモートリポジトリからクローンを行い、構成情報がどのように渡されるのかを見てみます。git clone を利用して、remote_test リポジトリから、local2 ディレクトリを作成し、管理情報を確認します。



remote_test リポジトリからのクローン

```
#remote_test リポジトリからのクローン
$ git clone ../remote_test local2
Cloning into 'local2'...
done.
$ cd local2
$ ls -a
.    ..    .git    a.txt
```

local2 ディレクトリに、remote_test からのクローンが完了しました。

local2 の tree 構造

```
$ cd local2
$ tree .git
.git
├── HEAD
#（省略）
├── config
├── objects
│   ├── 11
│   │   └── 62a5173ee3a6b9dff9a1d0941a28e4656840f0
│   ├── 1a
│   │   └── 24d97271e9e771f2717927f9b7a0a158ecc1fe
│   ├── b1
│   │   └── eb87387a92aa01e2bd12ddf8a7fab28dda14e1
#（省略）
```

objects以下の３つのオブジェクトファイルの管理情報は、リモートと同じハッシュ値で、commit オブジェクト・tree オブジェクト・blob オブジェクトが、そっくりそのまま伝わってきています。Git は分散管理システムであり、このようなリモートとローカルで同じ管理情報を所有します。.git/config の中身を確認します。

.git/config

```
$ cd local2
$ cat .git/config
#（省略）
[remote "origin"]
    url = /workspace/git/remote_test.git
```

```
    fetch = +refs/heads/*:refs/remotes/origin/*
[branch "master"]
    remote = origin
    merge = refs/heads/master
```

クローンで作成された .git/config は、すでにリモートの設定と、追跡ブランチの設定が作成されていて、git push や git pull が、引数なしでそのまま利用できるようになっています。

git tag と管理情報

再び local ディレクトリに戻って、最初のコミットにタグで名前を付け、どのように管理されているかを見ていきます。git tag コマンドを使って、最新のコミットに version1.0 というタグを付けてみます。

タグで version1.0 という名前を付ける

```
$ cd local

# タグの作成
$ git tag -a version1.0 -m 'first version'

# タグを確認
$ git tag
version1.0
```

git tag によって version1.0 タグを付けた際に、変更された差分ファイルは以下の通りです。

タグ付け後に変更された管理ファイル

```
$ tree .git
.git
├── objects
│   ├── 2f
│   │   └── be3fe9c638172f9bdf4a264e4698d69de2715e
│
└── refs
    └── tags
        └── version1.0
```

.git/refs/tags/version1.0 という参照と、2fbe3fe9c638172f9bdf4a264e4698d69de2715e として、オブジェクトが作成されています。

.git/refs/tags/version1.0

```
$ cat .git/refs/tags/version1.0
```

```
2fbe3fe9c638172f9bdf4a264e4698d69de2715e
```

参照が指しているのは、作成されたオブジェクト`2fbe3fe9c638172f9bdf4a264e4698d69de2715e です。

tag（タグ）オブジェクト

参照されているオブジェクト 2fbe3fe9c638172f9bdf4a264e4698d69de2715e のタイプと、中身を確認します。

2fbe3fe9c638172f9bdf4a264e4698d69de2715e を確認

```
$ git cat-file -t 2fbe3fe9c638172f9bdf4a264e4698d69de2715e
tag
$ git cat-file -p 2fbe3fe9c638172f9bdf4a264e4698d69de2715e
object 1162a5173ee3a6b9dff9a1d0941a28e4656840f0
type commit
tag version1.0
tagger Seigo Kawamura <seigo_kawamura@example.com> 1438770196 +0900

first version
```

2fbe3fe9c638172f9bdf4a264e4698d69de2715e のオブジェクトタイプは、tag となっています。これは、tag（タグ）オブジェクトです。ファイルの中身を見ると、タグの名称が、tag version1.0 として格納されており、commit オブジェクトとして object 1162a5173ee3a6b9dff9a1d0941a28e4656840f0("first greeting") を参照しています。

commit オブジェクト 1162a5173ee3a6b9dff9a1d0941a28e4656840f0

```
$ git cat-file -t 1162a5173ee3a6b9dff9a1d0941a28e4656840f0
commit
$ git cat-file -p 1162a5173ee3a6b9dff9a1d0941a28e4656840f0
tree 1a24d97271e9e771f2717927f9b7a0a158ecc1fe
author Seigo Kawamura <seigo_kawamura@example.com> 1438690167 +0900
committer Seigo Kawamura <seigo_kawamura@example.com> 1438690167 +0900

first greeting
```

タグを付けた commit オブジェクトには、変更がありません。tag オブジェクトにタグの詳細が格納され、commit オブジェクトへの参照を持つことで、タグとコミットを関連付けています。このことから、データ管理上、タグの管理と、commit の管理は疎結合であることがわかります。念のため、最新のコミットに付けたタグが、問題なく付加されているかどうか確認します。

タグを確認

```
$ git log --decorate=full
commit 1162a5173ee3a6b9dff9a1d0941a28e4656840f0 (HEAD -> refs/heads/
master, tag: refs/tags/version1.0, refs/remotes/origin/master)
Author: Seigo Kawamura <seigo_kawamura@example.com>
Date:   Tue Aug 4 21:09:27 2015 +0900

    first greeting
```

問題なく、tag: refs/tags/version1.0 と、1162a5173ee3a6b9dff9a1d0941a28e4656840f0 が関連付けられています。

git branch と管理情報

ブランチの作成はどのように管理されているか、git branch コマンドを使った後の管理情報を見ていきます。ここでは、チェックアウトはせずに、develop ブランチを作成して、管理情報への影響を絞っておきます。

develop ブランチの作成

```
$ cd local

# ブランチの作成
$ git branch develop
# ブランチの確認
$ git branch
  develop
* master
```

develop ブランチが作成だけが行われ、まだ master ブランチにいる状態です。

ブランチ作成後の管理情報の差分

```
$ tree .git
.git
├── logs
│   └── refs
│       ├── heads
│       │   └── develop
│
└── refs
    ├── heads
    │   ├── develop
    │
```

参照の履歴を格納する logs ディレクトリと、参照を格納する refs ディレクトリに、

develop という、ブランチ名と同じ名前のファイルが作成されています。ログから見ていきます。

.git/logs/refs/heads/develop

```
$ cat .git/logs/refs/heads/develop
0000000000000000000000000000000000000000 1162a5173ee3a6b9dff9a1d0941a28
e4656840f0 Seigo Kawamura <seigo_kawamura@example.com> 1438778966 +0900
branch: Created from master
```

branch: Created from master とあるように、develop ブランチを作成した起点が書かれています。

reflog で develop ブランチの履歴を確認

```
$ git reflog develop
1162a51 develop@{0}: branch: Created from master
```

develop ブランチのログを見ると、logs に格納されていたメッセージが表示されています。次に参照を確認します。

.git/refs/heads/develop

```
$ cat .git/refs/heads/develop
1162a5173ee3a6b9dff9a1d0941a28e4656840f0
```

develop ブランチは、commit オブジェクト 1162a5173ee3a6b9dff9a1d0941a28e4656840f0("first greeting") を参照しています。refs/heads/develop からコミットへの参照を、コミットログで確認してみます。

コミットログ

```
$ git log --decorate=full
commit 1162a5173ee3a6b9dff9a1d0941a28e4656840f0 (HEAD -> refs/heads/
master, tag: refs/tags/version1.0, refs/remotes/origin/master, refs/
heads/develop)
Author: Seigo Kawamura <seigo_kawamura@example.com>
Date:   Tue Aug 4 21:09:27 2015 +0900

    first greeting
```

コミット 1162a5173ee3a6b9dff9a1d0941a28e4656840f0 は、refs/heads/develop から参照されていることがわかります。このように、ブランチの正体は、コミットオブジェクトのハッシュ値、コミット ID への参照であることがわかります。

git checkout と管理情報

作成した develop ブランチをチェックアウトして、管理情報の変更を見てみます。

develop ブランチをチェックアウト

```
$ cd local

#develop ブランチのチェックアウト
$ git checkout develop
Switched to branch 'develop'

# ブランチの確認
$ git branch
* develop
  master
```

git checkout によるチェックアウト後の管理情報の変更差分は、以下の通りです。

チェックアウト後の管理情報の変更差分

```
$ tree .git
.git
├── HEAD
├── index
├── logs
│   └── HEAD
```

.git/HEAD ファイルが更新されています。HEAD ファイルは、現在のブランチの最新コミットへのシンボリック参照（"symbolic reference"）を格納しています。どんな管理情報が格納されているでしょうか。

.git/HEAD

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

develop ブランチへの参照、refs/heads/develop の値が、シンボリック参照として格納されています。シンボリック参照は、ハッシュ値への直接参照ではなく、他を参照します。refs/heads/develop を指していることから、現在のワーキングディレクトリが、develop ブランチを参照していることがわかります。ブランチの変更情報を参照できる git reflog の出力を確認します。

reflog の出力

```
$ git reflog
```

```
1162a51 HEAD@{0}: checkout: moving from master to develop
1162a51 HEAD@{1}: commit (initial): first greeting
```

これらの情報が、管理されている .git/logs/HEAD への更新を見てみます。

.git/logs/HEAD

```
$ cat .git/logs/HEAD
0000000000000000000000000000000000000000 1162a5173ee3a6b9dff9a1d0941a28
e4656840f0 Seigo Kawamura <seigo_kawamura@example.com> 1438690167 +0900
commit (initial): first greeting
1162a5173ee3a6b9dff9a1d0941a28e4656840f0 1162a5173ee3a6b9dff9a1d0941a28
e4656840f0 Seigo Kawamura <seigo_kawamura@example.com> 1438779507 +0900
checkout: moving from master to develop
```

develop ブランチをチェックアウトしたことが、checkout: moving from master to develop として、記録されています。

チェックアウトでは、ワーキングディレクトリから見ると、ファイルの変更などが行われるため複雑な印象を持ちますが、実際に必要なのは、.git/HEAD のシンボリック参照の変更と、インデックスとログの変更を行うだけの非常にシンプルな管理方法となっています。

git mv と管理情報

git mv でファイル移動をした場合は、どのように管理されるのかを見ていきます。以降の作業は、先ほどチェックアウトした、develop ブランチで行っていきます。

ファイルの移動

```
$ git branch
* develop
  master
$ ls -a
.    ..    .git    a.txt

# ファイルの移動
$ git mv a.txt b.txt
$ ls -a
.    ..    .git    b.txt
```

git mv 実施後、ワーキングディレクトリの a.txt は、b.txt に移動しています。管理情報ファイルで、変更された管理情報は以下です。

ファイル移動後の管理情報の差分

```
$ tree .git
.git
```

```
├── index
```

ファイルをステージングしたので、index が変更されています。git mv によって、ステージングされた内容は以下です。

ステージングを確認

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

    renamed:    a.txt -> b.txt
```

ステージングされた内容をコミットします。

コミット

```
$ git commit -m "move from a.txt to b.txt"
[develop be1eb7e] move from a.txt to b.txt
 1 file changed, 0 insertions(+), 0 deletions(-)
 rename a.txt => b.txt (100%)
```

コミット後の管理情報の差分は以下の通りです。

コミット後の管理情報の差分

```
$ tree .git
.git
├── COMMIT_EDITMSG
├── index
├── logs
│   ├── HEAD
│   └── refs
│       ├── heads
│       │   ├── develop
│
├── objects
│   ├── 0c
│   │   └── 1ce4b2d1da852658cb88f686cd8d43c90df5e4
│   ├── be
│   │   └── 1eb7e8911a20d7c616d20057928d72965f23a1
│
└── refs
    ├── heads
    │   ├── develop
```

objects ディレクトリ以下に、オブジェクトファイルが 2 つ増えています。git mv は、ファ

イルの移動を行いますが、ファイルのデータ自体は変更しません。データ変更が行われない場合の管理情報は、どのように変化するのでしょうか。be1eb7e8911a20d7c616d20057928d72965f23a1 から見ていきます。

データに変更がないコミット時の管理情報の変化

be1eb7e8911a20d7c616d20057928d72965f23a1 を確認

```
$ git cat-file -t be1eb7e8911a20d7c616d20057928d72965f23a1
commit
$ git cat-file -p be1eb7e8911a20d7c616d20057928d72965f23a1
tree 0c1ce4b2d1da852658cb88f686cd8d43c90df5e4
parent 1162a5173ee3a6b9dff9a1d0941a28e4656840f0
author Seigo Kawamura <seigo_kawamura@example.com> 1438780643 +0900
committer Seigo Kawamura <seigo_kawamura@example.com> 1438780643 +0900

move from a.txt to b.txt
```

commit オブジェクト be1eb7e8911a20d7c616d20057928d72965f23a1 は、tree オブジェクト 0c1ce4b2d1da852658cb88f686cd8d43c90df5e4 を指しています。これは、今回作成されたもう 1 つのオブジェクトです。

0c1ce4b2d1da852658cb88f686cd8d43c90df5e4 を確認

```
$ git cat-file -t 0c1ce4b2d1da852658cb88f686cd8d43c90df5e4
tree
$ git cat-file -p 0c1ce4b2d1da852658cb88f686cd8d43c90df5e4
100644 blob b1eb87387a92aa01e2bd12ddf8a7fab28dda14e1    b.txt
```

以上のように、今回の git mv 後のコミットで更新があったのは、commit オブジェクトと、tree オブジェクトだけでした。データ実体を指す、blob オブジェクトの更新はないのでしょうか？　tree オブジェクト 1ce4b2d1da852658cb88f686cd8d43c90df5e4 は、b.txt という名前で、blob オブジェクト b1eb87387a92aa01e2bd12ddf8a7fab28dda14e1 を指しています。この b1eb87387a92aa01e2bd12ddf8a7fab28dda14e1 は、最初の a.txt のコミット時に作成されたオブジェクトです。

b1eb87387a92aa01e2bd12ddf8a7fab28dda14e1 を確認

```
$ git cat-file -t b1eb87387a92aa01e2bd12ddf8a7fab28dda14e1
blob
$ git cat-file -p b1eb87387a92aa01e2bd12ddf8a7fab28dda14e1
good morning
```

データの中身 good morning には変更がないのでハッシュ値に変更はなく、blob オブジェクト b1eb87387a92aa01e2bd12ddf8a7fab28dda14e1 には変化はありませんでした。

tree オブジェクト 1ce4b2d1da852658cb88f686cd8d43c90df5e4 の作成だけで、ファイルが a.txt から b.txt に移動できたことになります。

commit（コミット）オブジェクトにおける parent

もう一度、今回追加した commit オブジェクト be1eb7e8911a20d7c616d20057928d72965f23a1 を見てみます。

be1eb7e8911a20d7c616d20057928d72965f23a1 を確認

```
$ git cat-file -t be1eb7e8911a20d7c616d20057928d72965f23a1
commit
$ git cat-file -p be1eb7e8911a20d7c616d20057928d72965f23a1
tree 0c1ce4b2d1da852658cb88f686cd8d43c90df5e4
parent 1162a5173ee3a6b9dff9a1d0941a28e4656840f0
author Seigo Kawamura <seigo_kawamura@example.com> 1438780643 +0900
committer Seigo Kawamura <seigo_kawamura@example.com> 1438780643 +0900

move from a.txt to b.txt
```

今回のコミットは、2 回目のコミットになります。2 回目以降のコミットで生成される commit オブジェクトには、parent という情報が付加されています。これは何を表しているでしょうか？　parent 1162a5173ee3a6b9dff9a1d0941a28e4656840f0 と表示されているように、parent は、オブジェクト 1162a5173ee3a6b9dff9a1d0941a28e4656840f0 を指しています。

1162a5173ee3a6b9dff9a1d0941a28e4656840f0 を確認

```
$ git cat-file -t 1162a5173ee3a6b9dff9a1d0941a28e4656840f0
commit
$ git cat-file -p 1162a5173ee3a6b9dff9a1d0941a28e4656840f0
tree 1a24d97271e9e771f2717927f9b7a0a158ecc1fe
author Seigo Kawamura <seigo_kawamura@example.com> 1438690167 +0900
committer Seigo Kawamura <seigo_kawamura@example.com> 1438690167 +0900

first greeting
```

parent が指し示している 1162a5173ee3a6b9dff9a1d0941a28e4656840f0 は、最初のコミットを管理する commit オブジェクトです。commit オブジェクトは、この parent による commit オブジェクトの参照を元にコミット履歴をたどります。そして、最初のコミット 1162a5173ee3a6b9dff9a1d0941a28e4656840f0 には、parent がありません。

コミットログを見て、parent による参照を、おさらいします。

コミットログ

```
$ git log --graph --pretty=format:"%H %s"
```

```
* be1eb7e8911a20d7c616d20057928d72965f23a1 move from a.txt to b.txt
* 1162a5173ee3a6b9dff9a1d0941a28e4656840f0 first greeting
```

1162a5173ee3a6b9dff9a1d0941a28e4656840f0("first greeting") が最初のコミット、次に、be1eb7e8911a20d7c616d20057928d72965f23a1("move from a.txt to b.txt") の順に並びました。

parent の向きに注目してください。2 番目のコミット be1eb7e8911a20d7c616d20057928d72965f23a1("move from a.txt to b.txt") から、1 番目のコミット 1162a5173ee3a6b9dff9a1d0941a28e4656840f0("first greeting") を指しています。このような、コミット履歴は parent のハッシュ値の参照によってたどることができます。

git commit（データを上書きする場合）と管理情報

b.txt に good afternoon という挨拶を追加してみます。

b.txt への上書き

```
$ git branch
* develop
  master
$ ls -a
.    ..    .git    b.txt

# ファイルへの文字列の追加
$ cat b.txt
good morning
$ echo "good afternoon" >> b.txt
$ cat b.txt
good morning
good afternoon

# 差分の確認
$ git diff
diff --git a/b.txt b/b.txt
index b1eb873..7367032 100644
--- a/b.txt
+++ b/b.txt
@@ -1 +1,2 @@
 good morning
+good afternoon

# 変更のコミット
$ git add b.txt
$ git commit -m "adding second greeting"
[develop 9a1df15] adding second greeting
 1 file changed, 1 insertion(+)
```

これまで、初回コミット、データの上書き変更のないコミットを見てきました。今回のコミットでは、b.txt に good afternoon という文字列が追記されました。データが上書きされた状態でのコミットでは、どのように管理情報の変更差分が変わるでしょうか？

local の .git の差分

```
$ tree .git
.git
├── COMMIT_EDITMSG
├── index
├── logs
│   ├── HEAD
│   └── refs
│       ├── heads
│       │   └── develop
│
├── objects
│   ├── 73
│   │   └── 670329a9741fd52674409adb06149e535988aa
│   ├── 9a
│   │   └── 1df1574f7aa38c6ab1d54c54432cde8a5b57d4
│   ├── a8
│   │   └── 8781bdbf18ee9ff43912ce0df57cbc17dfe70d
│
└── refs
    ├── heads
    │   └── develop
```

objects/ に、３つのオブジェクトが追加されました。これまで見てきた挙動から、commit オブジェクト・tree オブジェクト・blob オブジェクトが追加されていると予想できます。１つ１つ見てみます。まず、commit オブジェクトは以下のようになります。

9a1df1574f7aa38c6ab1d54c54432cde8a5b57d4 を確認

```
$ git cat-file -t 9a1df1574f7aa38c6ab1d54c54432cde8a5b57d4
commit
$ git cat-file -p 9a1df1574f7aa38c6ab1d54c54432cde8a5b57d4
tree a88781bdbf18ee9ff43912ce0df57cbc17dfe70d
parent be1eb7e8911a20d7c616d20057928d72965f23a1
author Seigo Kawamura <seigo_kawamura@example.com> 1438931924 +0900
committer Seigo Kawamura <seigo_kawamura@example.com> 1438931924 +0900

adding second greeting
```

コミットメッセージadding second greetingを始めとしたメタ情報が管理されています。commit オブジェクトから参照されている tree オブジェクトは、以下のようになります。

a88781bdbf18ee9ff43912ce0df57cbc17dfe70d を確認

```
$ git cat-file -t a88781bdbf18ee9ff43912ce0df57cbc17dfe70d
tree
$ git cat-file -p a88781bdbf18ee9ff43912ce0df57cbc17dfe70d
100644 blob 73670329a9741fd52674409adb06149e535988aa    b.txt
```

b.txt の上書きされた新しいデータ内容として、blob オブジェクト 73670329a9741fd52674409adb06149e535988aa が参照されています。blob オブジェクトは、以下の通りです。

73670329a9741fd52674409adb06149e535988aa を確認

```
$ git cat-file -t 73670329a9741fd52674409adb06149e535988aa
blob
$ git cat-file -p 73670329a9741fd52674409adb06149e535988aa
good morning
good afternoon
```

追加した、good afternoon という文字列を含むデータが、blob オブジェクト 73670329a9741fd52674409adb06149e535988aa に管理されています。変更差分ではなく、b.txt に上書きしたデータそのものが格納されていることがわかります。

Git はスナップショットでデータを管理する

b.txt の内容を上書き編集しましたが、直前のコミットが参照していた上書き前の b.txt の内容を保持していた blob オブジェクト b1eb87387a92aa01e2bd12ddf8a7fab28dda14e1 は、どうなっているのでしょうか？

b1eb87387a92aa01e2bd12ddf8a7fab28dda14e1 を確認

```
$ git cat-file -t b1eb87387a92aa01e2bd12ddf8a7fab28dda14e1
blob
$ git cat-file -p b1eb87387a92aa01e2bd12ddf8a7fab28dda14e1
good morning
```

ファイル b.txt に上書き編集が行われても、Git の管理システムは、最初のコミット時（上書き前）の b.txt の内容を保持しています。ファイルの各バージョンは、それぞれ blob オブジェクトとして保持します。コミットが何度行われたとしても、ファイルに変更がなければ、同じハッシュ値が算出され、同じ blob オブジェクトを指し続けることになります。ファイルに変更があれば、今回のように、各バージョンのデータを個別の blob オブジェクトでスナップショットとして管理することになります。

git merge（Fast-forward）と管理情報

develop ブランチの変更内容を master ブランチに merge して、管理情報の変更内容を見てみます。develop ブランチのコミットログは以下の通りになっています。

develop ブランチのコミットログ

```
$ git branch
* develop
  master

# コミットログの確認
$ git log --graph --pretty=format:"%H %s"
* 9a1df1574f7aa38c6ab1d54c54432cde8a5b57d4 adding second greeting
* be1eb7e8911a20d7c616d20057928d72965f23a1 move from a.txt to b.txt
* 1162a5173ee3a6b9dff9a1d0941a28e4656840f0 first greeting
```

develop ブランチには 3 つのコミットがあります。master ブランチに切り替えます。

master ブランチに切り替える

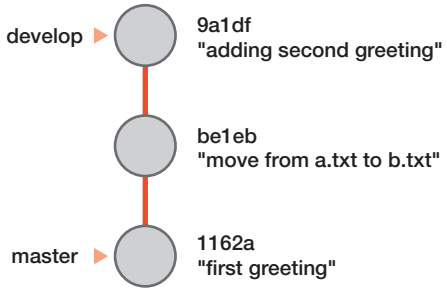
```
$ git branch
* develop
  master

#master ブランチのチェックアウト
$ git checkout master
Switched to branch 'master'
Your branch is up-to-date with 'origin/master'.
$ git branch
  develop
* master
```

master ブランチのコミットログを見てみます。

master ブランチのコミットログの参照

```
$ git log --graph --pretty=format:"%H %s"
* 1162a5173ee3a6b9dff9a1d0941a28e4656840f0 first greeting
```

master ブランチのコミットログに加えて、develop ブランチには 2 つ多くコミットログがあることがわかります。



master ブランチと develop ブランチの差

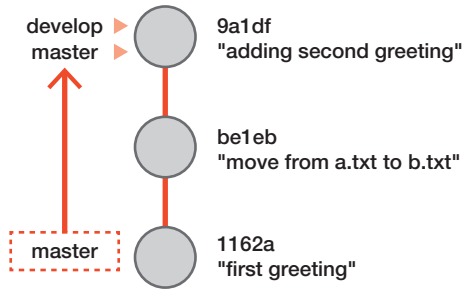
```
$ git rev-list --pretty=format:"%H %s" master...develop
commit 9a1df1574f7aa38c6ab1d54c54432cde8a5b57d4
9a1df1574f7aa38c6ab1d54c54432cde8a5b57d4 adding second greeting
commit be1eb7e8911a20d7c616d20057928d72965f23a1
be1eb7e8911a20d7c616d20057928d72965f23a1 move from a.txt to b.txt
```

develop ブランチのコミットを master ブランチに取り込むため、develop ブランチを master ブランチにマージしてみます。

develop ブランチを master ブランチにマージ

```
$ git merge develop
Updating 1162a51..9a1df15
Fast-forward
 a.txt | 1 -
 b.txt | 2 ++
 2 files changed, 2 insertions(+), 1 deletion(-)
 delete mode 100644 a.txt
 create mode 100644 b.txt
```

コミット 1162a5173ee3a6b9dff9a1d0941a28e4656840f0 を起点とした、develop ブランチの 2 つの追加コミット be1eb7e8911a20d7c616d20057928d72965f23a1 と、9a1df1574f7aa38c6ab1d54c54432cde8a5b57d4 を master ブランチに取り込みました。master ブランチの参照は、コミット 1162a5173ee3a6b9dff9a1d0941a28e4656840f0 であり master ブランチにはその後のコミットが存在しないため、このマージでは分岐がありません。git merge の出力結果を見ると、Fast-forward でマージされていることがわかります。



git merge（Fast-forward）における管理情報の差分を見てみます。

マージ後の管理情報の差分

```
$ tree .git
.git
├── ORIG_HEAD
├── index
├── logs
│   ├── HEAD
│   └── refs
│       ├── heads
│       │   └── master
│
└── refs
    ├── heads
    │   └── master
```

Fast-forward のマージでは、objects/ 以下に変化がないことに注目してください。objects/ 以下のオブジェクトには、全く更新が行われていないことがわかります。では、Fast-forward マージによって何が変わったのでしょうか？

index

```
$ git status
On branch master
Your branch is ahead of 'origin/master' by 2 commits.
  (use "git push" to publish your local commits)
nothing to commit, working directory clean
```

git status を見てみると、origin/master よりコミット 2 つ分（今回は git mv と b.txt の更新です）進んでいると表示されています。.git/refs/heads を確認してみます。

merge 前の .git/refs/heads/master

```
$ cat .git/refs/heads/master
1162a5173ee3a6b9dff9a1d0941a28e4656840f0
```

git merge 前の master は、1162a5173ee3a6b9dff9a1d0941a28e4656840f0 を指しています。

merge 後の .git/refs/heads/master

```
$ cat .git/refs/heads/master
9a1df1574f7aa38c6ab1d54c54432cde8a5b57d4
```

git merge 後の master は、9a1df1574f7aa38c6ab1d54c54432cde8a5b57d4 を指しています。では、マージ後の master ブランチの git log を確認してみます。

コミットログ

```
$ git log --graph --pretty=format:"%H %s"
* 9a1df1574f7aa38c6ab1d54c54432cde8a5b57d4 adding second greeting
* be1eb7e8911a20d7c616d20057928d72965f23a1 move from a.txt to b.txt
* 1162a5173ee3a6b9dff9a1d0941a28e4656840f0 first greeting
```

master ブランチの参照が、1162a5173ee3a6b9dff9a1d0941a28e4656840f0(first greeting) から、9a1df1574f7aa38c6ab1d54c54432cde8a5b57d4(adding second greeting) まで移動していることを、参照ファイル .git/refs/heads/master のハッシュ値の更新で管理していることがわかります。

Fast-forward では、blob オブジェクト・tree オブジェクト・commit オブジェクトに変更を加えることなく、参照を変えるだけで効率的にマージが完了していることがわかります。

git merge（recursive merge）と管理情報

Fast-forward でない場合のマージは、どのように管理されるのでしょうか？ 異なるマージ戦略の状態を作り出して、実際に管理情報の動きを見てみます。最初に、master ブランチの途中 be1eb7e8911a20d7c616d20057928d72965f23a1("move from a.txt to b.txt") から topic_nonff というブランチを作成して、分岐した状態を作ります。

topic_nonff ブランチの作成

```
$ git branch
  develop
* master

# "move from a.txt to b.txt" から topic_nonff ブランチを作成
$ git checkout -b topic_nonff be1eb7e8911a20d7c616d20057928d72965f23a1
Switched to a new branch 'topic_nonff'
$ git branch
  develop
  master
* topic_nonff
```

```
#topic_nonff ブランチでコミットログを確認
$ git log --graph --pretty=format:"%H %s"
* be1eb7e8911a20d7c616d20057928d72965f23a1 move from a.txt to b.txt
* 1162a5173ee3a6b9dff9a1d0941a28e4656840f0 first greeting
```

作成した topic_nonff は、master の途中からチェックアウトしているので、topic_nonff ブランチに直接コミットを加えると、起点となる be1eb7e8911a20d7c616d20057928d72965f23a1("move from a.txt to b.txt") から、分岐した状態を作り出すことができます。ここでは、topic_nonff ブランチに、c.txt を追加してみます。

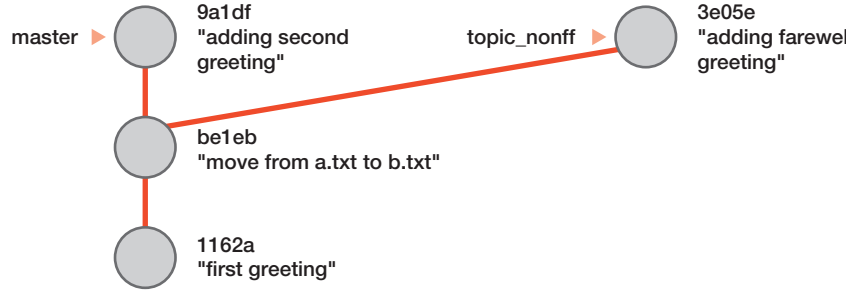
topic_nonff ブランチへの c.txt の追加

```
#c.txt の作成
$ echo 'Good bye' > c.txt
$ ls -a
.    ..    .git    b.txt    c.txt
$ cat c.txt
Good bye

#c.txt のコミット
$ git add c.txt
$ git commit -m "adding farewell greeting"
[topic_nonff 3e05e6a] adding farewell greeting
 1 file changed, 1 insertion(+)
 create mode 100644 c.txt
```

この時点の、topic_nonff ブランチのコミットログは以下のようになっています。

topic_nonff ブランチのコミットログ

```
$ git log --graph --pretty=format:"%H %s"
* 3e05e6abc129077bf33d2a3b2f22c69d11681d78 adding farewell greeting
* be1eb7e8911a20d7c616d20057928d72965f23a1 move from a.txt to b.txt
* 1162a5173ee3a6b9dff9a1d0941a28e4656840f0 first greeting
```



commit オブジェクト 3e05e6abc129077bf33d2a3b2f22c69d11681d78("adding

farewell greeting") の parent を確認します。

3e05e6abc129077bf33d2a3b2f22c69d11681d78 を確認

```
$ git cat-file -t 3e05e6abc129077bf33d2a3b2f22c69d11681d78
commit
$ git cat-file -p 3e05e6abc129077bf33d2a3b2f22c69d11681d78
tree b51a64ab272b1fb317739f51f85e71ebf411b2ab
parent be1eb7e8911a20d7c616d20057928d72965f23a1
author Seigo Kawamura <seigo_kawamura@example.com> 1439016695 +0900
committer Seigo Kawamura <seigo_kawamura@example.com> 1439016695 +0900

adding farewell greeting
```

parent は、be1eb7e8911a20d7c616d20057928d72965f23a1("move from a.txt to b.txt") となっています。

topic_nonff を master にマージするために、master ブランチをチェックアウトします。

master ブランチのチェックアウト

```
$ git checkout master
Switched to branch 'master'
Your branch is ahead of 'origin/master' by 2 commits.
  (use "git push" to publish your local commits)
$ ls -a
.    ..    .git    b.txt
```

master ブランチのコミットログは以下のようになっています。

master ブランチのコミットログ

```
$ git log --graph --pretty=format:"%H %s"
* 9a1df1574f7aa38c6ab1d54c54432cde8a5b57d4 adding second greeting
* be1eb7e8911a20d7c616d20057928d72965f23a1 move from a.txt to b.txt
* 1162a5173ee3a6b9dff9a1d0941a28e4656840f0 first greeting
```

現在 master ブランチが参照している、commit オブジェクト 9a1df1574f7aa38c6ab1d54c54432cde8a5b57d4("adding second greeting") の parent も見ておきます。

9a1df1574f7aa38c6ab1d54c54432cde8a5b57d4 を確認

```
$ git cat-file -t 9a1df1574f7aa38c6ab1d54c54432cde8a5b57d4
commit
$ git cat-file -p 9a1df1574f7aa38c6ab1d54c54432cde8a5b57d4
tree a88781bdbf18ee9ff43912ce0df57cbc17dfe70d
parent be1eb7e8911a20d7c616d20057928d72965f23a1
author Seigo Kawamura <seigo_kawamura@example.com> 1438931924 +0900
```

```
committer Seigo Kawamura <seigo_kawamura@example.com> 1438931924 +0900

adding second greeting
```

master ブランチと topic_nonff ブランチが参照しているコミットは、どちらも parent として be1eb7e8911a20d7c616d20057928d72965f23a1("move from a.txt to b.txt") を指しています。表にすると、以下のようになります。

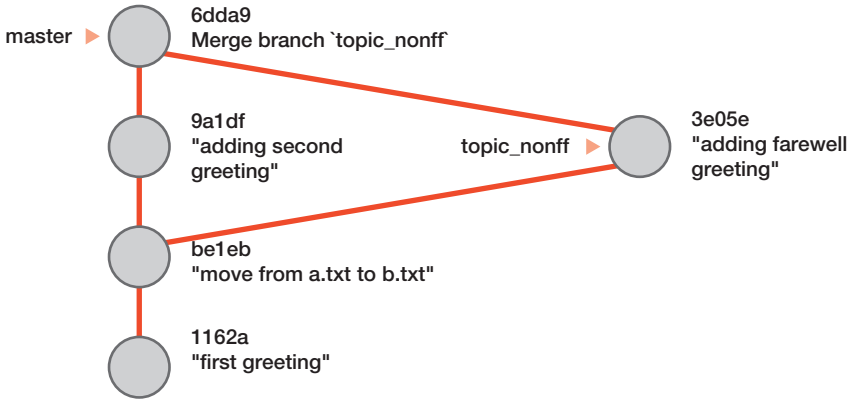
ブランチ	ブランチの参照	コミットの parent
master	9a1df157 ("adding second greeting")	be1eb7e ("move from a.txt to b.txt")
topic_nonff	3e05e6ab ("adding farewell greeting")	be1eb7e ("move from a.txt to b.txt")

topic_nonff ブランチを master ブランチにマージしてみます。

topic_nonff ブランチを master ブランチにマージ

```
# ブランチの確認
$ git branch
  develop
* master
  topic_nonff

#topic_nonff ブランチを master ブランチにマージ
$ git merge topic_nonff
Merge made by the 'recursive' strategy.
 c.txt | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 c.txt
```

このマージでは、Git によってマージ先とマージ元の共通の祖先が自動探索され、マージコミットが生成されます。git merge コマンドが表示するマージ方法について、Fast-forward ではなく、Merge made by the 'recursive' strategy. と表示されています。Git にはいくつかのマージ戦略がありますが、ここでは、デフォルトの'recursive'戦略を解説します。2 つのブランチ topic_nonff の参照しているコミットが、master の参照しているコミットを先祖としないため、共通の先祖 be1eb7e8911a20d7c616d20057928d72965f23a1 ("move from a.txt to b.txt") が Git によって探索され、2 つの子孫 9a1df1574f7aa38c6ab1d54c54432cde8a5b57d4 ("adding second greeting") と 3e05e6abc129077bf33d2a3b2f22c69d11681d78 ("adding farewell greeting") の間で 3 方向マージを行っています。



管理情報には何が起こっているでしょうか。ここでは、マージ前とマージ後の管理情報の差分を見ていきます。まずは、master ブランチのコミットログを見てみます。

master ブランチのコミットログ

```
$ git log --graph --pretty=format:"%H %s"
*   6dda908edca4e4d2024d1d37a7474eff504abab4 Merge branch 'topic_nonff'
|\
| * 3e05e6abc129077bf33d2a3b2f22c69d11681d78 adding farewell greeting
* | 9a1df1574f7aa38c6ab1d54c54432cde8a5b57d4 adding second greeting
|/
* be1eb7e8911a20d7c616d20057928d72965f23a1 move from a.txt to b.txt
* 1162a5173ee3a6b9dff9a1d0941a28e4656840f0 first greeting
```

be1eb7e8911a20d7c616d20057928d72965f23a1("move from a.txt to b.txt") から分岐して、生成されたマージコミット 6dda908edca4e4d2024d1d37a7474eff504abab4("Merge branch 'topic_nonff'") で収束していることが、図から読み取れます。

マージコミットとは

recursive マージ後の管理情報の差分を見ていきます。

マージ後の管理情報の差分

```
$ tree .git
.git
├── ORIG_HEAD
├── index
├── logs
│   ├── HEAD
│   └── refs
│       ├── heads
│       │   ├── master
```

```
│        │
│
├── objects
│   ├── 2a
│   │   └── c5cf7d7da24e2556b9aa494fb2d6b1d09f2303
│   ├── 6d
│   │   └── da908edca4e4d2024d1d37a7474eff504abab4
│
└── refs
    ├── heads
    │   ├── master
```

objects/ の下のマージ後の差分を見ると、わずかに 2 つのオブジェクトが増えただけです。

差分となる 2 つのオブジェクト

```
├── objects
│   ├── 2a
│   │   └── c5cf7d7da24e2556b9aa494fb2d6b1d09f2303
│   ├── 6d
│   │   └── da908edca4e4d2024d1d37a7474eff504abab4
```

commit オブジェクト 6dda908edca4e4d2024d1d37a7474eff504abab4 を見てみます。

6dda908edca4e4d2024d1d37a7474eff504abab4 を確認

```
$ git cat-file -t 6dda908edca4e4d2024d1d37a7474eff504abab4
commit
$ git cat-file -p 6dda908edca4e4d2024d1d37a7474eff504abab4
tree 2ac5cf7d7da24e2556b9aa494fb2d6b1d09f2303
parent 9a1df1574f7aa38c6ab1d54c54432cde8a5b57d4
parent 3e05e6abc129077bf33d2a3b2f22c69d11681d78
author Seigo Kawamura <seigo_kawamura@example.com> 1439017678 +0900
committer Seigo Kawamura <seigo_kawamura@example.com> 1439017678 +0900

Merge branch 'topic_nonff'
```

この commit オブジェクトでは、parent が 2 つになっていることに気が付きましたか？ parent が指している 2 つのコミットオブジェクトは、先ほどの履歴の通り、3e05e6abc129077bf33d2a3b2f22c69d11681d78("adding farewell greeting") と、9a1df1574f7aa38c6ab1d54c54432cde8a5b57d4 ("adding second greeting") です。マージコミットはこのような、分岐を束ねる役割を担います。

2 つになった parent

```
*   6dda908edca4e4d2024d1d37a7474eff504abab4 Merge branch 'topic_nonff'
|\
| * 3e05e6abc129077bf33d2a3b2f22c69d11681d78 adding farewell greeting
* | 9a1df1574f7aa38c6ab1d54c54432cde8a5b57d4 adding second greeting
```

ここで、重要なのは、3e05e6abc129077bf33d2a3b2f22c69d11681d78("adding farewell greeting") と、9a1df1574f7aa38c6ab1d54c54432cde8a5b57d4 ("adding second greeting") のコミットオブジェクトには修正が入っていないという点です。このような、Git では、複雑なマージであっても、1 つのコミットでまとめる効率のいい管理手法が取り入れられています。

最後に、objects/ のもう 1 つの差分ファイルは、tree オブジェクトです。マージによって、b.txt と c.txt が同一ディレクトリに並びましたが、そのスナップショットを記録しています。

2ac5cf7d7da24e2556b9aa494fb2d6b1d09f2303 を確認

```
$ git cat-file -t 2ac5cf7d7da24e2556b9aa494fb2d6b1d09f2303
tree
$ git cat-file -p 2ac5cf7d7da24e2556b9aa494fb2d6b1d09f2303
100644 blob 73670329a9741fd52674409adb06149e535988aa    b.txt
100644 blob c0ee9ab00ab41be0d401f00f7a4aaf2e478f9f1e    c.txt
```

git rebase と管理情報

git rebase で管理情報がどのように変更されるのかを見ていきます。準備として、先ほどの git merge 後の master の状態から、git merge(recursive merge) 前と同じ祖先の分岐状態を作り出します。

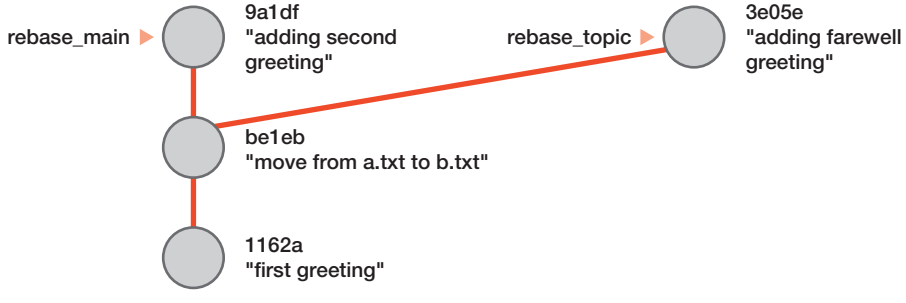
master から祖先の分岐状態を作り出す

```
$ git branch
  develop
* master
  topic_nonff

#master ブランチのコミットログを確認
$ git log --graph --pretty=format:"%H %s"
*   6dda908edca4e4d2024d1d37a7474eff504abab4 Merge branch 'topic_nonff'
|\
| * 3e05e6abc129077bf33d2a3b2f22c69d11681d78 adding farewell greeting
* | 9a1df1574f7aa38c6ab1d54c54432cde8a5b57d4 adding second greeting
|/
* be1eb7e8911a20d7c616d20057928d72965f23a1 move from a.txt to b.txt
* 1162a5173ee3a6b9dff9a1d0941a28e4656840f0 first greeting
```

```
# "adding second greeting" から rebase_main ブランチを作成
$ git branch rebase_main 9a1df1574f7aa38c6ab1d54c54432cde8a5b57d4

# "adding farewell greeting" から rebase_topic ブランチを作成
$ git branch rebase_topic 3e05e6abc129077bf33d2a3b2f22c69d11681d78

# 作成されたブランチを確認
$ git branch
  develop
* master
  rebase_main
  rebase_topic
  topic_nonff
```

rebase_main ブランチの index を 9a1df1574f7aa38c6ab1d54c54432cde8a5b57d4 ("adding second greeting") に、rebase_topic ブランチの index を 3e05e6abc129077bf33d2a3b2f22c69d11681d78 ("adding farewell greeting") に指定しました。これで 2 つのブランチ rebase_main と rebase_topic の共通の祖先は、be1eb7e8911a20d7c616d20057928d72965f23a1 ("move from a.txt to b.txt") になりました。これで準備は完了です。

ブランチ	ブランチの参照	コミットの parent
rebase_main	9a1df157 ("adding second greeting")	be1eb7e8 ("move from a.txt to b.txt")
rebase_topic	3e05e6ab ("adding farewell greeting")	be1eb7e8 ("move from a.txt to b.txt")



rebase_topic ブランチを、rebase_main ブランチへと Fast-forward を使って取り込むことを試みます。その流れの中で、git rebase によって、管理情報がどう変更されるかを見ていきます。

まずは、rebase_topic ブランチをチェックアウトします。

rebase_topic ブランチのチェックアウト

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

```
$ git branch
  develop
  master
  rebase_main
* rebase_topic
  topic_nonff
```

rebase_topic ブランチの git rebase 前のコミットログは以下のようになっています。

rebase 前の rebase_topic ブランチのコミットログ

```
$ git log --graph --pretty=format:"%H %s"
* 3e05e6abc129077bf33d2a3b2f22c69d11681d78 adding farewell greeting
* be1eb7e8911a20d7c616d20057928d72965f23a1 move from a.txt to b.txt
* 1162a5173ee3a6b9dff9a1d0941a28e4656840f0 first greeting
```

rebase_topic ブランチで、git rebase を実行します。

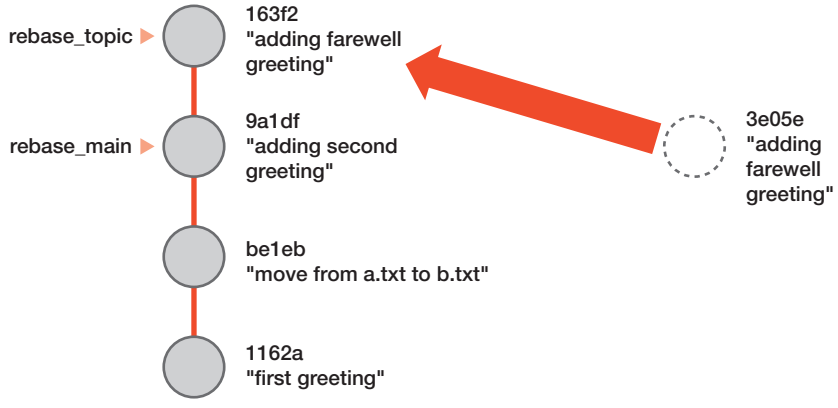
rebase_topic ブランチにおける rebase

```
$ git rebase rebase_main
First, rewinding head to replay your work on top of it...
Applying: adding farewell greeting
```

Applying: adding farewell greeting と表示され、コミットログが修正されたことがわかります。rebase_topic のコミットログを見てみます。

rebase 後のコミットログ

```
$ git log --graph --pretty=format:"%H %s"
* 16e3f2ebee95a4f1fe3170be98872841d7992aae adding farewell greeting
* 9a1df1574f7aa38c6ab1d54c54432cde8a5b57d4 adding second greeting
* be1eb7e8911a20d7c616d20057928d72965f23a1 move from a.txt to b.txt
* 1162a5173ee3a6b9dff9a1d0941a28e4656840f0 first greeting
```



rebase_topic ブランチの最新のコミットを注意深く見てください。"adding farewell greeting" というコミットメッセージは同じですが、ハッシュ値が、3e05e6abc129077bf33d2a3b2f22c69d11681d78 から 16e3f2ebee95a4f1fe3170be98872841d7992aae に変わっています。ハッシュ値は変わりましたが、git rebase コマンドによって、"adding second greeting" というコミットと、"adding farewell greeting" というコミットが一直線に並びました。管理情報の差分を見てみます。

ブランチ	ブランチの参照	コミットの parent
rebase_main	9a1df157 ("adding second greeting")	be1eb7e8 ("move from a.txt to b.txt")
rebase_topic	16e3f2eb ("adding farewell greeting")	9a1df157("adding second greeting")

rebase 後の管理情報の差分

```
$ tree .git
.git
├── HEAD
├── ORIG_HEAD
├── index
├── logs
│   ├── HEAD
│   └── refs
│       ├── heads
│       │   └── rebase_topic
│
├── objects
│   ├── 16
│   │   └── e3f2ebee95a4f1fe3170be98872841d7992aae
│   ├── 2a
│   │   └── c5cf7d7da24e2556b9aa494fb2d6b1d09f2303
│   ├── c0
│   │   └── ee9ab00ab41be0d401f00f7a4aaf2e478f9f1e
│
└── refs
    ├── heads
    │   └── rebase_topic
```

まず、rebase_topic ブランチの参照が更新されています。

.git/refs/heads/rebase_topic

```
$ cat .git/refs/heads/rebase_topic
16e3f2ebee95a4f1fe3170be98872841d7992aae
```

rebase_topic ブランチの参照は、16e3f2ebee95a4f1fe3170be98872841d7992aae であることは先ほどコミットログで確認した通りです。objects/ ディレクトリ以下の差分を見ていきます。

16e3f2ebee95a4f1fe3170be98872841d7992aae を確認

```
$ git cat-file -t 16e3f2ebee95a4f1fe3170be98872841d7992aae
commit
$ git cat-file -p 16e3f2ebee95a4f1fe3170be98872841d7992aae
tree 2ac5cf7d7da24e2556b9aa494fb2d6b1d09f2303
parent 9a1df1574f7aa38c6ab1d54c54432cde8a5b57d4
author Seigo Kawamura <seigo_kawamura@example.com> 1439016695 +0900
committer Seigo Kawamura <seigo_kawamura@example.com> 1439023949 +0900

adding farewell greeting
```


commit オブジェクト 16e3f2ebee95a4f1fe3170be98872841d7992aae("adding farewell greeting") が新規に作成されました。parent は、9a1df1574f7aa38c6ab1d54c54432cde8a5b57d4("adding second greeting") となっており、祖先を変更したコミットを作成することによって、コミットログが分岐せずにまっすぐ並ぶようになりました。このような git rebase は、同じ内容で parent の異なる新しいコミットを作成することによって、起点となる祖先を変更することができます。

もともとの commit オブジェクト 3e05e6abc129077bf33d2a3b2f22c69d11681d78("adding farewell greeting") はどうなっているのでしょうか？　実はそのまま存在しています。

commit オブジェクト 3e05e6abc129077bf33d2a3b2f22c69d11681d78

```
├── objects
│   ├── 3e
│   │   └── 05e6abc129077bf33d2a3b2f22c69d11681d78
```

3e05e6abc129077bf33d2a3b2f22c69d11681d78

```
$ git cat-file -t 3e05e6abc129077bf33d2a3b2f22c69d11681d78
commit
$ git cat-file -p 3e05e6abc129077bf33d2a3b2f22c69d11681d78
tree b51a64ab272b1fb317739f51f85e71ebf411b2ab
parent be1eb7e8911a20d7c616d20057928d72965f23a1
author Seigo Kawamura <seigo_kawamura@example.com> 1439016695 +0900
committer Seigo Kawamura <seigo_kawamura@example.com> 1439016695 +0900

adding farewell greeting
```

古いコミット3e05e6abc129077bf33d2a3b2f22c69d11681d78("adding farewell greeting") の parent は、be1eb7e8911a20d7c616d20057928d72965f23a1("move from a.txt to b.txt") を指しています。新しいコミットの parent は、9a1df1574f7aa38c6ab1d54c54432cde8a5b57d4("adding second greeting") となっていたのは先ほど見た通りです。

それでは、rebase_main ブランチをチェックアウトして、rebase_topic ブランチをマージしてみます。現在の rebase_main ブランチの状態を確認します。

rebase_main ブランチのチェックアウト

```
$ git checkout rebase_main
Switched to branch 'rebase_main'
$ git branch
  develop
  master
* rebase_main
  rebase_topic
  topic_nonff
```

merge 前の rebase_main ブランチのコミットログ

```
$ git log --graph --pretty=format:"%H %s"
* 9a1df1574f7aa38c6ab1d54c54432cde8a5b57d4 adding second greeting
* be1eb7e8911a20d7c616d20057928d72965f23a1 move from a.txt to b.txt
* 1162a5173ee3a6b9dff9a1d0941a28e4656840f0 first greeting
```

rebase_topic ブランチを rebase_main ブランチにマージします。

rebase_topic ブランチを rebase_main ブランチにマージ

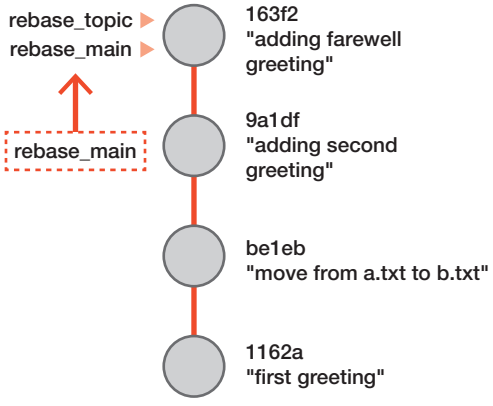
```
$ git merge rebase_topic
Updating 9a1df15..16e3f2e
Fast-forward
 c.txt | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 c.txt
```

Fast-forward と表示されていることを確認してください。「git merge(non Fast-forward)」の節と同じ分岐状態において、git rebase を利用することによって、Fast-forward でマージすることができました。

マージ後の rebase_main ブランチのコミットログ

```
$ git log --graph --pretty=format:"%H %s"
* 16e3f2ebee95a4f1fe3170be98872841d7992aae adding farewell greeting
* 9a1df1574f7aa38c6ab1d54c54432cde8a5b57d4 adding second greeting
* be1eb7e8911a20d7c616d20057928d72965f23a1 move from a.txt to b.txt
* 1162a5173ee3a6b9dff9a1d0941a28e4656840f0 first greeting
```

コミットログに分岐はなく、グラフがきれいにまっすぐ並んでいます。



マージ後の管理情報の差分

```
$ tree .git
.git
├── ORIG_HEAD
├── index
├── logs
│   ├── HEAD
│   └── refs
│       ├── heads
│       │   └── rebase_main
│
└── refs
    ├── heads
    │   └── rebase_main
```

rebase_topic ブランチの rebase_main ブランチへのマージは、Fast-forward で実施されたため、先ほど見てきた通り、参照のみが更新されます。objects/ の下のオブジェクトには変更が発生しません。

.git/refs/heads/rebase_main

```
$ cat .git/refs/heads/rebase_main
16e3f2ebee95a4f1fe3170be98872841d7992aae
```

rebase_main ブランチの参照は、コミットログの通り、16e3f2ebee95a4f1fe3170be98872841d7992aae ("adding farewell greeting") となりました。

03 Git のバージョン管理の仕組みを知る ～上級編～

いよいよ上級編です。管理情報がどのように圧縮されているか、add や commit のコマンドがどのように動いているか理解を深めます。

git gc と管理情報

実際に皆さんがチーム開発で利用しているリポジトリは、今まで説明した管理ファイルの構成とは少し違っていると思います。これまで見てきた Git の管理情報は、開発が進むにつれ、さらに効率の良い方法で管理されていきます。今までの管理情報の知識を踏まえて、実際にどのような方法で効率よく管理されているかを見ていきます。

リポジトリで作業するにつれて、/objects ディレクトリの下に、オブジェクトが増えていきますが、このままでは膨大な量のオブジェクトファイルが作成されることになります。Git には、オブジェクトファイルを圧縮しパックファイル（"packfile"）を作成することによって、より少ない容量でオブジェクトを管理する仕組みがあります。それに対して、ここまで見てきたオブジェクトファイルは、緩いオブジェクトフォーマット（"loose object format"）と呼ばれます。

Git は、git fetch や、git merge、git am、そしてプッシュ時（パックファイル転送）の実行時に、git gc --auto という、--auto オプション付きのコマンドを実行します。この --auto オプションは、git version 2.5 時点では、6700 以上のオブジェクトファイル（緩いオブジェクトフォーマット）か、50 以上のパックファイルがある場合にのみ、git gc を実行します。

ここでは、直接 git gc を実行して、どのようにファイルが管理されるのかを見てみます。

gc の実行

```
$ git gc
Counting objects: 15, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (9/9), done.
Writing objects: 100% (15/15), done.
Total 15 (delta 2), reused 0 (delta 0)
```

git gc 実行後の管理ファイルの差分は以下の通りです。多くのファイルが作成・更新されています。

管理ファイルの変更差分

```
$ tree .git
.git
├── info
│   └── refs
├── logs
```

```
│   ├── HEAD
│   └── refs
│       ├── heads
│       │   ├── develop
│       │   ├── master
│       │   ├── rebase_main
│       │   ├── rebase_topic
│       │   └── topic_nonff
│       └── remotes
│           └── origin
│               └── master
├── objects
│   ├── info
│   │   └── packs
│   └── pack
│       ├── pack-b124a0cc0961d0a7f27d195a38a5370fa95de3a4.idx
│       └── pack-b124a0cc0961d0a7f27d195a38a5370fa95de3a4.pack
├── packed-refs
└── refs
    ├── heads
    ├── remotes
    │   └── origin
    └── tags
```

.git/objects/ ディレクトリの下と、.git/refs/ ディレクトリの下にあるこれまで見てきたファイルが全て消えました。まずは、.git/refs/ の下を見てみます。.git/refs/ 以下の、heads と remotes/origin と tags は、ディレクトリなので、refs/ 以下には参照を格納したファイルがないことがわかります。これらの参照は、新規に作成された .git/packed-refs に格納されています。

.git/packed-refs

```
$ cat .git/packed-refs
# pack-refs with: peeled fully-peeled
9a1df1574f7aa38c6ab1d54c54432cde8a5b57d4 refs/heads/develop
6dda908edca4e4d2024d1d37a7474eff504abab4 refs/heads/master
16e3f2ebee95a4f1fe3170be98872841d7992aae refs/heads/rebase_main
16e3f2ebee95a4f1fe3170be98872841d7992aae refs/heads/rebase_topic
3e05e6abc129077bf33d2a3b2f22c69d11681d78 refs/heads/topic_nonff
1162a5173ee3a6b9dff9a1d0941a28e4656840f0 refs/remotes/origin/master
2fbe3fe9c638172f9bdf4a264e4698d69de2715e refs/tags/version1.0
^1162a5173ee3a6b9dff9a1d0941a28e4656840f0
```

.git/packed-refs には、各ブランチの参照が格納されているのがわかります。先ほどの Fast-forward マージ後の rebase_main の先頭が、変わらず 16e3f2ebee95a4f1fe3170be98872841d7992aae("adding farewell greeting") を参照しています。もともと、.git/refs/ フォルダで管理されていた情報が、git gc 後、.git/packed-refs にまとめて格納されたこと

がわかります。この操作は内部的には、git pack-refs によって行われています。

.git/objects/pack/ 以下には、パックファイルが格納されています。パックファイルのインデックスファイルを見てみます。

pack-b124a0cc0961d0a7f27d195a38a5370fa95de3a4.idx

```
$ git verify-pack -v .git/objects/pack/pack-b124a0cc0961d0a7f27d195a38a5
370fa95de3a4.idx
6dda908edca4e4d2024d1d37a7474eff504abab4 commit 317 205 12
16e3f2ebee95a4f1fe3170be98872841d7992aae commit 56 67 217 1 6dda908edca4
e4d2024d1d37a7474eff504abab4
3e05e6abc129077bf33d2a3b2f22c69d11681d78 commit 267 174 284
9a1df1574f7aa38c6ab1d54c54432cde8a5b57d4 commit 103 108 458 1 3e05e6abc1
29077bf33d2a3b2f22c69d11681d78
be1eb7e8911a20d7c616d20057928d72965f23a1 commit 267 174 566
1162a5173ee3a6b9dff9a1d0941a28e4656840f0 commit 209 135 740
2fbe3fe9c638172f9bdf4a264e4698d69de2715e tag    162 142 875
2ac5cf7d7da24e2556b9aa494fb2d6b1d09f2303 tree   66 71 1017
b51a64ab272b1fb317739f51f85e71ebf411b2ab tree   66 72 1088
a88781bdbf18ee9ff43912ce0df57cbc17dfe70d tree   33 43 1160
0c1ce4b2d1da852658cb88f686cd8d43c90df5e4 tree   33 44 1203
1a24d97271e9e771f2717927f9b7a0a158ecc1fe tree   33 44 1247
73670329a9741fd52674409adb06149e535988aa blob   28 35 1291
c0ee9ab00ab41be0d401f00f7a4aaf2e478f9f1e blob   9 18 1326
b1eb87387a92aa01e2bd12ddf8a7fab28dda14e1 blob   13 22 1344
non delta: 13 objects
chain length = 1: 2 objects
.git/objects/pack/pack-b124a0cc0961d0a7f27d195a38a5370fa95de3a4.pack: ok
```

従来、objects/ の下にあったオブジェクトファイルへの参照が並んでいます。.git/objects/pack/pack-b124a0cc0961d0a7f27d195a38a5370fa95de3a4.idx は、参照インデックスを格納し、各オブジェクトの実データは zlib で圧縮され、pack-b124a0cc0961d0a7f27d195a38a5370fa95de3a4.pack に格納されています。各データがどのぐらい小さくなっているかは、オブジェクトタイプの横に、サイズ・パックファイル内のサイズ・パックファイル内のオフセットの順に、表示されています。

例えば、blob オブジェクト b1eb87387a92aa01e2bd12ddf8a7fab28dda14e1 ("good morning") は、もともとのサイズが 13 バイトで、パックファイル内では 22 バイトであることがわかります（※データサイズが極端に小さいと、このように、圧縮のメリットが得られない場合があります）。

パックの効率的な格納を見るために、少し大きめのテキストをコミットして実験します。まずは、ファイルを作成します。普通にテキストエディタで作成しても構いません。

ファイル d.txt の作成

```
$ git branch
* rebase_main
```

```
#（省略）

# good night を 10 回つぶやく d.txt を作成
$ for i in {1..10}; do echo "good night"; done > d.txt
$ ls
b.txt    c.txt    d.txt
$ cat d.txt
good night
good night
good night
good night
good night
good night
good night
good night
good night
good night
```

d.txt が作成されました。まずは、このままコミットします。

コミット

```
#d.txt をコミット
$ git add d.txt
$ git commit -m "adding good night"
[rebase_main 676b023] adding good night
 1 file changed, 10 insertions(+)
 create mode 100644 d.txt

# コミットログの確認
$ git log --graph --pretty=format:"%H %s"
* 676b02302df075fa4f7836569d4ae413c7d5c46e adding good night
#（省略）

#commit オブジェクトの確認
$ git cat-file -p 676b02302df075fa4f7836569d4ae413c7d5c46e
tree a5d9af000f84e892e6e384dae96197a24f661f47
#（省略）

#tree オブジェクトの確認
$ git cat-file -p a5d9af000f84e892e6e384dae96197a24f661f47
#（省略）
100644 blob d332a58376635708a0f763b6637791761e63c18e    d.txt

#blob オブジェクトの確認
$ git cat-file -p d332a58376635708a0f763b6637791761e63c18e
good night
#（省略）
good night
```

d.txt のデータの blob オブジェクトは、d332a58376635708a0f763b6637791761e

63c18e です。サイズを見ておきます。

d332a58376635708a0f763b6637791761e63c18e のサイズ

```
$ git cat-file -s d332a58376635708a0f763b6637791761e63c18e
110
```

blob オブジェクトのサイズは 110 バイトでした。次に、d.txt の末尾に、適当な文字列を追加して、コミットします。

d.txt の編集を追加コミット

```
#d.txt の最後の行に bye bye! を追記してコミット
$ echo 'bye bye!' >> d.txt
$ git add d.txt
$ git commit -m "adding bye bye"
[rebase_main 84036e1] adding bye bye
 1 file changed, 1 insertion(+)

# コミットログを確認
$ git log --graph --pretty=format:"%H %s"
* 84036e1e4638ee5410dc23dfdb10cf83dafedca8 adding bye bye
(省略)

#commit オブジェクトを確認
$ git cat-file -p 84036e1e4638ee5410dc23dfdb10cf83dafedca8
tree 5e01ba00479820f21f04f944bb40e9a60a8fc8aa
#(省略)

#tree オブジェクトを確認
$ git cat-file -p 5e01ba00479820f21f04f944bb40e9a60a8fc8aa
#(省略)
100644 blob 9f6420eb248d65f290e20b8a6f928cebd99a9f8c    d.txt

#blob オブジェクトを確認
$ git cat-file -p 9f6420eb248d65f290e20b8a6f928cebd99a9f8c
good night
#(省略)
good night
bye bye!
```

変更後の d.txt にあるデータの blob オブジェクトは、9f6420eb248d65f290e20b8a6f928cebd99a9f8c でした。サイズを見ておきます。

9f6420eb248d65f290e20b8a6f928cebd99a9f8c のサイズ

```
$ git cat-file -s 9f6420eb248d65f290e20b8a6f928cebd99a9f8c
119
```

変更前のデータを格納する blob オブジェクトのサイズは 110 バイト、変更後のデータを格納する blob オブジェクトのサイズは、119 バイトでした。ここで、git gc を実行して、パックファイルの中身を確認してみます。

gc 実行後のパックファイルを確認

```
$ git gc
#（省略）

$ git verify-pack -v .git/objects/pack/pack-57697f7f370e0d736da6133a8b19
f8613caf135f.idx
#（省略）
9f6420eb248d65f290e20b8a6f928cebd99a9f8c blob   119 31 1755
d332a58376635708a0f763b6637791761e63c18e blob   4 14 1786 1 \ 9f6420eb24
8d65f290e20b8a6f928cebd99a9f8c
```


変更後のデータを格納する blob オブジェクト 9f6420eb248d65f290e20b8a6f928cebd99a9f8c は、元データサイズ 119 バイト、パックファイル内で 31 バイトとそのまま圧縮されてパックファイルに格納されていることがわかります。しかし、編集前の blob オブジェクト d332a58376635708a0f763b6637791761e63c18e のサイズを見てみると、元データサイズは、4 バイトとなり、もともとのサイズ、110 バイトとはかけ離れています。そして、d332a58376635708a0f763b6637791761e63c18e から、9f6420eb248d65f290e20b8a6f928cebd99a9f8c への参照が格納されています。

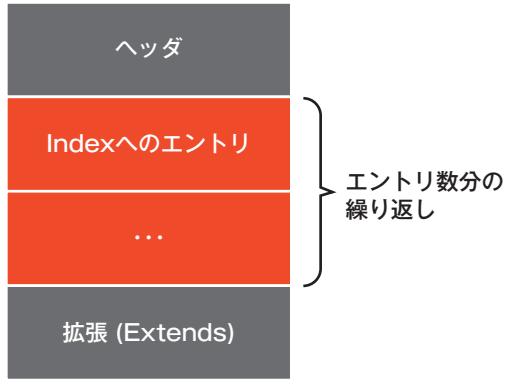
ハッシュ値	blob オブジェクトのサイズ	パック時のサイズ	パックファイル内のサイズ
9f6420eb	119	119	31
d332a583	110	4	14

実は、パックファイルに格納する場合は、このように、git gc の直前のデータは元通り格納し、それより前のデータは差分で格納することで、さらに効率よくデータを小さく格納できるようになっています。今回の例では、119 バイトと 110 バイトの合計 229 バイトの blob データが、31 バイトと 14 バイトの合計 45 バイトまで小さくなっていることがわかります。他のバージョン管理システムよりも、Git の格納データが小さい理由は、このように可能な限り効率よくデータを格納する仕組みにあります。

この項では、Git の管理がファイルを利用してどのように行われているかを見てきました。複雑な操作に対する Git の柔軟性は、このようにシンプルな手法を組み合わせることによって、非常に効率の良い工夫された管理方法で支えられています。Git には多彩なコマンドが用意されており、使いこなしていく上で、挙動が理解できないときは、管理ファイルやオブジェクトがどのように変化するかを見ることによって、理解を進めることができます。

index ファイルの中身を覗く

Git は、"https://github.com/git/git" で開発されています。ソースコードをちょっと覗いて、.git/index にどんな値が入っているのかを覗いてみます。ここでは、「git add と管理ファイル」時点での、.git/index ファイルを見ていきます。まず、.git/index ファイル全体の構造は以下のようになっています。



ヘッダ・エントリ・拡張の大きく 3 つの部分からなり、エントリ部分は、インデックスにエントリされたファイル数分だけ繰り返しとなります。今回は、「git add と管理ファイル」時点での構成になるので、エントリ数は a.txt の 1 つです。

ヘッダ（Header）

最初はヘッダ部分です。.git/index ファイルの構造のヘッダ部分は、cache.h にある cache_header 構造体です。

cache.h

```
#define CACHE_SIGNATURE 0x44495243    /* "DIRC" */
struct cache_header {
    uint32_t hdr_signature;
    uint32_t hdr_version;
    uint32_t hdr_entries;
};
```

hdr はヘッダ（header）の略です。各メンバー 4 バイトずつ、CACHE_SIGNATURE に規定されたキャッシュ署名、バージョン、エントリ数が並んでいます。この構造体の通りにファイルに書き出されているとすると、.git/index/ ファイルの 0 バイト目から順に、cache_header 構造体のメンバーの値が並んでいることになります。.git/index ファイルを、4 バイトずつヘッダに沿って表示してみます。

.git/index のヘッダを確認

```
# 最初の 4 バイトは署名
$ hexdump -C  -n 4 .git/index
00000000  44 49 52 43                                       |DIRC|
# 次の 4 バイトはバージョン
$ hexdump -C  -s 4 -n 4 .git/index
00000004  00 00 00 02                                       |....|
# 次の 4 バイトはエントリ数
$ hexdump -C  -s 8 -n 4 .git/index
00000008  00 00 00 01                                       |....|
```

署名は、ソース上の CACHE_SIGNATURE の define と同じ "DIRC" という文字列が格納され、バージョンは "2"、エントリ数は「git add と管理ファイル」時点では、"a.txt" だけなので "1" となっています。ここまでのヘッダは、.git/index ファイルに対して 1 組のみ格納されています。

エントリ（Entry）

ヘッダ部分に続いて、.git/index ファイルのエントリ部分を見てみます。メモリ上のデータのエントリは、read-cache.c の中で、cache_entry という構造体へ格納して管理していますが、ディスク上に書き出す前に、ondisk_cache_entry へのコピーが行われ、.git/index ファイル内の格納は、以下の ondisk_cache_entry 構造体の形式にならいます。少しだけ見やすくなるように、構造体にコメントを入れておきます。

read-cache.c

```
struct ondisk_cache_entry {
    struct cache_time ctime; // ステータス変更時間（秒・ミリ秒）：8 バイト
    struct cache_time mtime; // 更新時間（秒・ミリ秒）：8 バイト
    uint32_t dev; // データのあるデバイスの ID:4 バイト
    uint32_t ino; //inode：4 バイト
    uint32_t mode; // モード：4 バイト
    uint32_t uid; //user id：4 バイト
    uint32_t gid; //group id：4 バイト
    uint32_t size; // file size：4 バイト
    unsigned char sha1[20]; //SHA-1 hash：20 バイト
    uint16_t flags; // フラグ：2 バイト
    char name[FLEX_ARRAY]; // 名前：可変サイズ
};
```

ctime と mtime を格納する構造体 cache_time は、「秒」と「ミリ秒」を格納する構造体です。

read-cache.h

```
struct cache_time {
```

```
    uint32_t sec; // 秒：4 バイト
    uint32_t nsec; // ミリ秒：4 バイト
};
```

.git/index に、先頭 12 バイトのヘッダに続いて、上記の構造体 ondisk_cache_entry のメンバーの順番通りに、最初のエントリのデータが格納されているはずです。実際に、.git/index 内のエントリのデータと、構造体を照らし合わせて、どのような値が入っているのかを見ていきます。

「git add と管理ファイル」の時点では、a.txt という 1 ファイルを git add した状態です。格納データとしての期待値は、a.txt の 1 エントリのみです。

まずは、ファイルのステータス変更時間 ctime と更新時間 mtime です。.git/index のヘッダ部分のサイズ 12 バイトをスキップし、そこから cache_time 構造体にならって、4 バイトずつ見ていきます。

ctime と mtime を確認

```
#ctime ステータス変更時間：秒
$ hexdump -s 12 -n 4 .git/index
000000c 55 c0 a9 25
#ctime ステータス変更時間：ミリ秒
$ hexdump -s 16 -n 4 .git/index
0000010 00 00 00 00
#mtime 更新時間：秒
$ hexdump -s 20 -n 4 .git/index
0000014 55 c0 a9 25
#mtime 更新時間：ミリ秒
$ hexdump -s 24 -n 4 .git/index
0000018 00 00 00 00
```

ステータス変更時間も、更新時間も、「秒」のパラメータにのみ 55c0a925 が格納されています。

Unixtime を日時フォーマットに変換

```
#16 進数から 10 進数に変換
$ printf "%d\n" 0x55c0a925
1438689573
#Unixtime を時刻フォーマットに変換
$ date -r 1438689573
2015 年 8 月 4 日 火曜日 20 時 59 分 33 秒 JST
```

16 進数から 10 進数に換算すると、1438689573 秒です。date コマンドの -r オプション（MacOS）を利用して、Unixtime（秒）を日付に換算すると、2015/08/04 20:59:33 となります。a.txt のタイムスタンプを、確認してみます。

ファイルのタイムスタンプを確認

```
# 更新時間の確認
$ ls -l a.txt
-rw-r--r--  1 01008453  191071026  13  8  4 20:59 a.txt
# ステータス更新時間の確認
$ ls -lc a.txt
-rw-r--r--  1 01008453  191071026  13  8  4 20:59 a.txt
```

どちらも、2015/08/04 20:59 になっており、.git/index に格納されている時刻と一致します。

.git/index に次に格納されているのは、構造体のメンバー dev の 4 バイトです。データのあるデバイスの ID を格納しています。現在まで見た、28 バイト分をスキップして、29 バイト目から 4 バイト分を見ます。

デバイス ID

```
# 格納されているデバイス ID
$ hexdump -s 28 -n 4 .git/index
000001c 01 00 00 04
#10 進数に変換
$ printf "%d\n" 0x01000004
16777220
```

.git/index には、デバイス ID として 16777220 が格納されていました。stat コマンドの -r オプション（MacOS の場合）を使い、実際のデバイス ID を確認します。

ファイル格納先デバイス ID を確認

```
$ stat -r a.txt
16777220 10084559 0100644 1 969153908 191071026 0 13 1439974653
1438689573 1438689573 1438689573 4096 8 0 a.txt
```

デバイス ID として格納されている 16777220 と一致しました。

.git/index に次に格納されているのは、構造体のメンバー inode です。文字通りファイルの inode を格納しています。今まで見てきた、32 バイト分をスキップして、33 バイト目から 4 バイト分を表示します。

格納されている inode

```
# 格納されている inode
$ hexdump -s 32 -n 4 .git/index
0000020 00 99 e0 cf
#10 進数へ変換
$ printf "%d\n" 0x0099e0cf
10084559
```

inodeの値として、10084559が格納されていました。lsコマンドの-liオプションを使って、inodeを表示してみます。

inodeを確認

```
$ ls -lin a.txt
10084559 -rw-r--r--  1 969153908  191071026  13  8  4 20:59 a.txt
```

格納されているinodeと、出力されたinodeのどちらも、10084559で一致しており、正しくinodeが格納されています。

.git/indexに次に格納されているのは、モードを格納するmodeの4バイトです。37バイト目から4バイトを表示します。

mode

```
#modeの確認
$ hexdump -s 36 -n 4 .git/index
0000024 00 00 81 a4
#modeの8進数換算
$ printf "%o\n" 0x81a4
100644
```

モードは、8進数換算で100644として格納されていることがわかります。a.txtのパーミッションを確認してみます。

パーミッションを確認

```
# パーミッションの8進数表示
$ stat -f '%A' a.txt
644
```

a.txtのパーミッションは644となっており、格納データの下位9bitの8進数換算が644となっていたことより、一致していることがわかります。「git commitと管理ファイル」においてもモードについて触れていますが、モードの先頭4ビットはタイプを表しており、8進数100000というマスクは「通常のファイル」を意味します。100000とパーミッション644との論理和100644は、「実行できない通常のファイル」を意味します。

続いて、41バイト目からのuid（ユーザーID）、45バイト目からのgid（グループID）の2つの構造体メンバーを一気に見ていきます。

uidとgid

```
# 格納されているユーザーID
$ hexdump -s 40 -n 4 .git/index
0000028 39 c4 1d 74
$ printf "%d\n" 0x39c41d74
```

```
969153908
# 格納されているグループ ID
$ hexdump -s 44 -n 4 .git/index
000002c 0b 63 83 32
$ printf "%d\n" 0x0b638332
191071026
```

ユーザー ID に 969153908、グループ ID に 191071026 が格納されています。ls コマンドを使って、a.txt に属しているユーザー ID とグループ ID を確認します。

uid と gid を確認

```
$ ls -n a.txt
-rw-r--r--  1 969153908  191071026  13  8  4 20:59 a.txt
```


ユーザー ID に 969153908、グループ ID に 191071026 が表示され、正しく .git/index に格納されていることがわかります。

次のメンバーは、size です。まずは、49 バイト目から格納されているファイルサイズを確認します。

サイズ

```
# 格納されているファイルサイズ
$ hexdump -s 48 -n 4 .git/index
0000030 00 00 00 0d
#10 進数へ変換
$ printf "%d\n" 0x0d
13
```

ファイルサイズとして 13 という数値が格納されていました。実際のファイルサイズを見てみます。

サイズを確認

```
# サイズの確認
$ ls -n a.txt
-rw-r--r--  1 969153908  191071026  13  8  4 20:59 a.txt
```

格納されている数値と同じ、13 バイトであることが確認できます。

次のメンバー sha1 への期待値は、a.txt のデータを格納する blob オブジェクトのハッシュ値、b1eb87387a92aa01e2bd12ddf8a7fab28dda14e1 です。53 バイト目から 20 バイト分表示してみます。

sha1

```
#sha1 の確認
$ hexdump  -s 52 -n 20 .git/index
0000034 b1 eb 87 38 7a 92 aa 01 e2 bd 12 dd f8 a7 fa b2
0000044 8d da 14 e1
```

16 進数を追っていくと、b1eb87387a92aa01e2bd12ddf8a7fab28dda14e1 がそのまま格納されています。

続いては、フラグ flag です。フラグはファイルの状態管理のために利用されます。

flag

```
#flag の確認
$ hexdump -s 72 -n 2 .git/index
0000048 00 05
```

このフラグは、ファイルシステム側の管理情報ではなく、git の管理情報となります。cache.h で定義されている以下の定数の論理和（OR）が格納されています。

cache.h

```
#define ADD_CACHE_OK_TO_ADD 1        /* Ok to add */
#define ADD_CACHE_OK_TO_REPLACE 2    /* Ok to replace file/directory */
#define ADD_CACHE_SKIP_DFCHECK 4    /* Ok to skip DF conflict checks */
#define ADD_CACHE_JUST_APPEND 8        /* Append only; tree.c::read_
tree() */
#define ADD_CACHE_NEW_ONLY 16        /* Do not replace existing ones */
#define ADD_CACHE_KEEP_CACHE_TREE 32    /* Do not invalidate cache-tree
*/
```

今回 git add した a.txt は、git add する条件に問題がないので、「ADD_CACHE_OK_TO_ADD」と「ADD_CACHE_SKIP_DFCHECK」の論理和が格納されています。フラグには、その他に拡張マスクも存在していますが、ここでは説明を割愛します。

続いて、構造体のメンバー name にファイル名が格納されています。

ファイル名

```
#name の確認
$ hexdump -C -s 74 -n 6 .git/index
0000004a  61 2e 74 78 74 00                                       |a.txt.|
```

メンバー name には、期待通り、ファイル名 a.txt（終端文字を含む）が格納されています。

今回は a.txt1 ファイルのみのエントリとなるので、エントリ部分はここで終わりになります。エントリ数に応じて、エントリが繰り返しされます。

拡張（Extends）

エントリ終了後のここからのデータは、拡張（Extends）部分です。.git/index に、拡張データを持つことができるよう設計されています。「git add と管理ファイル」の段階では、拡張データは格納されていません。拡張には署名が付いています。ファイル名に続く、4 バイトは拡張を見分けるための署名です。

拡張の署名

```
$ hexdump -C -s 80 -n 4 .git/index
00000050  00 00 00 00                                       |....|
```

今回は、拡張がないので、署名にオール 0 が格納されています。拡張がある場合（cached tree または、resolve undo）は、署名の後に 4 バイト分のデータサイズが格納され、その後にデータサイズ分のデータが続きます。今回は、データサイズもデータも、ありません。最後に、.git/index の SHA-1 ハッシュ値が 20 バイト格納されています。

SHA-1 ハッシュ

```
$ hexdump -s 84 -n 20 .git/index
0000054 0d 5c 15 da 94 91 56 95 00 64 83 a0 b3 ff 0a 1d
0000064 7a 90 12 7e
```

以上 104 バイトで、「git add と管理ファイル」時点の .git/index ファイルは終わりです。

index ファイルのサイズを確認

```
$ ls -ln .git/index
-rw-r--r--  1 969153908  191071026  104  8  4 21:00 .git/index
```

今回の事例では、1 ファイルしかステージングされておらず、エントリ数も少ないため、非常にシンプルでしたが、ステージングされるファイルの数が増えればエントリが増え、もう少し複雑になります。ただ、普段、意識することはありませんので安心してください。.git/index ファイルはステージングされたファイルの情報を管理しているということがわかれば、問題ありません。

git add と commit の動きをシェルで実装して理解する

Git は、通常利用する高レベルの機能以外に、管理ファイル自体を操作する低レベルなコマンドが提供されています。低レベルコマンドを使って、git add と git commit を実装し、コマンドの動きを理解しましょう。

git add を実装する

まずは git add でファイルを追加するところから考えていきます。git add では、管理ファイルに対して、どのような作業が行われていたでしょうか？

- blob オブジェクトファイルを作成する
- ステージングする

という 2 つの機能を一度にこなしていました。

blob オブジェクトを作成する

Git の低レベル関数には、git hash-object というコマンドがあります。このコマンドは、オプション指定なしでファイル名を渡すと、指定したファイルからハッシュ値を算出します。

hash-object でハッシュ値の算出

```
# ファイルの作成
$ touch new_file.txt
$ echo "it's new." > new_file.txt
#hash-object でハッシュ値の算出
$ git hash-object new_file.txt
7262b8e6091c85b9d363f30037e4f6cdbb570b20
```

blob オブジェクトのハッシュ値が算出されています。実際に、オブジェクトを作成するには、-w オプションを利用します。

hash-object で blob オブジェクトの作成

```
#hash-object で blob オブジェクトの作成
$ git hash-object -w new_file.txt
7262b8e6091c85b9d363f30037e4f6cdbb570b20
# 作成されたオブジェクトの確認
$ git cat-file -t 7262b8e6091c85b9d363f30037e4f6cdbb570b20
blob
$ git cat-file -p 7262b8e6091c85b9d363f30037e4f6cdbb570b20
it's new.
```

blob オブジェクトが作成されました。ファイルからではなく、標準入力からもオブジェクトを作成できますが今回は利用しません。

標準入力からオブジェクトを作成

```
echo "it's new"  | git hash-object -w --stdin
```

ステージングする

blob オブジェクトの作成ができたら、後は、ステージングすれば、git add の動きを再現できます。オブジェクトからステージングを行うには、git update-index コマンドを利用します。

update-index コマンドで new_file.txt をステージングする

```
#update-index コマンドで new_file.txt をステージングする
$ git update-index --add new_file.txt
# ステージングされたか確認する
$ git status
On branch master
Your branch is ahead of 'origin/master' by 1 commit.
  (use "git push" to publish your local commits)
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   new_file.txt
```

git update-index コマンドの、--add オプションを利用して、new_file.txt をステージングしています。変更の追加であれば、--add オプションは不要です。

git-add.sh を実装する

ここまでで、新規ファイル追加の git add コマンドは作成できそうなので、シェルで実装して git-add.sh を実装してみましょう。git-add.sh の中身は以下のようになっています。

git-add.sh の中身

```
#!/bin/bash

# 指定のファイルがあったら
if [ -e $1 ]; then

# そのファイルの中身でオブジェクト作成
git hash-object -w $1
# ステージングする
git update-index --add $1

fi
```

これを、git-add.sh として保存し、実行パーミッションを付ければ完成です。エラーハンドリングにも対応しておらず、ファイル修正の場合には対応していませんが、ファイル追加してステージングするまでの動きを確認するには十分でしょう。git-add.sh の動作確認をしておきます。

git-add.sh の動作確認

```
# 初期化
$ git init
Initialized empty Git repository in /workspace/git-add-commit/.git/
# ファイルの作成
$ touch sample.txt
$ echo "sample" > sample.txt
#git-add.sh の実行
$ ../git-add.sh sample.txt
d64a3d962e787834f9b43312cdcdb96ef357709a
# ステージングの確認
$ git status
On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

    new file:   sample.txt
```

ステージングまで行われました。最後に blob オブジェクトが作成されたかを確認します。

動作確認

```
#blob オブジェクトが作成された確認する
$ git cat-file -t d64a3d962e787834f9b43312cdcdb96ef357709a
blob
$ git cat-file -p d64a3d962e787834f9b43312cdcdb96ef357709a
sample
```

git add コマンドと同じように動作することができました！次は git commit です。

git commit を実装する

git commit で、修正差分をコミットするとき、どんなことが行われているでしょうか？

- ステージングの内容に沿って tree オブジェクトを作成する
- HEAD がシンボリック参照しているブランチが参照するコミット ID のハッシュを親とした commit オブジェクトを、作成した tree オブジェクトを使って作成する
- 作成した commit オブジェクトに HEAD と、HEAD がシンボリック参照しているブランチの参照を新しいコミット ID に移動

かなり多くの手続きが 1 つの git commit コマンドに集約していることがわかります。1 つ 1 つの動作を低レベルコマンドで実行していきます。

tree オブジェクトを作成する

現在ステージングしている内容を元に、格納すべきワーキングディレクトリの状態を判断してツリーオブジェクトを作成します。実際には、一連の低レベルコマンドの中で最も簡単で、git write-tree コマンドを実行するだけで完了します。

tree オブジェクトを作成する

```
#tree オブジェクトを作成する
$ git write-tree
d1ac811bbf067073c4b59232eb6f39d8fc6bc2f8
#tree オブジェクトの確認
$ git cat-file -t d1ac811bbf067073c4b59232eb6f39d8fc6bc2f8
tree
$ git cat-file -p d1ac811bbf067073c4b59232eb6f39d8fc6bc2f8
100644 blob d64a3d962e787834f9b43312cdcdb96ef357709a    sample.txt
```

git write-tree コマンドの実行の結果、tree オブジェクトが作成され sample.txt と blob オブジェクトの参照が格納されていることがわかります。

commit オブジェクトを作成する

まず、親となるコミット ID を参照するため、現在のコミット ID を把握する必要があります。現在の HEAD がリファレンス参照しているコミット ID を確認するには、git rev-parse コマンドを利用します。

コミット ID を確認

```
# コミット ID の確認
$ git rev-parse HEAD
7d11569398309646da20ef0c6fb012db00946a5d
```

commit オブジェクトを作成するためには、git commit-tree コマンドを利用します。コミットコメントを入れるには、-m オプションを利用します。また、-p オプションを利用して、親となるコミット ID を指定します。親コミット ID を指定しないと、コミット履歴を参照したときに先祖がたどれないコミットとなってしまいます。今回は、上で調べた現在のコミット ID に続けた履歴にしたいので 7d11569398309646da20ef0c6fb012db00946a5d を -p オプションで指定します。

commit オブジェクトを作成する

```
#tree オブジェクトを使って commit オブジェクトを作成する
$ git commit-tree d1ac811bbf067073c4b59232eb6f39d8fc6bc2f8 -m "commit
test" -p 7d11569398309646da20ef0c6fb012db00946a5d
5f973977897dabd7fe88771e293787b715255005
# 作成した commit オブジェクトを参照する
$ git cat-file -t 5f973977897dabd7fe88771e293787b715255005
```

```
commit
$ git cat-file -p 5f973977897dabd7fe88771e293787b715255005
tree d1ac811bbf067073c4b59232eb6f39d8fc6bc2f8
parent 7d11569398309646da20ef0c6fb012db00946a5d
author Seigo Kawamura <seigo_kawamura@example.com> 1443933149 +0900
committer Seigo Kawamura <seigo_kawamura@example.com> 1443933149 +0900

commit test
```

parent に、7d11569398309646da20ef0c6fb012db00946a5d が指定された commit オブジェクトが作成されいていることを確認できました。しかし、この時点では、commit オブジェクトが作成されただけで、HEAD と、HEAD が参照している master ブランチは、1 つ前のコミット ID を参照しています。

作成した commit オブジェクト 5f973977897dabd7fe88771e293787b715255005 がまだ参照されていません。

○参照を移動する

参照を移動するには、git update-ref コマンドを利用します。HEAD や、ブランチのリファレンスを指定することで、参照を特定のコミットに移動できます。今回は、HEAD および HEAD がリファレンス参照しているブランチを移動したいので、HEAD を指定します。

参照を移動する

```
# 参照を移動する
$ git update-ref HEAD 5f973977897dabd7fe88771e293787b715255005
#master の参照が移動したか確認する
$ git rev-parse master
5f973977897dabd7fe88771e293787b715255005
#HEAD のリファレンス参照を確認する
$ git rev-parse HEAD
5f973977897dabd7fe88771e293787b715255005
# コミット履歴を確認する
$ git log --oneline
5f97397 commit test
7d11569 Final fix for release 1.00
```

参照が 5f97397（"commit test"）に移動していることがわかります。親のコミット 7d11569 へと、コミット履歴もたどれています。

同様に、git update-ref コマンドにブランチを指定することで、ブランチの参照を好きなコミットに移動することもできますので活用してみてください。

例

```
# 使用例
$ git update-ref <ブランチ名> <コミット ID>
```

git-commit.sh を実装する

ここまでの手続きを 1 つのシェルにまとめることで、git commit をトレースすることができます。一連のコマンドを並べ、直前の結果を次に受け渡ししながら順番に実行していきます。

git-commit.sh の基本的な実装は以下のようになっています。

git-commit.sh の実装

```
#!/bin/bash

# 現在のステージングの状態で tree オブジェクトを作成
TREEOBJ=`git write-tree`
#HEAD 現在リファレンス参照しているコミット ID を取得
CURRENTHASH=`git rev-parse HEAD`
# 先ほど取得したコミット ID を親とした commit オブジェクトを作成
COMMIT=`echo $1 | git commit-tree $TREEOBJ -p $CURRENTHASH`
# 作成したコミットに HEAD がシンボリック参照しているブランチの参照を移動
git update-ref HEAD $COMMIT
```

このままでも正常系では動作しますし、git-commit.sh として保存し、実行パーミッションを付ければ完成なのですが、リポジトリにおける初回コミットでは、commit コマンドの役割はもう少し複雑です。

- master ブランチを作成する
- 親コミットを指す parent の値は前のコミットではなく、なし、あるいは、オール 0

という作業があります。すでにブランチが 1 つ以上存在し、コミットが存在するかどうかによって多少動きを変える必要がありますので、今回は、実験的に初回コミットをする人も多いということで、ハンドリングを少し増やします。修正した git-commit.sh は以下のようになります。

修正した git-commit.sh

```
#!/bin/bash

# 現在のステージングの状態で tree オブジェクトを作成
TREEOBJ=`git write-tree`
# ブランチ数をカウント
NUM_OF_BRANCH=`git branch | wc -l`

if [ $NUM_OF_BRANCH = 0 ];then
    # ブランチが 1 つもない
    # 親コミット ID のない commit オブジェクトを作成
    COMMIT=`echo $1 | git commit-tree $TREEOBJ`
    #master ブランチを作成
    git branch master $COMMIT
    # 作成したコミットに HEAD がシンボリック参照しているブランチの参照を移動
```

```
    git update-ref HEAD $COMMIT
else
    #HEAD 現在リファレンス参照しているコミット ID を取得
    CURRENTHASH=`git rev-parse HEAD`
    # 先ほど取得したコミット ID を親とした commit オブジェクトを作成
    COMMIT=`echo $1 | git commit-tree $TREEOBJ -p $CURRENTHASH`
    # 作成したコミットに HEAD がシンボリック参照しているブランチの参照を移動
    git update-ref HEAD $COMMIT
fi
```

修正した git-commit.sh では、ブランチの数を数えて、ブランチが存在していない場合に、master ブランチを作成しています。HEAD は、git init の時点から master ブランチをすでにリファレンス参照しているので、作成する必要はありません。また、ブランチが存在しない、リポジトリの初回コミットでは、git commit-tree の -p オプションを使用していません。

あくまで暫定対応ですが、動作はします。では、やってみましょう。先ほど、git-add.sh でステージングした sample.txt をそのままコミットしてみます。

git-commit.sh の実行

```
#git-commit.sh 実行前のステージングの確認
$ git status
On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

    new file:   sample.txt

#git-commit.sh の実行
$ ../git-commit.sh "adding sample.txt"
# 実行後のブランチの確認（今回は初回コミットなので master が作成されます）
$ git branch
* master
#master ブランチの参照の確認
$ git rev-parse master
e46b50af137ce63e9a2ccc5a18f4116a69ee944f
#master ブランチをリファレンス参照している HEAD の確認
$ git rev-parse HEAD
e46b50af137ce63e9a2ccc5a18f4116a69ee944f
#master ブランチのコミット履歴の確認
$ git log master --oneline
e46b50a adding sample.txt
# 作成されたコミットオブジェクトの確認
$ git cat-file -t e46b50a
commit
$ git cat-file -p e46b50a
tree 30ebb81289ebdcdb08633ef3999df098c963c290
```

```
author Seigo Kawamura <seigo_kawamura@example.com> 1443966333 +0900
committer Seigo Kawamura <seigo_kawamura@example.com> 1443966333 +0900

adding sample.txt
```

コミットコメント "adding sample.txt" を入れて、git-commit.sh を実行しました。結果を確認すると、初期化したばかりのリポジトリにおける初回コミットであるため、master ブランチが作成されています。master ブランチの参照を確認すると、参照は作成されたコミット ID に移動しており、初期設定では master ブランチをリファレンス参照している HEAD に関しても同様に同じコミット ID が参照されています。作成されたコミットオブジェクトは、parent の値を持っていません。（通常の git commit における初回コミットではオール 0 の parent 値が挿入されます）

例が初回コミットになってしまったので、2 回目以降のコミットを、git-add.sh と git-commit.sh を連続実行してみましょう。

git-add.sh と git-commit.sh の動作確認

```
# ファイルの追加
$ touch sample2.txt
$ echo "sample2" > sample2.txt
#git-add.sh の実行
$ ../git-add.sh sample2.txt
d45470ccf4d3ee8d677f2ca51ccafec005c42ec7
# ステージングの確認
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   sample2.txt

#git-commit.sh によるコミット
$ ../git-commit.sh "2nd commit"
#master ブランチの参照の確認
$ git rev-parse master
b4f990933bae5d6ff2c8b5a44193adf406ebc5d8
#master ブランチをリファレンス参照している HEAD の確認
$ git rev-parse HEAD
b4f990933bae5d6ff2c8b5a44193adf406ebc5d8
#master ブランチのコミット履歴
$ git log master --oneline
b4f9909 2nd commit
e46b50a adding sample.txt
#commit オブジェクトの確認
$ git cat-file -t b4f9909
commit
$ git cat-file -p b4f9909
tree bf7c67bca6e13bf21b07519f885598d976e469af
```

```
parent e46b50af137ce63e9a2ccc5a18f4116a69ee944f
author Seigo Kawamura <seigo_kawamura@r.example.com> 1443967123 +0900
committer Seigo Kawamura <seigo_kawamura@r.example.com> 1443967123 +0900

2nd commit
```

先ほどの実行履歴でコミットした e46b50a("adding sample.txt") の commit オブジェクトを parent として持つ b4f9909("2nd commit") が作成され、master ブランチの参照と、HEAD のリファレンス参照が新しいコミットに移動しています。

ひと口に git add や git commit といっても、この節で見てきた通り、内部的にはさまざまな処理が行われています。コマンドの内部で行われている一連の処理を理解することで、コマンド実行時にどの部分に何が起きているかを把握し、より Git を使いこなしていけるように、練習として触ってみてください。

まとめ

以上で「上級編〜 Git 内部の仕組みを理解する、は終了です。ここで抑えておきたいポイントは、以下の通りです。

- Git は Git オブジェクトによって履歴が管理されている
- ワーキングディレクトリで見えているファイルやディレクトリの実態と異なり、リポジトリはメタデータと実態にわけたオブジェクトで管理している
- 主に、メタデータ (commit オブジェクト・tree オブジェクト）とファイルの中身の管理（blob オブジェクト）によって管理する
- 全てのオブジェクトはオブジェクトの中身に応じて SHA-1 で生成された一意なハッシュ値を持つ
- オブジェクトの中のデータが全く同一であれば、同じハッシュ値が生成されるため、同じオブジェクトが再利用される
- Git は全ての変更履歴をその時点のスナップショットで保管しており、オブジェクト間の差分管理はしていない
- コミット ID は commit オブジェクトのハッシュ値であり、タイムスタンプ・メールアドレス等を含むため実質世界に唯一の ID となる
- ブランチはコミットへの参照であり、HEAD はブランチ参照への参照（リファレンス参照）である
- ブランチは先頭への参照しか情報として持たず、各ブランチの履歴はコミットの parent を親へ親へとたどる事で実現される
- recursive マージによる統合は、複数の parent をもつマージコミットを作成することで実現し、他には何の変更もないシンプルな方法である
- rebase は同じ内容のコミットを別の parent に付け替えた新しいコミットを行うだけ

で実現される

- Index にはワーキングディレクトリ上のファイルの情報が格納される
- git add では、blob オブジェクトの作成とステージングの２つの操作が行われる
- git commit では、tree オブジェクトの作成後、parent を参照しながら commit オブジェクトの作成を行い、作成した commit オブジェクトにブランチの参照を移動する

Git 全体の運用をまかされた時に、思い通りにブランチ構成を維持したり、特定のブランチの履歴を繰り返し行われるマージを通して思い通りに維持するのは、難しいことです。メンバーが増えていけば、トラブルもその分だけ増え、緊急のトラブルシュートや、Git の管理におけるクリティカルな操作が求められる場面が増えてくるでしょう。その時に、これまで通して学んできた Git の内部構造を把握しておくことで、何が正しい処置で、何をやってはいけないのかの切り分けが一段とクリアになってきます。この知識があれば、低レベルなコマンドを自分で調べて、その時管理ファイルがどう変更されそうかなどを想像できるようになっており、新しいトラブルでも動揺することなく対応できます。また、普段のブランチの維持や履歴の変更も、ぐっと楽になっていることと思います。

一度通しで上級編を読むことができた人は、ぜひ実際に手を使いながら管理ファイルを見ていただければと思います。Git のブラックボックスだと思っていた部分が明らかになることで、最初にもっていたイメージと、全く違った動作に見えてくるでしょう。あとは、Git によるバージョン管理を楽しんでください！

デザイン　宮嶋章文
企画・ディレクション　関根康浩
編集・DTP　株式会社リブロワークス

エンジニアのための Git の教科書［上級編］

Git 内部の仕組みを理解する

2016 年 1 月 19 日　初版 1 刷発行

著　　者　河村 聖悟
発 行 人　佐々木 幹夫
発 行 所　株式会社 翔泳社（http://www.shoeisha.co.jp）

ISBN 978-4-7981-4591-4