

やりなおしGit

本書は [Git](#) を何となく使っているけど、もっと理解を深めるために学習しなおすためのドキュメントです。

他でもない著者が何となく Git を使っているものの、もう少ししっかりと勉強しなおすかと重い腰をあげて調べたことをまとめただけです。

基本的に本書の内容は Git のドキュメント、および [Pro Git 2nd Edition](#) に基づいています。

すべてを網羅しているわけではないのであしからず。

対象

- Gitを何となく使っている人

本ドキュメントについて

- この文書は Zenn で公開しているものを PDF にしたものです。オリジナルは「やりなおしGit」にあります。オリジナルの方では一部画像がアニメーションになっていてわかりやすいものがあります。
- この PDF は `mdbook` によって作成されました。

はじめに

Git とは何かという根本的な説明はここでは割愛します。知りたい方はAIなどに聞いてください。

動作環境は Windows で、PowerShellを使います。詳しくは次の通りです。

```
Windows 11 Pro 22H2  
PowerShell 7.3.2  
git version 2.39.1.windows.1
```

本書では Rust の環境があることを想定しています。Rustについては [Rust入門](#) などを参考にしてください。

Info

現在は開発環境だけでなくソフトウェアのインストールも含めて、 Go , Rust , Python , Node の環境は必須だと思っています。

▷ 使用ツール

ファイルやディレクトリツリーの表示などで次のツールを使います。

fd

<https://github.com/sharkdp/fd>

find コマンドのクローンです。インストールは cargo instal fd-find です。

LSD (LSDeluxe)

<https://github.com/Peltoche/lsd>

ls コマンドのクローンです。インストールは cargo install lsd です。

hexyl

<https://github.com/sharkdp/hexyl>

バイナリビューアです。インストールは cargo install hexyl です。

bat

<https://github.com/sharkdp/bat>

cat コマンドのクローンです。これは任意で。インストールは cargo install bat です。

delta

<https://github.com/dandavison/delta>

差分表示ツールです。これも任意で。インストールは `cargo install git-delta` です。

➤ PowerShell関数

上記のツールに加えて、以下のPowerShell関数を使います。

lat

`lsd`を使ってディレクトリツリーを表示する関数です。隠しファイルも表示します。

```
function lat() {
    lsd -A --tree $args
}
```

laf/laft

`fd`を使って現在のディレクトリ以下にあるファイルを列挙し、最終更新時間で降順ソートしてオブジェクトを返します。オブジェクトはファイル名、最終更新時間、サイズプロパティを持っています。ファイルサイズの単位を自動で設定し、右詰めで表示するようにした `laft` も作成しました。

```
function laf() {
    fd -t f -H | ForEach-Object {
        $prop = (Get-ItemProperty $_)
        [PSCustomObject]@{
            Name=$_
            LastWriteTime=$prop.LastWriteTime
            Size=($prop.Length | Convert-Size)
        }
    } | Sort-Object LastWriteTime -Descending
}
function laft() {
    laf | Format-Table Name, LastWriteTime, @{n='Size';e={$_.Size};align='right'}
}
```

ファイルサイズの変換に使っている `Convert-Size` は以下から拝借。

<https://stackoverflow.com/questions/24616806/powershell-display-files-size-as-kb-mb-or-gb>

```

Function Convert-Size {
    <#
    .SYNOPSIS
        Converts a size in bytes to its upper most value.

    .DESCRIPTION
        Converts a size in bytes to its upper most value.

    .PARAMETER Size
        The size in bytes to convert

    .NOTES
        Author: Boe Prox
        Date Created: 22AUG2012

    .EXAMPLE
        Convert-Size -Size 568956
        555 KB

    Description
    -----
    Converts the byte value 568956 to upper most value of 555 KB

    .EXAMPLE
        Get-ChildItem | ? {! $.PSIsContainer} | Select -First 5 | Select Name, @{L='Size';E={$_.Convert-Size}}
        Name                                Size
        ----
        Data1.cap                            14.4 MB
        Data2.cap                            12.5 MB
        Image.iso                            5.72 GB
        Index.txt                            23.9 KB
        SomeSite.lnk                          1.52 KB
        SomeFile.ini                          152 bytes

    Description
    -----
    Used with Get-ChildItem and custom formatting with Select-Object to list the uppermost size.

#>
[cmdletbinding()]
Param (
    [parameter(ValueFromPipeline=$True,ValueFromPipelineByPropertyName=$True)]
    [Alias("Length")]
    [int64]$Size
)
Begin {
    If (-Not $ConvertSize) {
        Write-Verbose ("Creating signature from Win32API")
        $Signature = @"
            [DllImport("Shlwapi.dll", CharSet = CharSet.Auto)]
```

```
    public static extern long StrFormatByteSize( long fileSize, System.Text.StringBuilder buffer,
int bufferSize );
"@  
    $Global:ConvertSize = Add-Type -Name SizeConverter -MemberDefinition $Signature -PassThru
}  
    Write-Verbose ("Building buffer for string")
    $stringBuilder = New-Object Text.StringBuilder 1024
}
Process {
    Write-Verbose ("Converting {0} to upper most size" -f $Size)
    $ConvertSize::StrFormatByteSize( $Size, $stringBuilder, $stringBuilder.Capacity ) | Out-Null
    $stringBuilder.ToString().Replace("/バイト", " B")
}
}
```

オリジナルコードから　バイト　を　B　として表示するように調整しています。

■ セットアップ

➤ Git のインストール

WinGetを使いましょう。

```
PS > winget install Git.Git
```

➤ .gitconfig

Gitの設定ファイルはWindowsなら `C:\Users\<user>\.gitconfig` です。これは `--global` の設定ファイルです。`--system` の場合、環境やバージョンによって変わるように。その場合、`git config -e --system` とすればシステムの設定ファイルがエディタで開かれます。ちなみに、私の環境では `C:\Program Files\Git\etc\config` です。

設定の確認

リストで確認するなら、以下のコマンドを使います。

```
PS > git config --list
PS > git config --global --list
PS > git config --system --list
```

個別は次の通りです。

```
PS > git config user.name
PS > git config --global user.email
```

ユーザー名とメールアドレス

最初に設定すべきこと。

```
PS > git config --global user.name "Yourname"
PS > git config --global user.email "your@email.com"
```

エディタの設定

Visual Studio Code なら次のようにします。

```
PS > git config --global core.editor "code --wait"
```

改行の扱い

Windowsなら `input` がいいと思います。標準は `false`。あとは、必要であればプロジェクトごとに設定します。

```
PS > git config core.autocrlf input
```

`core.autocrlf` の設定は次の関係になっています。

値	index	worktree
true	LF	CRLF
input	LF	(pass)
false	(pass)	core.eol

`input` の場合、インデックス（ステージングするとき）に `LF` に変換されます。クローンしたときや、追跡されていないときのファイルは変換しません。

Info

`core.autocrlf` に `false` を指定したとき、作業ツリーの改行は `core.eol` に設定したものになります。`core.eol` には `native` , `lf` , `crlf` を指定できます。

設定の取り消し

`--unset` を使います。

```
PS > git config --unset core.autocrlf
```

④ その他の設定

いくつか便利かもしれないもの。

core.quotePath

日本語の文字化け対策。

```
PS > git config --global core.quotePath false
```

init.defaultBranch

デフォルトブランチの名前を設定。`main` にしておきましょう。

```
PS > git config --global init.defaultBranch main
```

fetch.prune

リモートをfetch時に自動で --prune を指定します。

```
PS > git config --global fetch.prune true
```

pull.ff

pull時に fast-forward でマージできるなら merge を、そうでなければ fetch のみを行います。

```
PS > git config --global pull.ff only
```

autosquash

rebase時に自動で --autosquash を指定します。

```
PS > git config --global rebase.autosquash true
```

■ 基本

① 定義

残念なことに表記が結構ぶれていて、いくつもの言い方があって混乱しやすくなっているのが現状です。本書では次のようにします。

snapshot (スナップショット)

バージョンごとのファイルそのもの。

worktree (作業ツリー)

ローカルにあるファイルを編集する場所です。リモートには存在しません。作業ツリーがないリポジトリのことを bare リポジトリといいます。

index (インデックス)

次のコミットのスナップショットです。ステージングエリアとも言われます。作業ツリーからインデックスにスナップショットを記録することを staging(ステージング) といいます。そのため、staged とも言われます。他には cached とも言います。Gitのコマンドでは、stage(d) , cached を指定することがあるので覚えておく必要があります。

database (データベース)

Gitオブジェクト(後述)のデータベースです。ローカルリポジトリとも言われる部分ですが、ローカルリポジトリはコミット履歴や設定ファイルなどさまざまな情報が含まれているものです。

Note

インデックスは作業ツリーとデータベースの橋渡し的な存在です。bareリポジトリには作業ツリーがありません。ですから、インデックスも必要ありません。

② ファイルの状態

Git から見た作業ツリーのファイルは次の3種類に分けられます。

- バージョン管理されていないファイル (Untracked)
- バージョン管理されていて、更新されていないファイル (Unmodified)
- バージョン管理されていて、更新されたファイル (Modified)

バージョン管理されていないファイルを管理対象とするには、まずステージングしてコミットします。

Untracked -> Staged -> Unmodified

バージョン管理されているファイル(Unmodified)を編集して更新すると(Modified)になって、ステージングできます。

Unmodified -> Modified

ステージングすると更新されたファイルのスナップショットがインデックスに登録されます。

Modified -> Staged

これらの関係は次の図をご覧ください。

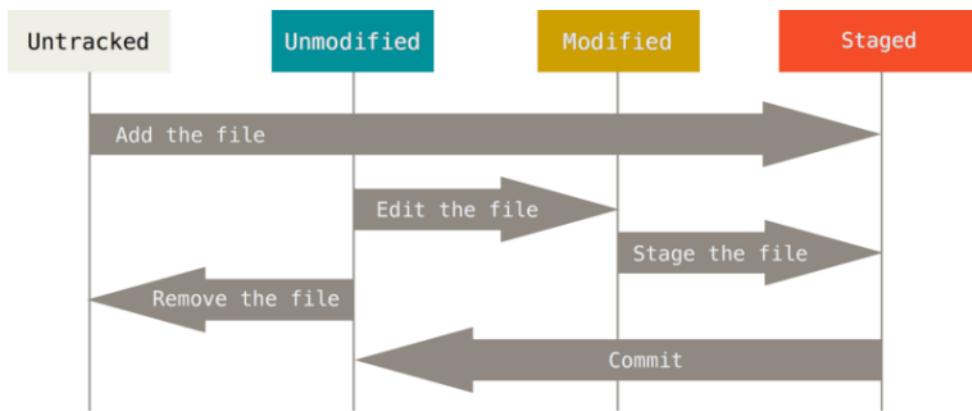


図 8. ファイルの状態の流れ

▲図：ファイルの状態（「Pro Git 2nd edition 日本語版」p.25）

➤ Gitリポジトリ

git init コマンドでGitリポジトリを作成できます。

```
PS > git init
Initialized empty Git repository in F:/re-git/.git/
```

Gitリポジトリの中は次のようにになっています。

```
PS > lat
.
└── .git
    ├── hooks
    │   ├── applypatch-msg.sample
    │   ├── commit-msg.sample
    │   ├── fsmonitor-watchman.sample
    │   ├── post-update.sample
    │   ├── pre-applypatch.sample
    │   ├── pre-commit.sample
    │   ├── pre-merge-commit.sample
    │   ├── pre-push.sample
    │   ├── pre-rebase.sample
    │   ├── pre-receive.sample
    │   ├── prepare-commit-msg.sample
    │   ├── push-to-checkout.sample
    │   └── update.sample
    ├── info
    │   └── exclude
    ├── objects
    │   ├── info
    │   └── pack
    ├── refs
    │   ├── heads
    │   └── tags
    ├── config
    ├── description
    └── HEAD
```

hooks/

フックスクリプトを格納するディレクトリです。ここでは詳しく解説しません。

Info

説明のために、hooksディレクトリは削除します。削除したあとも、`git init` コマンドでリポジトリを再初期化することでhooksディレクトリを復元することができます。

```
PS > rm -Force .git\hooks
PS > git init
Reinitialized existing Git repository in F:/re-git/.git/
```

info/exclude

バージョン対象外にしたいファイルを指定できます。これは `.gitignore` でも同様のことができますが、ローカル環境でのみ適用したい場合に使います。

```
PS > 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]
# *~
```

objects/

Gitオブジェクトを格納するディレクトリです。Gitオブジェクトについては後述します。

refs/

参照情報を格納するディレクトリです。参照とは、コミットを指すポインタのことです。実体はコミットのハッシュ値が記録されたファイルです。 refs/heads/ ディレクトリには各ブランチの最新のコミット情報が格納されています。

config

ローカルの設定ファイルです。

```
PS > cat .git\config
[core]
    repositoryformatversion = 0
    filemode = false
    bare = false
    logallrefupdates = true
    symlinks = false
    ignorecase = true
```

filemode

filemode はパーミッションの設定です。true だと環境によってパーミッションが変化する場合があります。デフォルトは false です。

bare

bare はペアリポジトリかどうかです。false なら作業ツリーを認識します。

symlinks

symlinks はシンボリックリンクを有効にするかどうかです。デフォルトは false です。

repositoryformatversion

repositoryformatversion はリポジトリのフォーマットバージョンです。extensions.* キーを指定する場合は 1 を、そうでないなら 0 を指定します。

ignorecase

ignorecase はファイル名の大文字小文字を区別しないかどうかです。

logallrefupdates

logallrefupdates は reflog を取るかどうかです。ペアリポジトリでは false 、そうでないなら true になります。

description

Gitリポジトリの説明が格納されています。

```
PS > git cat .git\description
Unnamed repository; edit this file 'description' to name the repository.
```

HEAD

現在のHEAD情報です。HEADは通常ブランチを指します。

```
PS > cat .git\HEAD
ref: refs/heads/main
```

Gitリポジトリ初期化時では refs/heads/main が存在しないので、この HEAD は無効になっています。また、ブランチではなくコミットを直接指すこともあります。この場合、 detached HEAD となります。

➡ ファイルの状態を確認

git status で確認できます。確認できるのは作業ツリーとインデックスでの状態です。

```
PS > git status
On branch main

No commits yet

nothing to commit (create/copy files and use "git add" to track)
```

➡ ファイルの追加

バージョン管理したいファイルを追加してみましょう。まずファイル(README.md)を作成します。

```
PS > echo 'My Project' > README.md
PS > git status
On branch main

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    README.md

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

ファイル状態を確認すると README.md ファイルは Untracked であることがわかります。バージョン管理するには、まずステージングします。これは `git add` を使います。

```
PS > git add README.md
warning: in the working copy of 'README.md', CRLF will be replaced by LF the next time Git touches it
PS > git status
On branch main

No commits yet

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

>Note

`add` したときに以下の表示を見ることができます。

```
warning: in the working copy of 'README.md', CRLF will be replaced by LF the next time Git touches it
```

これは作業ツリーのファイルで、改行コードが CRLF になっており、データベースに登録するときに LF に変換されますという意味です。リポジトリには LF で格納されますので、特に問題がなければ無視して構いません。もし気になるのであれば、作業ツリーにあるファイルの改行コードを LF に変換してください。

出力にもある通り、ここで `git rm --cached` を使うとインデックスから取り除くことができます。

```
PS > git rm --cached README.md
```

このとき、作業ツリーにあるファイルは削除されません。もし、作業ツリーにあるファイルも無かったことにしたい場合、`-f` を指定します。

```
PS > git rm -f README.md
```

ファイルを追加した状態でGitリポジトリの中身はどうなっているか確認します。

```
PS > ls
```

```
.
```

```
|   └── .git
```

```
|       ├── info
```

```
|       |   └── exclude
```

```
|       ├── objects
```

```
|       |   ├── 56
```

```
|       |       └── 266d360f3da9f922766101055bd78ffa3724bf
```

```
|       |   ├── info
```

```
|       |   └── pack
```

```
|       ├── refs
```

```
|       |   ├── heads
```

```
|       |       └── tags
```

```
|       ├── config
```

```
|       ├── description
```

```
|       ├── HEAD
```

```
|       └── index
```

```
└── README.md
```

objectsディレクトリにファイルが追加され、indexファイルが作成されています。これらが一体何なのか理解するためにGitオブジェクトについて知る必要があります。

➤ Gitオブジェクト

GitオブジェクトはGitがバージョン管理するために作成するオブジェクトのことです。これらは `.git\objects` ディレクトリに作成されます。本書ではこのディレクトリのことをデータベースと呼ぶことにします。

Gitオブジェクトは以下の4種類あります。

- blobオブジェクト
- treeオブジェクト
- commitオブジェクト
- tagオブジェクト

Gitオブジェクトはバイナリ形式で、名前がハッシュ値(SHA-1)になっています。データベースにはハッシュ値の先頭2文字のディレクトリの中に、その文字分除外したファイル名で格納されています。ですから、

`56/266d360f3da9f922766101055bd78ffa3724bf` というのは `56266d360f3da9f922766101055bd78ffa3724bf` というハッシュ値になります。

Gitオブジェクトはその種類とその情報が圧縮されて保存されています。これらを中身を見るためのコマンド(`git cat-file`)が用意されています。

`-t` を指定すると種類、`-p` を指定すると情報、`-s` を指定すると圧縮前のサイズがわかります。

```
PS > git cat-file -t 56266d360f3da9f922766101055bd78ffa3724bf
blob

PS > git cat-file -p 56266d360f3da9f922766101055bd78ffa3724bf
My Project

PS > git cat-file -s 56266d360f3da9f922766101055bd78ffa3724bf
11
```

README.mdファイルを追加したこと、blobオブジェクトが生成されたことがわかります。

indexファイル

indexは次のコミットのスナップショットであると前述しました。その実体がこのファイルになります。indexファイルはバイナリで、バージョン管理しているすべてのスナップショットを記録しています。

```
PS > hexyl .git\index
```

これは次のように出力されます。

00000000	44 49 52 43 00 00 00 02	00 00 00 00 39 d8 90 13	DIRC....9xx..
00000010	9e e5 35 6c 7e f5 72 21	6c eb cd 27 aa 41 f9 df	xx5l~xr!	lxx'xAxx

このファイルに記録されている情報を確認するコマンドがあります。それは `git ls-files --stage` です。

```
PS > git ls-files --stage
100644 56266d360f3da9f922766101055bd78ffa3724bf 0 README.md
```

これは次のような形式になっています。

```
<mode> <object> <stage> <file>
```

mode

100644 は 0b100000110100100 のことで、3つのパート(10000-000-110100100)に分けられます。

1000 の部分は、ファイルの種類です。次の種類があります。

- 1000 : regular file
- 1010 : symbolic link
- 1110 : gitlink

次の 000 は使用されていません。最後の 110100100 はUNIXパーミッションです。通常ファイルでは 0755 と 0644 のみです。シンボリックリンクとgitリンクは 0 です。

object

Gitオブジェクトのハッシュ値です。

stage

ステージ番号はマージ衝突時に使われます。

- Slot 0: “normal”, unconflicted , all-is-well entry.
- Slot 1: “base”, the common ancestor version.
- Slot 2: “ours”, the target (HEAD) version.
- Slot 3: “theirs”, the being-merged-in version.

file

ファイル名です。

Note

git ls-files はオプションを指定しなければ、バージョン管理しているファイルの一覧が表示されます。

Note

ファイルを作成して追加した後、 git rm でインデックスから取り除いたり、作業ツリーから削除してもGitオブジェクトは残ったままになります。これはキャッシュとして再利用できるので、インデックスはキャッシュであるとも考えられますね。ですから、 index, staged, cached はどれもが同じでもあります。細かいことは知りませんけど。では、このキャッシュはいつになつたら消えるのかについては別のところで。

➤ コミット

インデックスに更新されたスナップショットが記録されていればコミットできます。コミットしてみましょう。

```
PS > git commit
```

上記のコマンドを実行するとコミットメッセージ(.git\COMMIT_EDITMSG ファイル)がエディタで開かれます。メッセージを書いて保存し、閉じるとコミットが実行されます。もし、コマンドと一緒にメッセージを書きたい場合、 -m を指定します。

```
PS > git commit -m "first commit"
[main (root-commit) b4c9631] first commit
 1 file changed, 1 insertion(+)
 create mode 100644 README.md
```

この時点のGitリポジトリがどうなっているか確認してみます。

```
PS > lat
.
├── .git
│   ├── info
│   │   └── exclude
│   ├── logs
│   │   └── refs
│   │       └── heads
│   │           └── main
│   └── HEAD
├── objects
│   ├── 56
│   │   └── 266d360f3da9f922766101055bd78ffa3724bf
│   ├── b4
│   │   └── c96315ae08b029192fa57b46b035d0317018cc
│   ├── c8
│   │   └── a887332c98b8d43304c2d8c4ba0389f2665d60
│   ├── info
│   └── pack
└── refs
    ├── heads
    │   └── main
    └── tags
├── COMMIT_EDITMSG
├── config
├── description
├── HEAD
└── index
└── README.md
```

いくつか追加されていますね。更新されたファイルをわかりやすく見てみましょう。

```
PS > laft
```

これは次のような出力になります。

Name	LastWriteTime	Size
.git\logs\refs\heads\main	2023/02/12 19:09:29	162 B
.git\logs\HEAD	2023/02/12 19:09:29	162 B
.git\refs\heads\main	2023/02/12 19:09:29	41 B
.git\objects\b4\c96315ae08b029192fa57b46b035d0317018cc	2023/02/12 19:09:29	125 B
.git\COMMIT_EDITMSG	2023/02/12 19:09:29	13 B
.git\index	2023/02/12 19:09:29	137 B
.git\objects\c8\a887332c98b8d43304c2d8c4ba0389f2665d60	2023/02/12 19:09:29	53 B
.git\objects\56\266d360f3da9f922766101055bd78ffa3724bf	2023/02/12 18:36:24	27 B
README.md	2023/02/12 18:36:21	12 B
.git\config	2023/02/12 18:13:17	130 B
.git\HEAD	2023/02/12 18:13:17	21 B
.git\info\exclude	2023/02/12 18:13:17	240 B
.git\description	2023/02/12 18:13:17	73 B

タイムスタンプを見ると以下のファイルがコミット時に生成または更新されたことがわかります。

```
.git\logs\refs\heads\main  
.git\logs\HEAD  
.git\refs\heads\main  
.git\objects\b4\c96315ae08b029192fa57b46b035d0317018cc  
.git\COMMIT_EDITMSG  
.git\index  
.git\objects\c8\c887332c98b8d43304c2d8c4ba0389f2665d60
```

それぞれ見ていきましょう。

logs/

コミットの変更が記録されます。それぞれのファイルは次のようになっています。

```
PS > cat .git\logs\refs\heads\main  
00000000000000000000000000000000 b4c96315ae08b029192fa57b46b035d0317018cc mebiusbox <mebiusbox@gmail.com>  
1676196569 +0900 commit (initial): first commit  
  
PS > cat .git\logs\HEAD  
00000000000000000000000000000000 b4c96315ae08b029192fa57b46b035d0317018cc mebiusbox <mebiusbox@gmail.com>  
1676196569 +0900 commit (initial): first commit
```

これは 00000000000000000000000000000000 から b4c96315ae08b029192fa57b46b035d0317018cc のコミットに変わったことを表しています。この履歴は git reflog で確認できます。

```
PS > git reflog  
b4c9631 HEAD@{0}: commit (initial): first commit
```

refs/heads/main

これは main ブランチの最新のコミットを指す参照です。内容は次のようになっています。

```
PS > cat .git\refs\heads\main  
b4c96315ae08b029192fa57b46b035d0317018cc
```

つまり、 b4c96315ae08b029192fa57b46b035d0317018cc はコミットを表すオブジェクトであることが想像できると思います。

commit オブジェクト

この commit オブジェクトを見てましょう。

```
PS > git cat-file -t b4c96315ae08b029192fa57b46b035d0317018cc
commit

PS > git cat-file -p b4c96315ae08b029192fa57b46b035d0317018cc
tree c8a887332c98b8d43304c2d8c4ba0389f2665d60
author mebiusbox <mebiusbox@gmail.com> 1676196569 +0900
committer mebiusbox <mebiusbox@gmail.com> 1676196569 +0900

first commit

PS > git cat-file -s b4c96315ae08b029192fa57b46b035d0317018cc
175
```

commitオブジェクトの中は、 tree , author , committer があります。この tree オブジェクトを見てみましょう。

Note

authorとcommitterの違いは何でしょうか。Gitではコミットした後に、コミットの履歴を書き換えることができます。書き換えたユーザーがcommitterになります。オリジナルがauthorです。しかし、オプションでauthorも書き換えることができます。

treeオブジェクト

先ほどのcommitオブジェクトが指していたtreeオブジェクトを見てみましょう。

```
PS > git cat-file -t c8a887332c98b8d43304c2d8c4ba0389f2665d60
tree

PS > git cat-file -p c8a887332c98b8d43304c2d8c4ba0389f2665d60
100644 blob 56266d360f3da9f922766101055bd78ffa3724bf README.md

PS > git cat-file -s c8a887332c98b8d43304c2d8c4ba0389f2665d60
37
```

treeオブジェクトの中身はインデックスに書かれたものに似ていますね。treeオブジェクトはディレクトリを表すので、ツリー構造になっています。今回はルートディレクトリにあるファイルだけですが、サブディレクトリがあれば、treeオブジェクトの中で別のtreeオブジェクトを参照します。

Gitオブジェクトと参照

これまで見てきたように、HEAD はブランチを指し、ブランチはcommitオブジェクトを参照し、commitオブジェクトはtreeオブジェクトを参照し、treeオブジェクトはblobオブジェクトやTreeオブジェクトを参照します。

```
HEAD -> Branch -> commit -> tree -> blob, tree
```

今回は最初のコミットでしたが、通常はcommitオブジェクトに親のコミット情報(parent)が含まれています。

これらの関係から HEAD やブランチ、タグは唯一のコミットを特定できるので、Gitコマンドでコミットを指定する部分に使うことができます。Gitコマンドのパラメータで commit-ish と指定されている部分は、コミットが特定できるものなら指定できることになります。そして、参照しているコミットを調べるコマンド(git rev-parse)があります。

```
PS > git rev-parse main  
b4c96315ae08b029192fa57b46b035d0317018cc
```

```
PS > git rev-parse HEAD  
b4c96315ae08b029192fa57b46b035d0317018cc
```

④ 初回コミットの取り消し

最初のコミットを取り消すにはどうすればよいでしょうか。それには、 git update-ref コマンドを使います。これは参照を安全に書き換えることができるコマンドです。 -d オプションを指定して、HEADを無効な値(00000000000000000000000000000000 = 0)にすることで、初回のコミットを取り消すことができます。コミットのログで、40文字分の0が並んだ値は、無効な値だったんですね。

```
PS > git update-ref -d HEAD
```

.git/logs/HEAD はHEADを操作するとその記録が残ります。もう一度見てみましょう。

```
PS > cat .git/logs/HEAD  
00000000000000000000000000000000 b4c96315ae08b029192fa57b46b035d0317018cc mebiusbox <mebiusbox@gmail.com>  
1676196569 +0900 commit (initial): first commit  
b4c96315ae08b029192fa57b46b035d0317018cc 00000000000000000000000000000000 mebiusbox <mebiusbox@gmail.com>  
1676294876 +0900
```

無効な値に戻っていることが確認できます。しかし、この状態では reflog では参照できません。

```
PS > git reflog
```

何も表示されません。なぜでしょうか。ここからは不正確な情報なので注意です。まず、git reflog は以下のコマンドのエイリアスです。

```
git log -g --abbrev-commit --pretty=oneline
```

-g (--walk-reflogs) オプションに注目です。ドキュメントには次のように書かれています。

Instead of walking the commit ancestry chain, walk reflog entries from the most recent

one to older ones.

要約すると、コミット履歴を辿るのではなく、reflogを辿ると書いてあります。まあ、その通りです。このコマンドで表示されるのは現在のブランチのreflogです。現在のブランチ(main)のreflogは .git/logs/refs/heads/main です。しかし、コミットがない状態だとこのファイルは存在しません。では、初回コミットして、git update-ref コマンドを使って取り消した後に、さらにその処理を取り消したい場合はどうするのでしょうか。再コミットすればいいという野暮な発言は却下です。.git/logs ディレクトリを見てみると .git/logs/HEAD は残っています。これを使うことは出来ないでしょうか。

もちろん、できます。この場合、git log --reflog を使います。この --reflog オプションは何でしょうか。ドキュメントには次のように書かれています。

Pretend as if all objects mentioned by reflogs are listed on the command line as <commit>.

これも要約すると、reflogに記録されているすべてのGitオブジェクトをコミットとして表示するそうです。試してみましょう。

```
PS > git log --reflog
commit b4c96315ae08b029192fa57b46b035d0317018cc
Author: mebiusbox <mebiusbox@gmail.com>
Date:   Sun Feb 12 19:09:29 2023 +0900

first commit
```

表示されました。--oneline を指定すれば1行で表示されます。

```
PS > git log --reflog --oneline
b4c9631 first commit
```

データベースには、キャッシュとしてまだ残っています。これを使って復元しましょう。

```
PS > git reset --hard b4c9631
HEAD is now at b4c9631 first commit
```

ログを見ると次のようになっています。

```
PS > git log
commit b4c96315ae08b029192fa57b46b035d0317018cc (HEAD -> main)
Author: mebiusbox <mebiusbox@gmail.com>
Date:   Sun Feb 12 19:09:29 2023 +0900

first commit
```

問題なさそうです。ここで、reflogを見てみましょう。

```
PS > git reflog  
b4c9631 (HEAD -> main) HEAD@{0}: reset: moving to b4c9631  
b4c9631 (HEAD -> main) HEAD@{2}: commit (initial): first commit
```

resetしたことがきちんと記録されています。コミットも元に戻っているので、`.git/logs/refs/heads/main` も存在します。見てみましょう。

```
PS > cat .git\logs\refs\heads\main
00000000000000000000000000000000 b4c96315ae08b029192fa57b46b035d0317018cc mebiusbox <mebiusbox@gmail.com>
1676295921 +0900      reset: moving to b4c9631
```

reflogで表示されている HEAD@{2} の情報はここにないですね。.git/logs/HEAD を見てみましょう。

こちらを参照しているようです。このあたりの挙動はどうなっているのかはちょっとわかりません。

▶ ガベージコレクト

せっかく復元したのですが、また初回コミットを取り消しましょう。

```
PS > git update-ref -d HEAD
```

コミットはなくなりましたが、作業ツリーとインデックスには残ったままです。

```
PS > git status
On branch main

No commits yet

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

    new file:   README.md
```

すべて無かったことにしましょう。git reset コマンドを使います。

PS > git reset --hard

作業ツリー やインデックスからは削除されました。確認してみましょう。

```
PS > lat
.
└── .git
    ├── info
    │   └── exclude
    ├── logs
    │   ├── refs
    │   │   └── heads
    │   └── HEAD
    ├── objects
    │   ├── 56
    │   │   └── 266d360f3da9f922766101055bd78ffa3724bf
    │   ├── b4
    │   │   └── c96315ae08b029192fa57b46b035d0317018cc
    │   ├── c8
    │   │   └── a887332c98b8d43304c2d8c4ba0389f2665d60
    │   ├── info
    │   └── pack
    ├── refs
    │   ├── heads
    │   └── tags
    ├── COMMIT_EDITMSG
    ├── config
    ├── description
    ├── HEAD
    └── index
```

データベースには、Gitオブジェクトが残っています。もし、これがこのまま使われなければ不要ですよね。不要なGitオブジェクトを削除する `git prune` コマンドがあります。試しに、`-n` を指定(dry-run)して確認します。

```
PS > git prune -n
```

何も表示されません。`git prune` コマンドが削除するオブジェクトは到達できないオブジェクトです。作業ツリーやインデックスになくとも、前のところで説明したとおり、`.git/logs/HEAD` から参照しているので、到達できます。では、どうするかというとreflogを削除します。通常は `git reflog` コマンドを使って削除します。しかし、今回は初回コミットなので、`.git\logs\HEAD` を直接削除します。

```
PS > rm .git\logs\HEAD
```

それでは、もう一度 `git prune` コマンドを実行してみましょう。

```
PS > git prune -n
56266d360f3da9f922766101055bd78ffa3724bf blob
b4c96315ae08b029192fa57b46b035d0317018cc commit
c8a887332c98b8d43304c2d8c4ba0389f2665d60 tree
```

表示されました。実際に削除してみましょう。`-v` を指定すれば削除されたファイルが出力されます。

```
PS > git prune -v
56266d360f3da9f922766101055bd78ffa3724bf blob
b4c96315ae08b029192fa57b46b035d0317018cc commit
c8a887332c98b8d43304c2d8c4ba0389f2665d60 tree
```

確認してみましょう。

```
PS > ls
.
└── .git
    ├── info
    │   └── exclude
    ├── logs
    │   └── refs
    │       └── heads
    ├── objects
    │   ├── info
    │   └── pack
    ├── refs
    │   ├── heads
    │   └── tags
    ├── COMMIT_EDITMSG
    ├── config
    ├── description
    ├── HEAD
    └── index
```

削除されていますね。

追跡されていないファイルに関しては `git prune` で削除できることがわかりました。それでは追跡できるファイルはどうでしょうか。Gitはスナップショットを保存しているので、ファイルが増えれば増えるほど、バージョンが重なるたびにデータベースが肥大化していくことになります。それに対して、Gitはガベージコレクトで容量節約する機能がしっかりあります。そのコマンドは `git gc` です。先ほど見た `git prune` コマンドはガベージコレクトの処理の1つとして実行されるようです。

ここまで手順をやってきた人は、ほとんど空っぽなGitリポジトリの状態になっていると思います。ガベージコレクトを試すために、再び初回コミットして、そのコミットを取り消してリセットした状態にしてください。

```
PS > echo 'My Project' > README.md
PS > git add README.md
PS > git commit -m "first commit"
PS > git update-ref -d HEAD
PS > git reset --head
PS > lat
.
└── .git
    ├── info
    │   └── exclude
    ├── logs
    │   ├── refs
    │   │   └── heads
    │   └── HEAD
    ├── objects
    │   ├── 56
    │   │   └── 266d360f3da9f922766101055bd78ffa3724bf
    │   ├── 81
    │   │   └── 1993fddcde91487288bcccd3016971cab6c7a4
    │   ├── c8
    │   │   └── a887332c98b8d43304c2d8c4ba0389f2665d60
    │   ├── info
    │   └── pack
    ├── refs
    │   ├── heads
    │   └── tags
    ├── COMMIT_EDITMSG
    ├── config
    ├── description
    ├── HEAD
    └── index
```

Note

commitオブジェクトのハッシュ値が変わるのは?

ユーザ名など変更せず、同じ手順をしているのにcommitオブジェクトのハッシュ値が実行する度に変わります。これは、commitオブジェクトにタイムスタンプが含まれているからです。

reflogは残した状態です。データベースにあるGitオブジェクトは追跡可能なので、`git prune` コマンドでは消せません。この状態で、`git gc` コマンドを実行しましょう。

```
PS > git gc
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Writing objects: 100% (4/4), done.
Total 4 (delta 0), reused 0 (delta 0), pack-reused 0
```

4つのオブジェクトが処理されたようです。確認してみましょう。

```
PS > lat
.
└── .git
    ├── info
    │   └── exclude
    │   └── refs
    ├── logs
    │   └── refs
    │       └── heads
    └── HEAD
    ├── objects
    │   ├── info
    │   │   └── packs
    │   └── pack
    │       ├── pack-c3e54044977a6ee8b9652fd225f51978c072b92c.idx
    │       └── pack-c3e54044977a6ee8b9652fd225f51978c072b92c.pack
    ├── refs
    │   ├── heads
    │   └── tags
    ├── COMMIT_EDITMSG
    ├── config
    ├── description
    ├── HEAD
    ├── index
    └── packed-refs
```

いくつかファイルが追加されています。

```
PS > laf | %{$_.Name}
.git\info\refs
.git\objects\info\packs
.git\objects\pack\pack-c3e54044977a6ee8b9652fd225f51978c072b92c.idx
.git\objects\pack\pack-c3e54044977a6ee8b9652fd225f51978c072b92c.pack
.git\logs\HEAD
.git\packed-refs
...
```

(info/refs)

ここにはブランチの一覧が格納されています。今回の例では特に関係がありません。サイズも 0 です。

objects/info/packs

ファイルの中身は次のようになっています。

```
PS > cat .git\objects\info\packs  
P pack-c3e54044977a6ee8b9652fd225f51978c072b92c.pack
```

pack-c3e54044977a6ee8b9652fd225f51978c072b92c.pack を参照しているようです。

objects/pack

2つのファイルが追加されています。

- pack-c3e54044977a6ee8b9652fd225f51978c072b92c.idx
- pack-c3e54044977a6ee8b9652fd225f51978c072b92c.pack

これらは packfile と呼ばれるものです。これはGitオブジェクトを1つのバイナリファイル(.pack)にまとめたものと、そのインデックスファイル(.idx)です。blobオブジェクトはファイルのスナップショットです。最初はファイルの完全なデータが含まれています。このようなオブジェクトのことを loose object といいます。Gitはガベージコレクトで loose object をバイナリファイルにまとめます。その際、似たような名前とサイズのファイルを探して、ファイルのあるバージョンから次のバージョンまでの差分を格納します。これで容量を節約します。packfileは git verify-pack コマンドで中身を確認できます。 -v を指定し詳細を表示してみましょう。 git verify-pack コマンドには packfile のインデックスファイルを指定します。

```
PS > git verify-pack -v .\.git\objects\pack\pack-c3e54044977a6ee8b9652fd225f51978c072b92c.idx  
d566fe024bafa6e4e7be6bd429ee1a96010f3fba commit 175 118 12  
4b825dc642cb6eb9a060e54bf8d69288fbbe4904 tree 0 9 130  
c8a887332c98b8d43304c2d8c4ba0389f2665d60 tree 37 47 139  
56266d360f3da9f922766101055bd78ffa3724bf blob 11 20 186  
non delta: 4 objects  
.\.git\objects\pack\pack-c3e54044977a6ee8b9652fd225f51978c072b92c.pack: ok
```

目録や差分を格納したかどうか、対応するバイナリファイルの情報が表示されています。このようにガベージコレクトを実行することで容量の節約ができるわけです。ガベージコレクトは定期的に実行されます。また、リポジトリの更新頻度が高いとガベージコレクトを促すプロンプトが出力された気がします。

Note

git verify-pack のGitオブジェクト情報は次のようになっています

```
<SHA-1> <type> <size> <size-in-packfile> <offset-in-packfile> [<depth>] [<base-SHA-1>]
```

logs/HEAD

packfileの内容によって情報が書き換わっています。

packed-refs

.git/refs ディレクトリにある情報が packed-refs ファイルにまとめられます。

➤ .git/ORIG_HEAD

コミットを行うと、その前の有効なHEADがORIG_HEADに記録されます。これを使えば、1つ前のHEADに戻すことができます。

```
PS > git reset --hard ORIG_HEAD
```

▣ サンプルプロジェクト(Rust)

Gitを使ったサンプルプロジェクトとして、Rustの単純なバイナリパッケージを使います。ここでは、基本では説明しなかった部分を少し解説します。また、このプロジェクトは以降の説明で使用することがあります。

④ プロジェクトの作成

以下のコマンドでRustのパッケージを作成します。

```
PS > cargo new re-git
Created binary (application) package
```

または、先に `re-git` ディレクトリを作成しておいて、中で次のコマンドを実行します。

```
PS > cargo init
Created binary (application) package
```

このときのディレクトリツリーは次のようにになっています。(hooksフォルダは削除しています)

```
PS > ls
.
├── .git
│   ├── info
│   │   └── exclude
│   ├── objects
│   │   ├── info
│   │   └── pack
│   ├── refs
│   │   ├── heads
│   │   └── tags
│   ├── config
│   ├── description
│   └── HEAD
└── src
    └── main.rs
└── .gitignore
└── Cargo.toml
```

⑤ ファイルの状態

`git status` コマンドでファイル状態を確認できます。

```
PS > git status
On branch main

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    .gitignore
    Cargo.toml
    src/

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

-s または --short を指定すると短縮表示されます。

```
PS > git status -s
?? .gitignore
?? Cargo.toml
?? src/
```

各行の先頭2文字がファイルの状態を表しています。1文字目はインデックス、2文字目は作業ツリーでのファイル状態を表しています。以下の表は代表的なものです。

##	説明
??	追跡されていない
A_	インデックスにスナップショットを追加している
M_	インデックスのスナップショットを更新している
_M	作業ツリーのスナップショットを更新している
MM	インデックスと作業ツリーのスナップショットを更新している

➤ .gitignore

.gitignore ファイルはバージョン管理から除外するファイルを指定します。ディレクトリを指定することも可能です。

```
PS > cat .gitignore
/target
```

この例では、ルートディレクトリ直下の target ディレクトリが除外されています。.gitignore は次のようになっています。

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

.gitignore のテンプレートが以下の場所で公開されています。

<https://github.com/github/gitignore>

また、このテンプレートをブラウザで検索して生成するサービスもあります。

<https://www.toptal.com/developers/gitignore>

このファイルの効果を確認するために、プロジェクトをビルドします。

```
PS > cargo build
Compiling re-git v0.1.0 (F:\re-git)
Finished dev [unoptimized + debuginfo] target(s) in 1.26s
```

ディレクトリツリーを確認してみると次のようになります。

```
PS > lat
.
├── .git
│   ├── info
│   │   └── exclude
│   ├── objects
│   │   ├── info
│   │   └── pack
│   ├── refs
│   │   ├── heads
│   │   └── tags
│   ├── config
│   ├── description
│   └── HEAD
└── src
    └── main.rs
├── target
    ├── debug
    │   ├── .fingerprint
    │   │   └── re-git-8bc8387ee01ca6c3
    │   │       ├── bin-re-git
    │   │       ├── bin-re-git.json
    │   │       ├── dep-bin-re-git
    │   │       └── invoked.timestamp
    │   ├── build
    │   ├── deps
    │   │   ├── re_git.d
    │   │   ├── re_git.exe
    │   │   └── re_git.pdb
    │   ├── examples
    │   ├── incremental
    │   │   └── re-git-25t8xd60jcr1i
    │   │       ├── s-gi7ndmztp1-sq6ioe-1tmg7305nz81n
    │   │       │   ├── 1anjrfyimb5ng9rd.o
    │   │       │   ├── 391sfavvdd56qi16.o
    │   │       │   ├── 3hpznkcqqj7rlnjk.o
    │   │       │   ├── 3xr84zbncooiwpnq.o
    │   │       │   ├── 53vh9ifml86q7sz5.o
    │   │       │   ├── 562ebllc75edeyo5.o
    │   │       │   ├── dep-graph.bin
    │   │       │   ├── mzgrtbckyxvieww4.o
    │   │       │   ├── query-cache.bin
    │   │       │   ├── vv48kxfqqfg2tru.o
    │   │       │   └── work-products.bin
    │   │       └── s-gi7ndmztp1-sq6ioe.lock
    │   ├── .cargo-lock
    │   ├── re-git.d
    │   ├── re-git.exe
    │   └── re-git.pdb
└── .rustc_info.json
```

```
|   └─ CACHEDIR.TAG  
├─ .gitignore  
├─ Cargo.lock  
└─ Cargo.toml
```

target ディレクトリにビルドの成果物が出力されています。また、Cargo.lock ファイルも生成されています。ここでファイルの状態を確認してみましょう。

```
PS > git status -s  
?? .gitignore  
?? Cargo.lock  
?? Cargo.toml  
?? src/
```

Gitからは target ディレクトリが除外されています。そして、Cargo.lock は追跡されていないファイルとして認識されています。

ステージングの取り消し

それでは、これらのファイルをステージングします。

```
PS > git add .
```

ファイルの状態を確認すると次のようになります。

```
PS > git status -s  
A .gitignore  
A Cargo.lock  
A Cargo.toml  
A src/main.rs
```

ステージングを取り消したい場合、 git reset を使います。

```
PS > git reset Cargo.toml  
PS > git status -s  
A .gitignore  
A Cargo.lock  
A src/main.rs  
?? Cargo.toml
```

ファイルを指定しなければ、ステージングしているすべてのファイルが対象となります。

```
PS > git reset  
PS > git status -s  
?? .gitignore  
?? Cargo.lock  
?? Cargo.toml  
?? src/
```

では、もう一度ステージングしてコミットしておきましょう。

```
PS > git add .  
PS > git commit -m "first commit"  
[main (root-commit) fdc9308] first commit  
 4 files changed, 19 insertions(+)  
 create mode 100644 .gitignore  
 create mode 100644 Cargo.lock  
 create mode 100644 Cargo.toml  
 create mode 100644 src/main.rs
```

コミットログ

前回のサンプルプロジェクトを使って説明します。コミットログを表示するには `git log` コマンドを使います。

```
PS > git log
commit fdc930870489a7f9afc8d84f2eea50a389ffb596
Author: mebiusbox <mebiusbox@gmail.com>
Date:   Tue Feb 14 10:50:12 2023 +0900

first commit
```

今は1つだけしかコミットがないので、何回かコミットしましょう。`src/main.rs` を書き換えてコミットします。

```
fn main() {
    println!("Hello, Git!");
}
```

`git add` コマンドに `-u` を追加すると、バージョン管理しているファイルを対象に変更しているスナップショットがあればステージングします。

```
PS > git add -u
```

また、コミットするときに `-a` を追加するとコミットする前に上記のコマンドと同じことをしてくれます。それではコミットします。

```
PS > git commit -m "second commit"
[main 096663c] second commit
 1 file changed, 1 insertion(+), 1 deletion(-)
```

続いて、もう1回コミットしましょう。`src/lib.rs` ファイルを作成して次のようにします。

```
pub fn hello() {
    println!("Hello, Git!");
}
```

そして、`src/main.rs` も書き換えます。

```
use re_git::hello;

fn main() {
    hello();
}
```

動作確認してみます。次のコマンドを実行します。

```
PS > cargo run
Compiling re-git v0.1.0 (F:\re-git)
Finished dev [unoptimized + debuginfo] target(s) in 0.28s
Running `target\debug\re-git.exe`
Hello, Git!
```

問題なければコミットします。

```
PS > git add .
PS > git commit -m "add lib module"
[main 96b3dce] add lib module
2 files changed, 6 insertions(+), 1 deletion(-)
create mode 100644 src/lib.rs
```

この時点でのコミットログを確認してみましょう。

```
PS > git log
commit 96b3dced736be2f2ca3a4c66bacbec7caacfdd64
Author: mebiusbox <mebiusbox@gmail.com>
Date:   Tue Feb 14 11:12:04 2023 +0900

    add lib module

commit 096663ced4acef379cbbeafe94439cefc21539b7
Author: mebiusbox <mebiusbox@gmail.com>
Date:   Tue Feb 14 11:05:05 2023 +0900

    second commit

commit fdc930870489a7f9afc8d84f2eea50a389ffb596
Author: mebiusbox <mebiusbox@gmail.com>
Date:   Tue Feb 14 10:50:12 2023 +0900

    first commit
```

pretty

--pretty で出力する際の書式を指定できます。 oneline や format などがあります。

```
PS > git log --pretty=oneline
96b3dced736be2f2ca3a4c66bacbec7caacfdd64 add lib module
096663ced4acef379cbbeafe94439cefc21539b7 second commit
fdc930870489a7f9afc8d84f2eea50a389ffb596 first commit
```

```
PS > git log --pretty=format:"%h - %an, %ar : %s"
96b3dce - mebiusbox, 2 hours ago : add lib module
096663c - mebiusbox, 2 hours ago : second commit
fdc9308 - mebiusbox, 2 hours ago : first commit
```

書式で指定できるものは次の通りです。

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

書式で `%C` を指定すると色を付けることができます。例えば、`%C(green)%h` または `%C(green)%h` というように指定します。色を元に戻す場合、`%reset` または `%(reset)` とします。

```
PS > git log --date=short --decorate=short --pretty=format:"%C(green)%h %C(reset)%cd %C(yellow)%cn %C(cyan)%d %C(reset)%s"
```

```
96b3dce 2023-02-14 mebiusbox (HEAD → main) add lib module
096663c 2023-02-14 mebiusbox second commit
fdc9308 2023-02-14 mebiusbox first commit
```

④ ハッシュ値の短縮形

`--abbrev-commit` を指定すると、ハッシュ値が一意に特定できる範囲の文字数に省略されて出力されます。通常は 7 文字です。

```
PS > git log --abbrev-commit --pretty=oneline
96b3dce add lib module
096663c second commit
fdc9308 first commit
```

この `--abbrev-commit --pretty=oneline` は `--oneline` でも同じです。

```
PS > git log --oneline
96b3dce add lib module
096663c second commit
fdc9308 first commit
```

変更箇所の表示

`-p` を指定することで変更した箇所を出力します。

```
PS > git log -p
commit 96b3dced736be2f2ca3a4c66bacbec7caacfdd64
Author: mebiusbox <mebiusbox@gmail.com>
Date:   Tue Feb 14 11:12:04 2023 +0900

add lib module

diff --git a/src/lib.rs b/src/lib.rs
new file mode 100644
index 000000..3317b77
--- /dev/null
+++ b/src/lib.rs
@@ -0,0 +1,3 @@
+pub fn hello() {
+    println!("Hello, Git!");
+}

diff --git a/src/main.rs b/src/main.rs
index 869abe6..7552ba0 100644
--- a/src/main.rs
+++ b/src/main.rs
@@ -1,3 +1,5 @@
+use re_git::hello;
+
fn main() {
-    println!("Hello, Git!");
+    hello();
}

...
```

`-p -2` というように `-<n>` を指定すると、直近のその数だけ出力します。

```
PS > git log -p -2
commit 96b3dced736be2f2ca3a4c66bacbec7caacfdd64
Author: mebiusbox <mebiusbox@gmail.com>
Date:   Tue Feb 14 11:12:04 2023 +0900
```

```
    add lib module

diff --git a/src/lib.rs b/src/lib.rs
new file mode 100644
index 000000..3317b77
--- /dev/null
+++ b/src/lib.rs
@@ -0,0 +1,3 @@
+pub fn hello() {
+    println!("Hello, Git!");
+}
diff --git a/src/main.rs b/src/main.rs
index 869abe6..7552ba0 100644
--- a/src/main.rs
+++ b/src/main.rs
@@ -1,3 +1,5 @@
+use re_git::hello;
+
fn main() {
-    println!("Hello, Git!");
+    hello();
}


```

```
commit 096663ced4acef379cbbeafe94439cefc21539b7
Author: mebiusbox <mebiusbox@gmail.com>
Date:   Tue Feb 14 11:05:05 2023 +0900
```

```
    second commit

diff --git a/src/main.rs b/src/main.rs
index e7a11a9..869abe6 100644
--- a/src/main.rs
+++ b/src/main.rs
@@ -1,3 +1,3 @@
fn main() {
-    println!("Hello, world!");
+    println!("Hello, Git!");
}
```

-p では行単位の相違が表示されます、 --word-diff も指定するとワード単位の相違が表示されます。

```
PS > git log -p --word-diff --oneline
96b3dce add lib module
diff --git a/src/lib.rs b/src/lib.rs
new file mode 100644
index 0000000..3317b77
--- /dev/null
+++ b/src/lib.rs
@@ -0,0 +1,3 @@
{+pub fn hello() {+}
{+    println!("Hello, Git!");+}
{+}+}
diff --git a/src/main.rs b/src/main.rs
index 869abe6..7552ba0 100644
--- a/src/main.rs
+++ b/src/main.rs
@@ -1,3 +1,5 @@
{+use re_git::hello;+}

fn main() {
    [-println!("Hello, Git!");-]{+hello();+}
}

096663c second commit
diff --git a/src/main.rs b/src/main.rs
index e7a11a9..869abe6 100644
--- a/src/main.rs
+++ b/src/main.rs
@@ -1,3 +1,3 @@
fn main() {
    println!("Hello, [-world!];-]{+Git!");+
}

fdc9308 first commit
diff --git a/.gitignore b/.gitignore
new file mode 100644
index 0000000..ea8c4bf
--- /dev/null
+++ b/.gitignore
@@ -0,0 +1 @@
{+/target+}
diff --git a/Cargo.lock b/Cargo.lock
new file mode 100644
index 0000000..e4e9e8b
--- /dev/null
+++ b/Cargo.lock
@@ -0,0 +1,7 @@
{+# This file is automatically @generated by Cargo.+}
{+# It is not intended for manual editing.+}
{+version = 3+}

{+[[package]]+}
{+name = "re-git"+}
```

```
{+version = "0.1.0"+}
diff --git a/Cargo.toml b/Cargo.toml
new file mode 100644
index 0000000..2cf315e
--- /dev/null
+++ b/Cargo.toml
@@ -0,0 +1,8 @@
{+[package]+}
{+name = "re-git"+}
{+version = "0.1.0"+}
{+edition = "2021"+}

{+# See more keys and their definitions at https://doc.rust-lang.org/cargo/reference/manifest.html+}

{+[dependencies]+}
diff --git a/src/main.rs b/src/main.rs
new file mode 100644
index 0000000..e7a11a9
--- /dev/null
+++ b/src/main.rs
@@ -0,0 +1,3 @@
{+fn main() {+
{+    println!("Hello, world!");+
{+}+}
```

統計情報

--stat を指定すると変更箇所の統計情報を出力します。

```
PS > git log --stat --oneline
96b3dce add lib module
src/lib.rs | 3 +++
src/main.rs | 4 +++
2 files changed, 6 insertions(+), 1 deletion(-)

096663c second commit
src/main.rs | 2 ++
1 file changed, 1 insertion(+), 1 deletion(-)

fdc9308 first commit
.gitignore | 1 +
Cargo.lock | 7 ++++++++
Cargo.toml | 8 ++++++++
src/main.rs | 3 +++
4 files changed, 19 insertions(+)
```

➡ アスキーグラフ

--graph でブランチやマージをアスキーグラフで出力します。

```
PS > git log --graph --pretty=format:"%h %s"  
* 96b3dce add lib module  
* 096663c second commit  
* fdc9308 first commit
```

➡ ファイルの一覧

--name-only で変更のあったファイルの一覧を出力します。

```
PS > git log --name-only --oneline  
96b3dce add lib module  
src/lib.rs  
src/main.rs  
096663c second commit  
src/main.rs  
fdc9308 first commit  
.gitignore  
Cargo.lock  
Cargo.toml  
src/main.rs
```

➡ コミット時のファイルの状態

--name-status は名前と一緒にファイルの状態を出力します。ファイル状態の出力は git status -s と似ています。

```
PS > git log --name-status --oneline  
96b3dce add lib module  
A      src/lib.rs  
M      src/main.rs  
096663c second commit  
M      src/main.rs  
fdc9308 first commit  
A      .gitignore  
A      Cargo.lock  
A      Cargo.toml  
A      src/main.rs
```

④ 日付

--date で日付を表示します。--pretty と一緒に使います。

```
PS > git log --date=short --pretty=format:"%h %cd %cn %s"
96b3dce 2023-02-14 mebiusbox add lib module
096663c 2023-02-14 mebiusbox second commit
fdc9308 2023-02-14 mebiusbox first commit
```

⑤ 参照

--decorate でブランチやタグなどの参照を出力します。デフォルトは auto になっています。また、--decorate は --decorate=short の省略形です。

```
PS > git log --decorate=full --oneline
96b3dce (HEAD -> refs/heads/main) add lib module
096663c second commit
fdc9308 first commit
```

--all ですべての参照を出力します。通常は現在のブランチのコミットログのみです。

```
PS > git log --decorate --all --oneline
96b3dce (HEAD -> main) add lib module
096663c second commit
fdc9308 first commit
```

特定のブランチを除外したい場合、--not を使います。

```
PS > git log --decorate --all --not origin/gh-pages --oneline
```

⑥ その他

-since, -after

指定した日付より後のコミット

-until, -before

指定した日付より前のコミット

-author, -committer

指定したauthorまたはcommitterに一致したコミット

-grep

指定した文字列がメッセージに含まれるコミット

-g (-walk-reflog), -reflog

「基本」を参照してください。

➤ delta

差分表示ツール delta を使うとログが見やすくなります。 .gitconfig で core.pager に delta を指定します。

```
PS > git config --global core.pager delta
```

-p を指定してログを出力します。

```
PS > git log -p --oneline
```

```
96b3dce (HEAD → main) add lib module

src/lib.rs:1:
1 | pub fn hello() {
2 |     println!("Hello, Git!");
3 | }

src/main.rs:1:
1 | use re-git::hello;
2 |
3 | fn main() {
4 |     println!("Hello, Git!");
5 |     hello();
6 |
7 | }

096663c second commit
```

-c delta.side-by-side=true を指定すると2画面で出力します。

```
PS > git -c delta.side-by-side=true -p --oneline
```

```
96b3dce (HEAD -> main) add lib module

src/lib.rs:1:
| |
| |
| | 1 pub fn hello() {
| | 2     println!("Hello, Git!");
| | 3 }

src/main.rs:1:
| |
| |
| | 1 fn main() {
| | 2     println!("Hello, Git!");
| | 3 }

096663c second commit
```

詳細はドキュメントを参照してください。

■ ブランチ

ブランチはコミットを指すポインタです。ブランチごとにコミットすることができ、各ブランチが独立して開発を進めることができます。また、別のブランチからコミットを取り込むこともできます。それはマージやりベース、チェリーピックなどを使用します。

ブランチごとに役割を持たせて開発を進めていくのが一般的です。各ブランチがどのようなものか、そしてどのように連携させていくのかをパターン化したものをGitワークフローともいいます。例えば、Git flow, GitHub flowなどがあります。個人的にはベースとなるワークフローを選んで、環境に合わせてカスタマイズしたものを使えばいいと思います。

大まかに、main/releaseブランチ、developブランチ、topicブランチの3つのブランチを軸に運用するとシンプルにまとまっていい感じです。topicブランチは短期的もしくは最も単位の小さい作業ブランチです。課題(issue)、機能の追加(feature)、不具合修正(hotfix)などで使います。

▷ ブランチの一覧

git branch コマンドを使います。

```
PS(main) > git branch
  develop
* main
```

-v を指定すると各ブランチの直近コミットを確認できます。

```
PS > git branch -v
  develop c5e8b09 fixup! fixup! refactor the main function
* main    0d22a3f add README
```

--merged や --no-merged を指定すると、現在のブランチにマージ済みなもの、マージ済みでないブランチを確認できます。

```
PS > git branch --merged
* main
```

```
PS > git branch --no-merged
  develop
```

▷ ブランチの作成

現在のブランチから別のブランチを作成するには次のようにします。

```
PS > git branch <name>
```

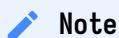
例えば、issue ブランチを作成する場合、次のようにします。

```
PS(main) > git branch issue
```

▷ ブランチの切り替え

ブランチを切り替えるには `git switch` コマンドを使います。

```
PS(main) > git switch develop
```



Note

ブランチの切り替えは `git checkout` コマンドでも可能です。 `git checkout` は多くの機能があることから、ブランチの切り替え機能を `git switch` コマンドとして使えるようになりました。

ブランチを作成すると同時に、作成したブランチに切り替える場合、`-c` を指定します。

```
PS(main) > git switch -c develop
```



Note

`HEAD`は現在のブランチを指し、ブランチは**コミット**を指します。

▷ ログにブランチを表示

`git log` コマンドに `--decorate` をつけます。

```
PS > git log --decorate --oneline
0d22a3f (HEAD -> main, origin/main, origin/HEAD, hoge/main) add README
77da8c1 add poem.txt
228d01e Listing 12-2
88045a6 Reading the Argument Values
567c4c6 add Cargo.lock
3142e1b first commit
```

▷ ブランチの削除

`git branch` コマンドに `-d` を指定します。 例えば、issue ブランチを削除する場合、次のようになります。

```
PS > git branch -d issue
```

削除したいブランチが現在のブランチにマージされていない場合、削除に失敗します。強制的に削除したい場合、`-d` の代わりに `-D` を指定します。

```
PS > git branch -D issue
```

➡ リモート参照

リモートリポジトリにある参照のことをリモート参照といいます。リモート参照を確認するには `git ls-remote` コマンドを使います。

```
PS > git ls-remote
0d22a3f04e0c11a5be123e36c34ac545dbd303d2      HEAD
c5e8b097c3eaba935d134fe21efd7d0115dc9a7d      refs/heads/develop
0d22a3f04e0c11a5be123e36c34ac545dbd303d2      refs/heads/main
```

また、リポジトリのURLを直接指定することもできます。

```
PS > git ls-remote https://github.com/mebiusbox/re-git-sample.git
0d22a3f04e0c11a5be123e36c34ac545dbd303d2      HEAD
c5e8b097c3eaba935d134fe21efd7d0115dc9a7d      refs/heads/develop
0d22a3f04e0c11a5be123e36c34ac545dbd303d2      refs/heads/main
```

➡ 追跡ブランチ

リモート参照のうち、ブランチのことをリモートブランチといいます。このリモートブランチを追跡するブランチのことをリモート追跡ブランチといいます。リモート追跡ブランチを確認するには `git branch` コマンドに `--remote` または `-r` を指定します。

```
PS > git branch -r
hoge/develop
hoge/main
origin/HEAD -> origin/main
origin/develop
origin/main
```

リモート追跡ブランチの名前は `<remote>/<branch>` となります。すでに述べたように、リモート追跡ブランチはリモートブランチを追跡しているもので、リモートブランチに切り替えることはできません。その場合、リモートブランチを追跡するローカルなブランチを作成します。このようなブランチを追跡ブランチといいます。

Note

リモート追跡ブランチおよび追跡ブランチはローカルに作成される参照です。リモート追跡ブランチは、ローカルで参照するために、リモートブランチに名前をつけたものと考えるといいかかもしれません。これは、`.git/refs/remotes` ディレクトリにキャッシュされます。追跡ブランチは、ローカルブランチ（`.git/refs/heads` ディレクトリにブランチのHEAD情報がある）がリモート追跡ブランチと紐づいているものです。

追跡ブランチに紐づいているリモート追跡ブランチのことを **上流ブランチ(upstream branch)** といいます。リモート追跡ブランチから追跡ブランチを作成するには `git switch` コマンドを使います。

```
PS > git switch -c develop origin/develop
```

または、`--track`（`-t`）を使います。

```
PS > git switch -t origin/develop
```

ブランチの上流ブランチを確認したい場合、`git branch` コマンドに `-vv` を指定します。

```
PS > git branch -vv
  develop c5e8b09 [origin/develop] fixup! fixup! refactor the main function
* main      53b97a9 [origin/main: ahead 1] test
```

Note

リモートブランチを削除する場合、次のようになります。

```
PS > git push <remote> --delete <name>
```

マージ

別のブランチから現在のブランチにマージする場合、`git merge` コマンドを使います。例えば、`main`ブランチに `develop`ブランチをマージしたい場合、次のようになります。

```
PS(main) > git merge develop
```

マージには `Fast-Forward` や `3-ways` の戦略があります。詳しくはドキュメントなどを参照してください。マージの衝突の対応については割愛。

Note

まとめられたらいつか書くかも。

➤ チェリーピック

別のブランチにある一部のコミットだけを現在のブランチにマージしたい場合、チェリーピック(cherry-pick)を使います。例えば、mainブランチとdevelopブランチの2つで開発を進めていたとします。mainブランチはマイルストーンごとにdevelopブランチからマージしていました。ここで、developブランチで作業中に重大な不具合が見つかったとしましょう。不具合を修正してコミットしました。しかし、すでにdevelopブランチにはいくつかのコミットがあって、まだmainブランチにマージしたくありません。そこで、チェリーピックを使って不具合を修正したコミットだけをmainブランチにマージします。もちろん、この問題については別のアプローチもあるでしょう。

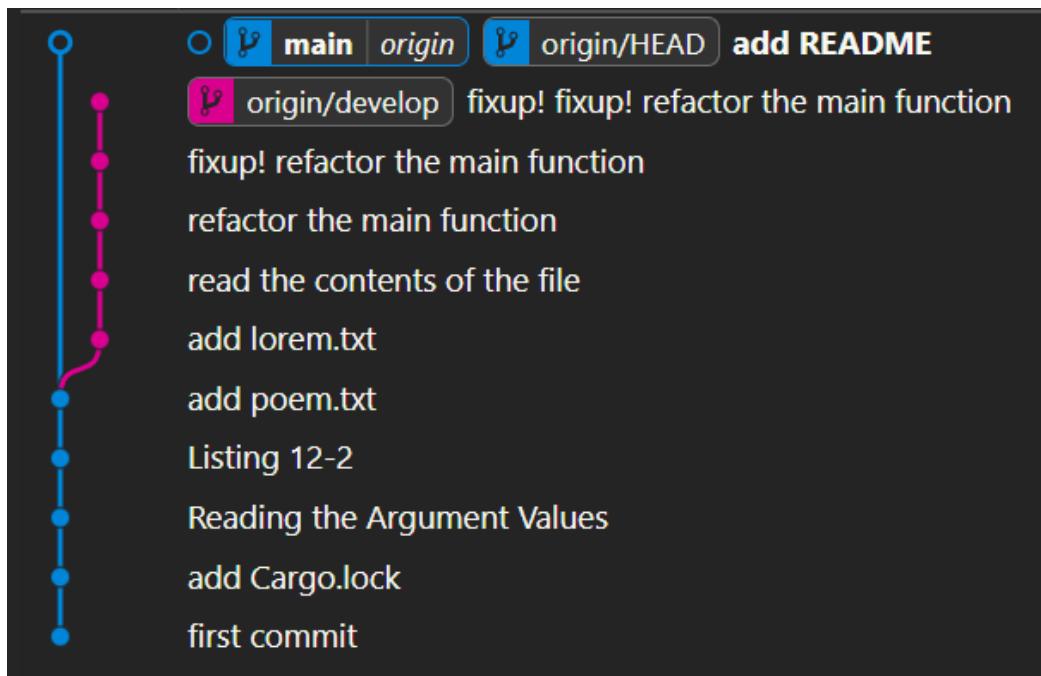
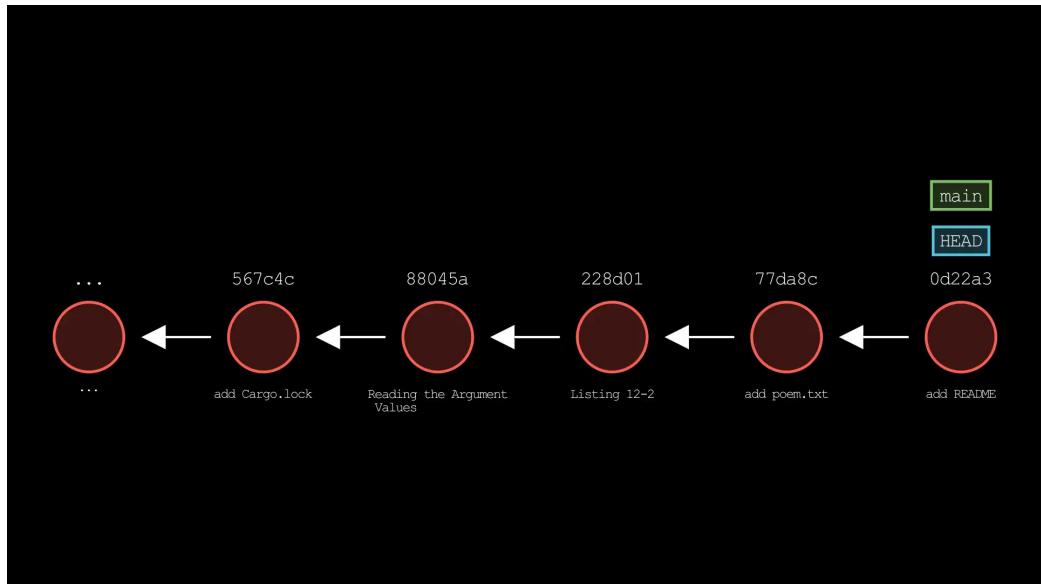
では、チェリーピックを試してみましょう。用意したサンプルプロジェクトをクローンします。

```
PS > git clone https://github.com/mebiusbox/re-git-sample.git
Cloning into 're-git-sample'...
remote: Enumerating objects: 43, done.
remote: Counting objects: 100% (43/43), done.
remote: Compressing objects: 100% (23/23), done.
remote: Total 43 (delta 15), reused 42 (delta 14), pack-reused 0
Receiving objects: 100% (43/43), 4.83 KiB | 4.83 MiB/s, done.
Resolving deltas: 100% (15/15), done.
```

ログを確認します。

```
PS(main) > git log --oneline --decorate --all
0d22a3f (HEAD -> main, origin/main, origin/HEAD) add README
c5e8b09 (origin/develop) fixup! fixup! refactor the main function
16cfdd7 fixup! refactor the main function
9cc0d7b refactor the main function
e386762 read the contents of the file
b3ed661 add lorem.txt
77da8c1 add poem.txt
228d01e Listing 12-2
88045a6 Reading the Argument Values
567c4c6 add Cargo.lock
3142e1b first commit
```

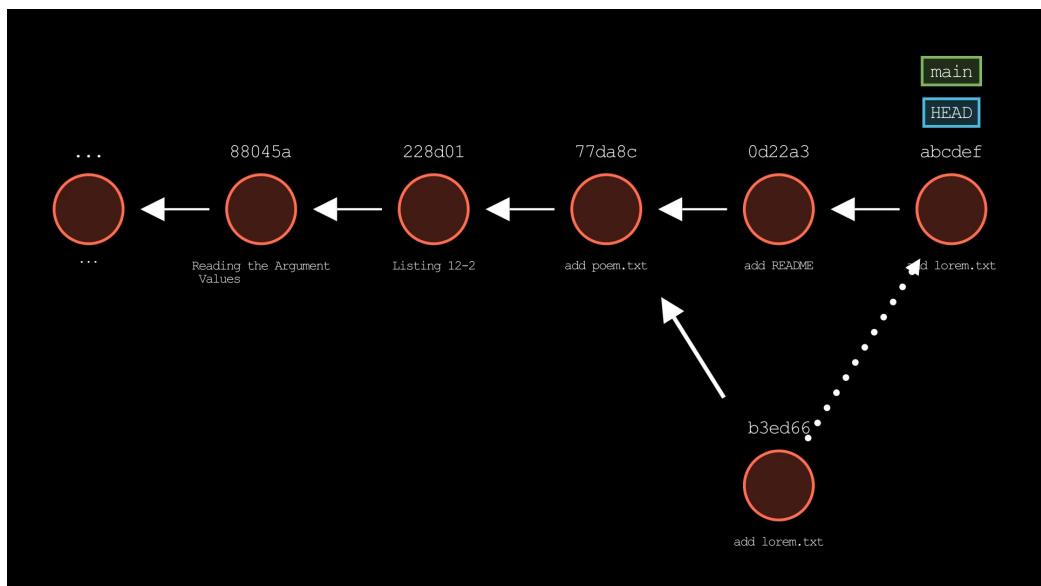
コミットグラフは次のようになっています。



それでは、developブランチにある `b3ed661 add lorem.txt` コミットをmainブランチにチェリーピックしてみましょ。う。

```
PS(main) > git cherry-pick b3ed661
[main 8346cd7] add lorem.txt
Date: Wed Feb 15 16:59:45 2023 +0900
1 file changed, 1 insertion(+)
create mode 100644 lorem.txt
```

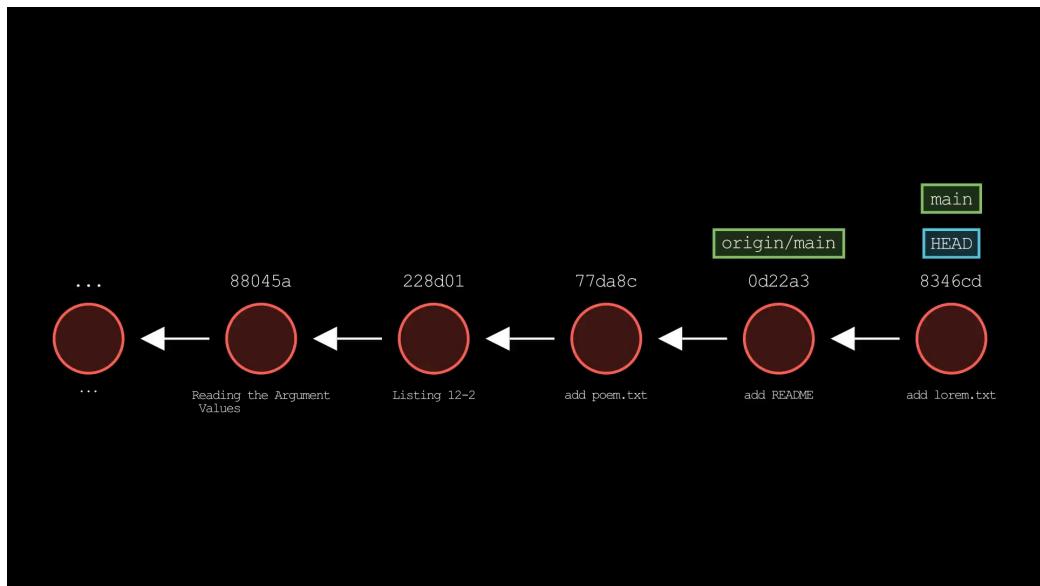
0:00 / 0:09



ログを確認してみます。

```
PS(main) > git log --oneline --decorate --all
8346cd7 (HEAD -> main) add lorem.txt
0d22a3f (origin/main, origin/HEAD) add README
c5e8b09 (origin/develop) fixup! fixup! refactor the main function
16cfdd7 fixup! refactor the main function
9cc0d7b refactor the main function
e386762 read the contents of the file
b3ed661 add lorem.txt
77da8c1 add poem.txt
228d01e Listing 12-2
88045a6 Reading the Argument Values
567c4c6 add Cargo.lock
3142e1b first commit
```

コミットグラフは次のようになっています。



特定のコミットがマージされていますね。

■ タグ

タグもHEADやブランチと同様に参照の1つです。ですから、コミットを参照します。ブランチは常にコミットグラフの先頭を指しますが、タグは違います。コミットのブックマークと考えられますね。タグには2種類あります。軽量タグ(lightweight)と、注釈付きタグ(annotated)です。注釈付きタグはメッセージを書くことができます。

➤ タグの確認

git tag コマンドを使います。

```
PS > git tag
v0.1
v0.2
```

このとき、ログは次のようになっています。

```
PS > git log --oneline --graph --decorate --all
* 0d22a3f (HEAD -> main, tag: v0.2, origin/main, origin/HEAD) add README
| * c5e8b09 (origin/develop) fixup! fixup! refactor the main function
| * 16cfdd7 fixup! refactor the main function
| * 9cc0d7b refactor the main function
| * e386762 read the contents of the file
| * b3ed661 add lorem.txt
|/
* 77da8c1 (tag: v0.1) add poem.txt
* 228d01e Listing 12-2
* 88045a6 Reading the Argument Values
* 567c4c6 add Cargo.lock
* 3142e1b first commit
```

コミットグラフは次のようになっています。



タグの詳細を確認する場合、`git show` コマンドを使います。

```
PS > git show v0.2
tag v0.2
Tagger: mebiusbox <mebiusbox@gmail.com>
Date: Thu Feb 16 17:11:37 2023 +0900

annotated tag

...
```

参照ですから、`git rev-parse` コマンドで参照しているコミットを確認できます。

```
PS > git rev-parse v0.1
77da8c11a1aca8b1c27d18bac1666142b62117ee
```

➤ タグの作成

`git tag` コマンドにタグ名を指定すると、現在のHEADに軽量タグを作成できます。

```
PS > git tag v0.1
```

HEADではなく任意のコミットに軽量タグを作成する場合、タグ名のあとにコミットを特定できるもの(commit-ish)を指定します。

```
PS > git tag v0.1 77da8c1
```

注釈付きタグを作成する場合、`git tag` コマンドに `-a` を指定します。

```
PS > git tag -a v0.1 77da8c1
```

メッセージを追加する場合、`-m` を指定します。

```
PS > git tag -a v0.1 -m "message" 77da8c1
```

➤ タグの削除

`git tag` コマンドに `-d` を指定します。

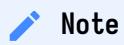
```
PS > git tag -d v0.1
```

➡ タグからブランチを作成

git switch コマンドでタグからブランチを作成できます。

```
PS > git switch -c v0.1 v0.1
```

➡ タグのプッシュ



Note

「リモートリポジトリ」を参照してください

リベース

リベースはコミットを並び替えたり、コミットの内容を修正したり、まとめることができます。つまり、コミットの履歴を書き換えます。リベースは別のブランチにあるコミットを現在のブランチに再配置してマージのような処理もできます。

リベースの処理を確認するために、Rustの公式ドキュメントにある入出力プロジェクトのサンプルで話を進めていきます。

<https://doc.rust-lang.org/book/ch12-00-an-io-project.html>

プロジェクト作成

適当なフォルダを作成して、`cargo init` をするか、`cargo new` で新規プロジェクトを作成します。

```
PS > cargo new minigrep  
PS > cd minigrep
```

フックフォルダは削除しておきます。

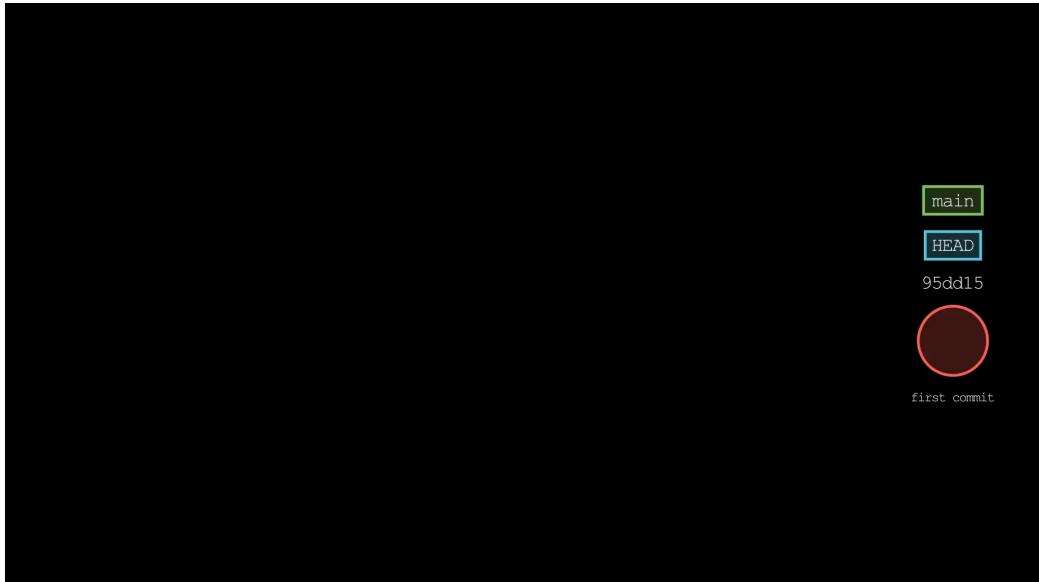
```
PS > rm -Force .git\hooks
```

この時点の作業ツリーは次のようになっています。

```
PS > ls  
.  
├── .git  
│   ├── info  
│   │   └── exclude  
│   ├── objects  
│   │   ├── info  
│   │   └── pack  
│   ├── refs  
│   │   ├── heads  
│   │   └── tags  
│   ├── config  
│   ├── description  
│   └── HEAD  
└── src  
    └── main.rs  
└── .gitignore  
└── Cargo.toml
```

最初のコミットをしましょう。

```
PS > git add .
PS > git commit -m "first commit"
PS > git log --oneline
95dd157 (HEAD -> main) first commit
```



プロジェクトをビルドして実行します。

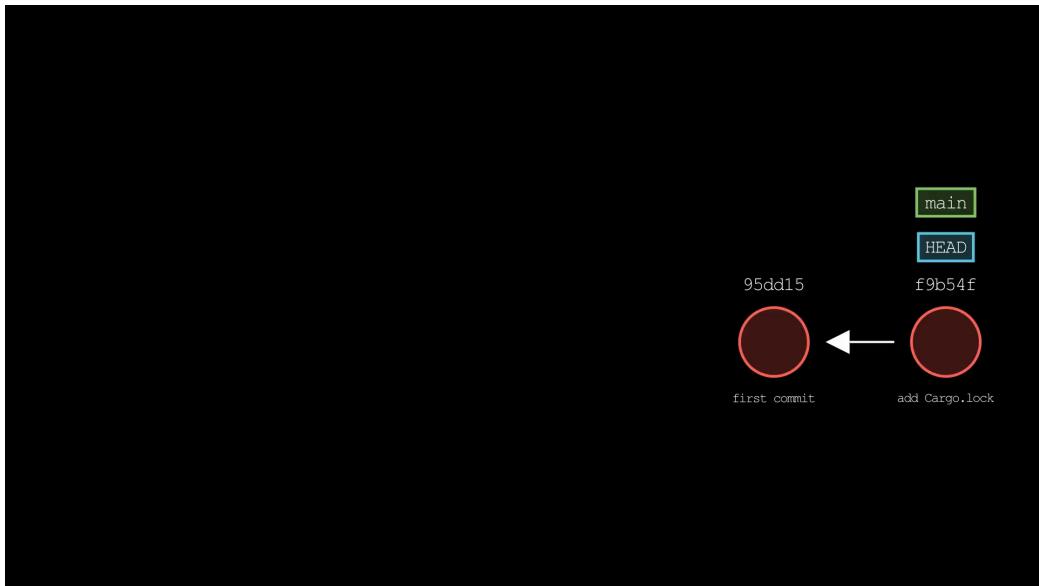
```
PS(main) > cargo run
Compiling minigrep v0.1.0 (F:\minigrep)
    Finished dev [unoptimized + debuginfo] target(s) in 1.14s
        Running `target\debug\minigrep.exe`
Hello, world!
```

Cargo.lock ファイルが作成されます。

```
PS(main) > git status -s
?? Cargo.lock
```

追加して、コミットしましょう。

```
PS(main) > git add .
PS(main) > git commit -m "add Cargo.lock"
[main d10db36] add Cargo.lock
 1 file changed, 7 insertions(+)
 create mode 100644 Cargo.lock
```



続いて、src/main.rs を編集します。

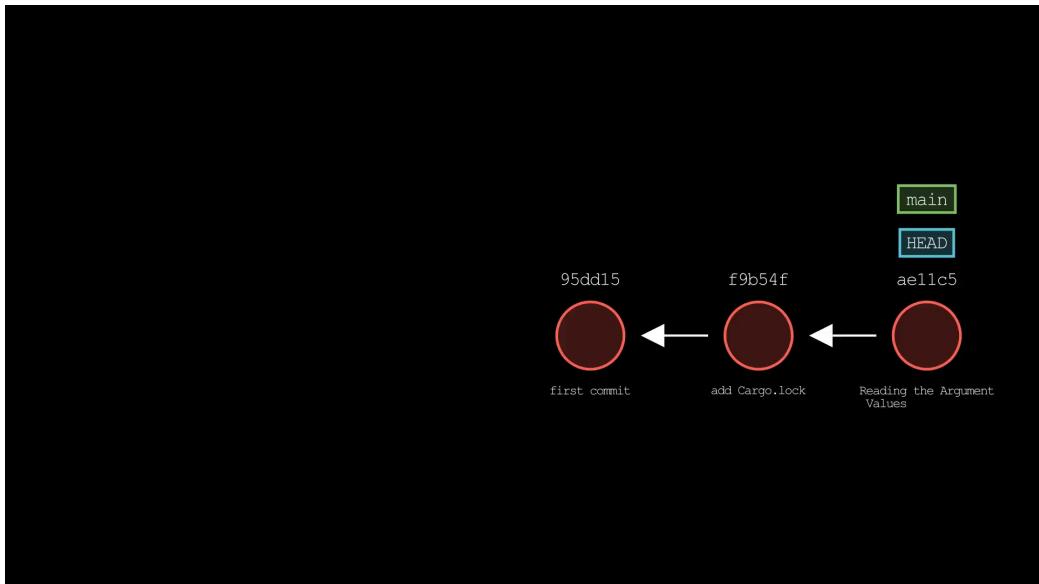
```
use std::env;

fn main() {
    let args: Vec<String> = env::args().collect();
    dbg!(args);
}
```

動作確認してコミットします。

```
PS(main) > cargo run
Compiling minigrep v0.1.0 (F:\minigrep)
  Finished dev [unoptimized + debuginfo] target(s) in 0.36s
    Running `target\debug\minigrep.exe`
[src\main.rs:5] args = [
  "target\\debug\\minigrep.exe",
]

PS(main) > git add -u
PS(main) > git commit -m "Reading the Argument Values"
[main ae11c5e] Reading the Argument Values
  1 file changed, 4 insertions(+), 1 deletion(-)
```



もう一度 `src/main.rs` を編集してコミットします。

```
use std::env;

fn main() {
    let args: Vec<String> = env::args().collect();

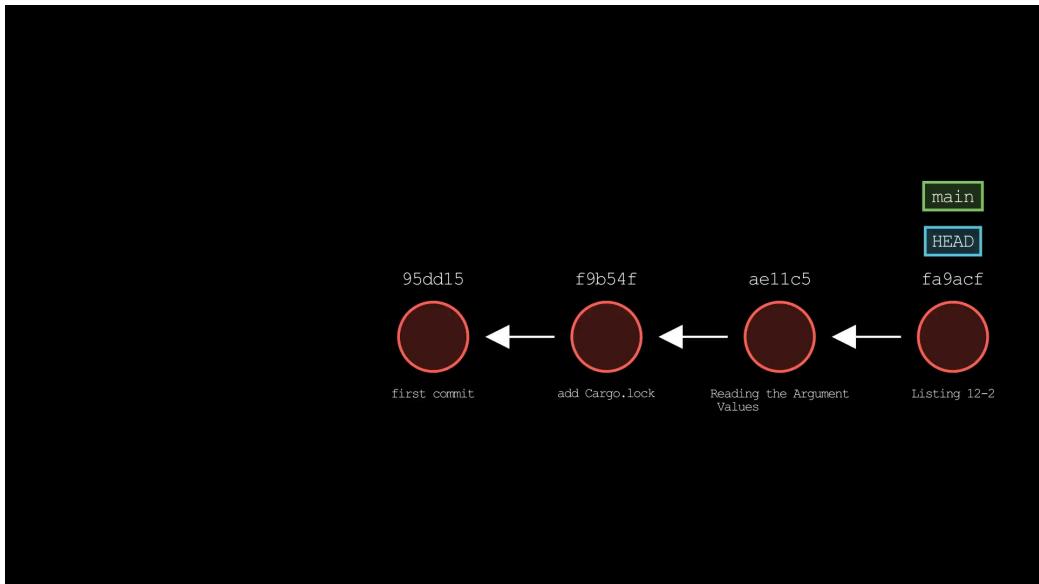
    let query = &args[1];
    let file_path = &args[2];

    println!("Searching for {}", query);
    println!("In file {}", file_path);
}
```

また、動作確認してコミットします。

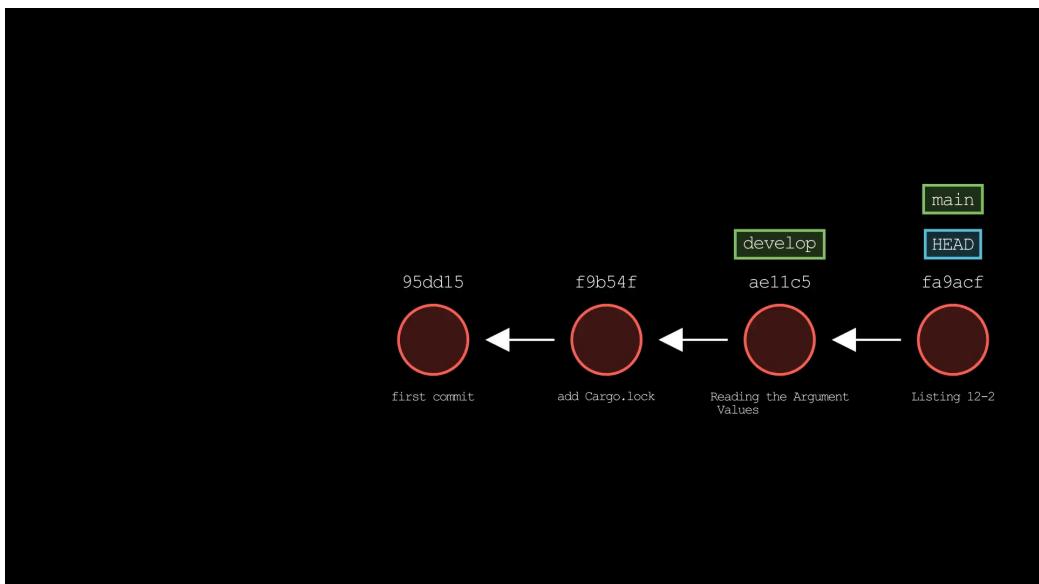
```
PS(main) > cargo run -- test sample.txt
Compiling minigrep v0.1.0 (F:\minigrep)
Finished dev [unoptimized + debuginfo] target(s) in 0.43s
Running `target\debug\minigrep.exe test sample.txt`
Searching for test
In file sample.txt

PS(main) > git add -u
PS(main) > git commit -m "Listing 12-2"
[main fa9acf8] Listing 12-2
1 file changed, 6 insertions(+), 1 deletion(-)
```



ここで、1つ前のコミットから分岐します。ブランチ名は `develop` とします。

```
PS(main) > git switch -c develop HEAD~  
Switched to a new branch 'develop'
```



`develop` ブランチで `poem.txt` ファイルを追加してコミットします。

```
I'm nobody! Who are you?  
Are you nobody, too?  
Then there's a pair of us - don't tell!  
They'd banish us, you know.
```

```
How dreary to be somebody!  
How public, like a frog  
To tell your name the livelong day  
To an admiring bog!
```

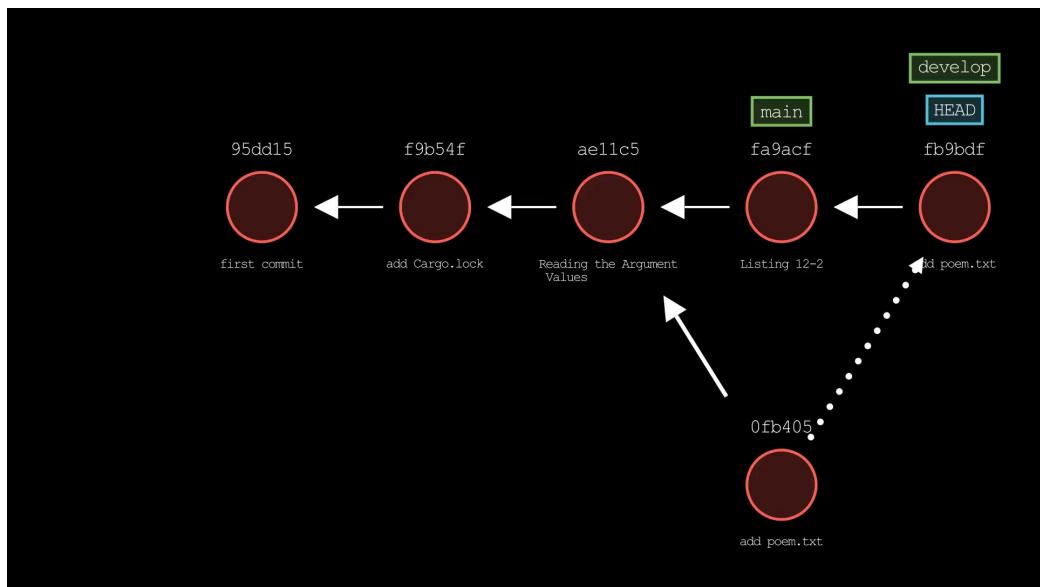
```
PS(develop) > git add poem.txt
PS(develop) > git commit -m "add poem.txt"
[develop 0fb4052] add poem.txt
 1 file changed, 9 insertions(+)
 create mode 100644 poem.txt
```

④ リベース

ここで、リベースを実行して先ほどコミットした内容を、mainブランチの後ろに並びかえてみます。

```
PS(develop) > git rebase main
Successfully rebased and updated refs/heads/develop.
```

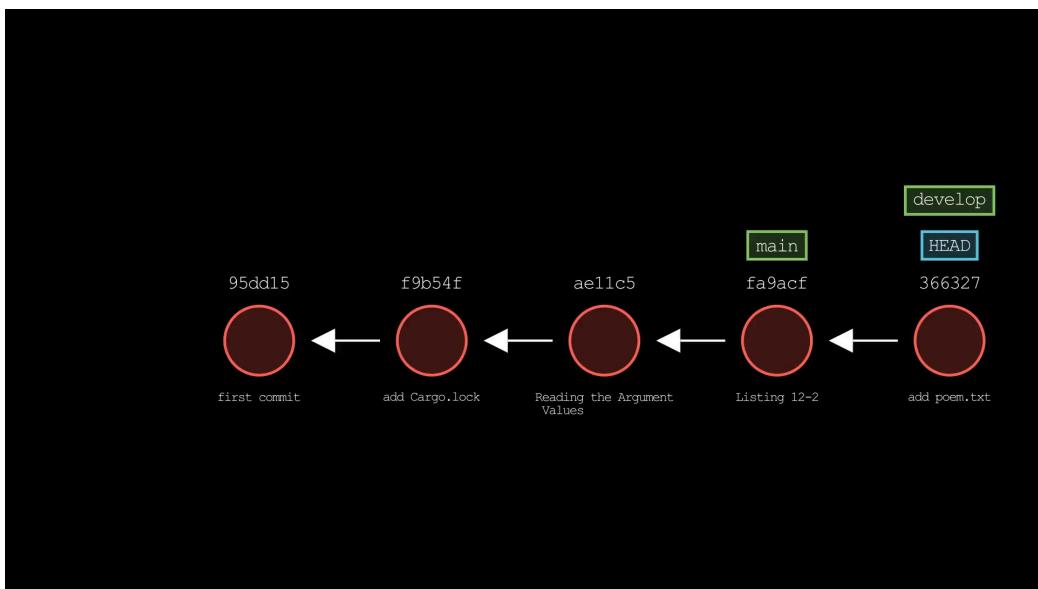
リベースをすると次のようなグラフになります。



アニメーションで見ると次のようにになります

0:00 / 0:09

リベース後には次のようにグラフが一本化します。

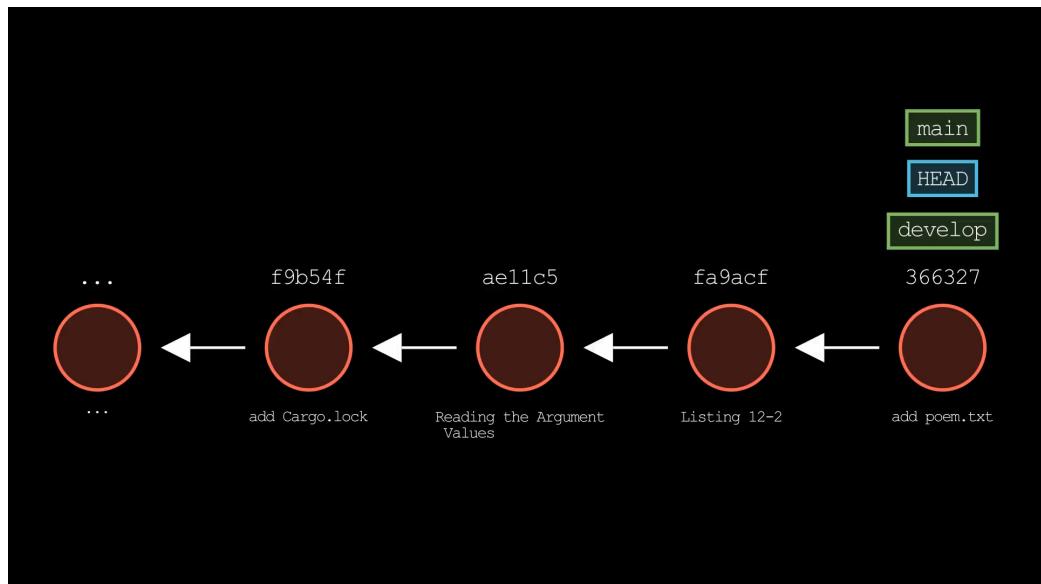


このようなグラフになると、`main` ブランチで `develop` ブランチをマージするときに、Fast-Forwardになります。これがプルリクエスト前にリベースする理由の1つです。

```
PS(develop) > git switch main
Switched to branch 'main'

PS(main) > git merge develop
Updating fa9acf8..366327b
Fast-forward
 poem.txt | 9 ++++++++
 1 file changed, 9 insertions(+)
 create mode 100644 poem.txt
```

0:00 / 0:07



もし、リベースせずに main ブランチが develop ブランチをマージしようとすると、マージコミットが作成されます。試してみましょう。まず、main ブランチで行ったマージを戻します。

```
PS(main) > git reset --hard HEAD~
```

次に、develop ブランチに変えて、リベースを戻します。リベースを戻すには reflog を使います。 git reflog を実行して rebase 実行直前のログを探します。

```
PS(main) > git switch develop
PS(develop) > git reflog

...
9ac6088 (main) HEAD@{6}: rebase (start): checkout main
b242a1f HEAD@{7}: commit: add poem.txt
...
```

ここでは `HEAD@{7}` なので `reset` に指定します。

```
PS(develop) > git reset --hard "HEAD@{7}"
```

これでリベース前に戻りました。

Note

`git reset` は現在のブランチにしか影響しません。

Note

PowerShellの場合 `@{...}` は特別な意味を持つため、クオーテーションで囲む必要があります。

`main` ブランチに変えてマージしてみます。

```
PS(develop) > git switch main
PS(main) > git merge develop
```

0:00 / 0:09

このように余計なマージコミットが作成されます。プルリクエストする際はなるべくリベースしておきましょう。

➤ コミットメッセージの書き換え

また、何回かコミットします。作業はdevelopブランチで行います。

```
PS(main) > git switch develop
```

コミット(1回目)

src/main.rs を次のように書き換えます。

```
use std::env;
use std::fs;

fn main() {
    // --snip--
    let args: Vec<String> = env::args().collect();

    let query = &args[1];
    let file_path = &args[2];

    println!("Searching for {}", query);
    println!("In file {}", file_path);

    let contents = fs::read_to_string(file_path)
        .expect("Should have been able to read the file");

    println!("With text:\n{}", contents);
}
```

動作確認してコミットします。

```
PS(develop) > cargo run -- the poem.txt
Compiling minigrep v0.1.0 (F:\minigrep)
  Finished dev [unoptimized + debuginfo] target(s) in 0.79s
    Running `target\debug\minigrep.exe test poem.txt`
Searching for test
In file poem.txt
With text:
I'm nobody! Who are you?
Are you nobody, too?
Then there's a pair of us - don't tell!
They'd banish us, you know.

How dreary to be somebody!
How public, like a frog
To tell your name the livelong day
To an admiring bog!
```

```
PS > git add -
PS > git commit -m "Listing 12-4"
[develop 262cd2f] Listing 12-4
 1 file changed, 6 insertions(+)
```

コミット(2回目)

src/main.rs を書き換えてコミットします。

```

use std::env;
use std::fs;

fn main() {
    let args: Vec<String> = env::args().collect();

    let (query, file_path) = parse_config(&args);

    // --snip--

    println!("Searching for {}", query);
    println!("In file {}", file_path);

    let contents = fs::read_to_string(file_path)
        .expect("Should have been able to read the file");

    println!("With text:\n{}", contents);
}

fn parse_config(args: &[String]) -> (&str, &str) {
    let query = &args[1];
    let file_path = &args[2];

    (query, file_path)
}

```

```

PS(develop) > cargo run -- the poem.txt
Compiling minigrep v0.1.0 (F:\minigrep)
    Finished dev [unoptimized + debuginfo] target(s) in 0.39s
        Running `target\debug\minigrep.exe the poem.txt`
Searching for the
In file poem.txt
With the:
I'm nobody! Who are you?
Are you nobody, too?
Then there's a pair of us - don't tell!
They'd banish us, you know.

How dreary to be somebody!
How public, like a frog
To tell your name the livelong day
To an admiring bog!

```

```

PS(develop) > git add -
PS(develop) > git commit -m "Listing 12-5"
[develop fd7a7a5] Listing 12-5
  1 file changed, 8 insertions(+), 2 deletions(-)

```

コミット(3回目)

src/main.rs を書き換えてコミットします。

```
use std::env;
use std::fs;

fn main() {
    let args: Vec<String> = env::args().collect();

    let config = parse_config(&args);

    println!("Searching for {}", config.query);
    println!("In file {}", config.file_path);

    let contents = fs::read_to_string(config.file_path)
        .expect("Should have been able to read the file");

    // --snip--

    println!("With text:\n{}\n{}", contents);
}

struct Config {
    query: String,
    file_path: String,
}

fn parse_config(args: &[String]) -> Config {
    let query = args[1].clone();
    let file_path = args[2].clone();

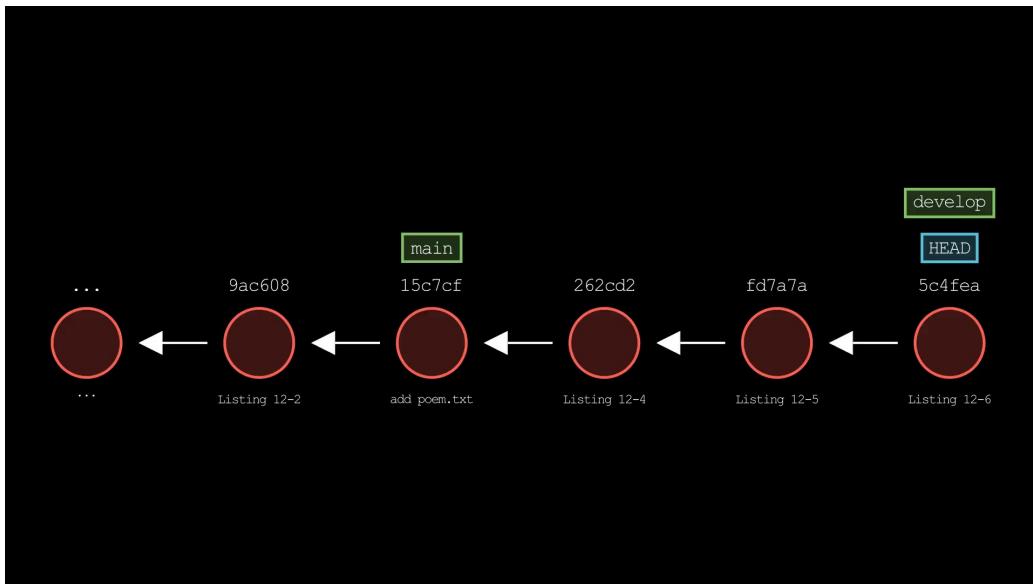
    Config { query, file_path }
}
```

```
PS(develop) > cargo run -- the poem.txt
Compiling minigrep v0.1.0 (F:\minigrep)
    Finished dev [unoptimized + debuginfo] target(s) in 0.37s
        Running `target\debug\minigrep.exe the poem.txt`
Searching for the
In file poem.txt
With text:
I'm nobody! Who are you?
Are you nobody, too?
Then there's a pair of us - don't tell!
They'd banish us, you know.

How dreary to be somebody!
How public, like a frog
To tell your name the livelong day
To an admiring bog!
```

```
PS(develop) > git add -u
PS(develop) > git commit -m "Listing 12-6"
[develop 5c4fea] Listing 12-6
1 file changed, 14 insertions(+), 8 deletions(-)
```

この時点でコミットグラフは次のようになっています。



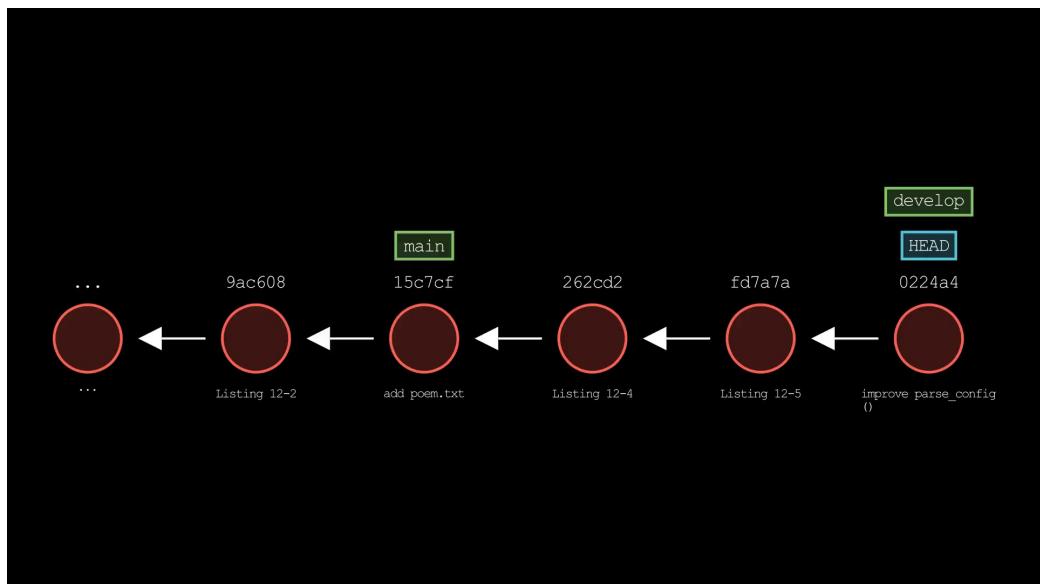
直前のコミットメッセージを修正

直前のコミットメッセージを修正するにはいくつか方法があります。まず、直前のコミットを取り消して、再度コミットする方法です。コミットの取り消しは `git reset` を使います。

```
PS(develop) > git reset --soft HEAD~
```

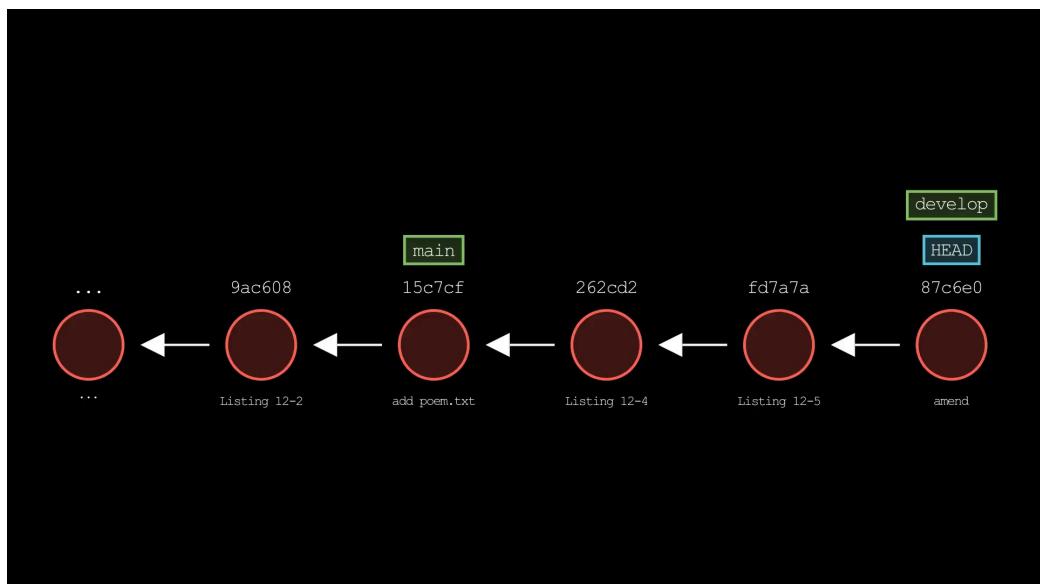
`git reset` については別の章で詳しく説明します。これで、コミットが1つ前に戻り、作業ツリーとインデックスはそのままになっています。この状態でメッセージを変えてコミットすると、コミットメッセージを書き換えたことになります。

```
PS(develop) > git commit -m "improve parse_config()  
[develop 0224a4c] improve parse_config()  
1 file changed, 14 insertions(+), 8 deletions(-)
```



他の方法として、git commit に --amend オプションを追加すると、最後にコミットしたメッセージを書き換えることができます。

```
PS(develop) > git commit --amend -m "amend"  
[develop 87c6e08] amend  
Date: Sat Feb 11 11:05:48 2023 +0900  
1 file changed, 14 insertions(+), 8 deletions(-)
```



リベースを使った書き換え

そして、リベースを使うと最後だけでなく、いくつもまとめて変更できます。たとえば、直近の3つコミットメッセージを変更したい場合、次のようにします。

```
PS(develop) > git rebase -i HEAD~3
```

-i は対話形式のオプションです。実行するとエディタが開きます。

```
pick 262cd2f Listing 12-4
pick fd7a7a5 Listing 12-5
pick 87c6e08 amend

# Rebase 15c7cf3..87c6e08 onto 15c7cf3 (3 commands)
#
# Commands:
# p, pick <commit> = use commit
# r, reword <commit> = use commit, but edit the commit message
# e, edit <commit> = use commit, but stop for amending
# s, squash <commit> = use commit, but meld into previous commit
# f, fixup [-C | -c] <commit> = like "squash" but keep only the previous
#                               commit's log message, unless -C is used, in which case
#                               keep only this commit's message; -c is same as -C but
#                               opens the editor
# x, exec <command> = run command (the rest of the line) using shell
# b, break = stop here (continue rebase later with 'git rebase --continue')
# d, drop <commit> = remove commit
# l, label <label> = label current HEAD with a name
# t, reset <label> = reset HEAD to a label
# m, merge [-C <commit> | -c <commit>] <label> [# <oneline>]
#       create a merge commit using the original merge commit's
#       message (or the oneline, if no original merge commit was
#       specified); use -c <commit> to reword the commit message
# u, update-ref <ref> = track a placeholder for the <ref> to be updated
#                      to this position in the new commits. The <ref> is
#                      updated at the end of the rebase
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
```

3つのコミットが並んでいます。この場合、3つのコミットが適用される前に巻き戻った後、上から順番にコミットが再度実行されます。ここでコミットメッセージを編集します。書き換えたいコミットメッセージの `pick` を `reword` に変更して、コミットメッセージを書き換えます。

```
reword 262cd2f Listing 12-4
reword fd7a7a5 Listing 12-5
reword 87c6e08 amend
...
```

保存すると、コミットが順番に実行され、 `reword` に設定したコミットの場合、コミットメッセージを編集します。それぞれ、次のように書き換えます

- Listing 12-4 → read the contents of the file
- Listing 12-5 → add parse_config()
- amend → improve parse_config()

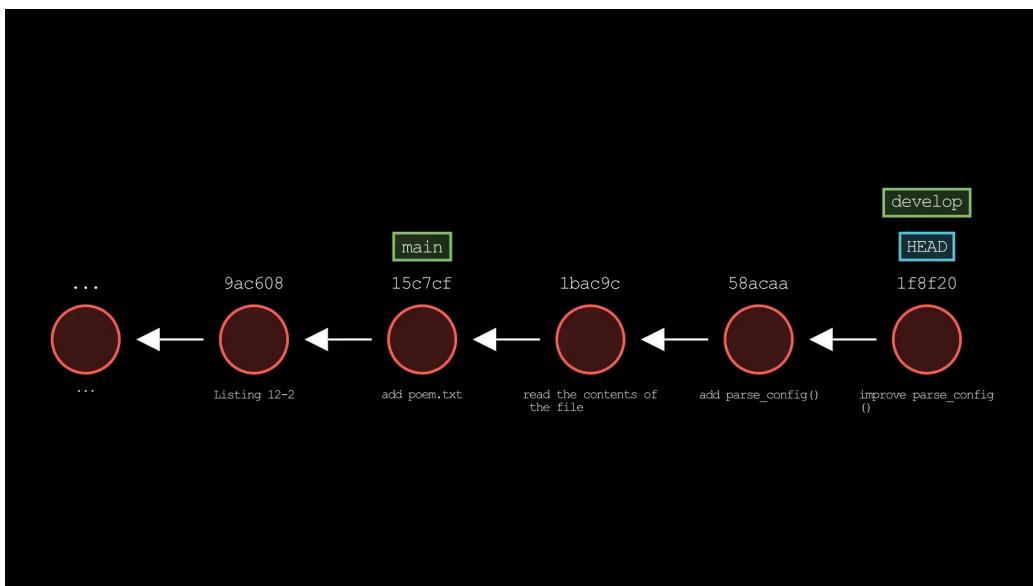
```
PS(develop) > git rebase -i HEAD~3
[detached HEAD 1bac9c2] read the contents of the file
  Date: Sat Feb 11 10:51:32 2023 +0900
  1 file changed, 6 insertions(+)

[detached HEAD 58acaa7] add parse_config()
  Date: Sat Feb 11 10:55:58 2023 +0900
  1 file changed, 8 insertions(+), 2 deletions(-)

[detached HEAD 1f8f204] improve parse_config()
  Date: Sat Feb 11 11:05:48 2023 +0900
  1 file changed, 14 insertions(+), 8 deletions(-)

Successfully rebased and updated refs/heads/develop.
```

この時点でのコミットグラフは次の通りです。



git log でコミットメッセージが書きわかっていることを確認しましょう。

```
PS(develop) > git log --oneline
1f8f204 (HEAD -> develop) improve parse_config()
58acaa7 add parse_config()
1bac9c2 read the contents of the file
15c7cf3 (main) add poem.txt
9ac6088 Listing 12-2
ae11c5e Reading the Argument Values
f9b54f3 add Cargo.lock
95dd157 first commit
```

rebaseのときに pick を reword ではなく、edit にしても同様のことができます。edit の場合、コミットされる度に、git commit --amend で書き換えられます。書き換えたら git rebase --continue でリベースの処理を続行します。この方法では、コミットの状態を確認しながらコミットメッセージを書き換えることができます。試してみましょう。まず、リベースを戻します。git reflog でリベース前のコミットを見つけます。

```
PS(develop) > git reflog
...
262cd2f HEAD@{6}: rebase: fast-forward
15c7cf3 (main) HEAD@{7}: rebase (start): checkout HEAD~3
87c6e08 HEAD@{8}: commit (amend): amend
...
```

ここでは、HEAD@{8} の位置なので、そのコミットまで戻します。

```
PS(develop) > git reset --hard "HEAD@{8}"
HEAD is now at 87c6e08 amend

PS(develop) > git log --oneline
87c6e08 (HEAD -> develop) amend
fd7a7a5 Listing 12-5
262cd2f Listing 12-4
15c7cf3 (main) add poem.txt
9ac6088 Listing 12-2
ae11c5e Reading the Argument Values
f9b54f3 add Cargo.lock
95dd157 first commit
```

前回同様にリベースを実行します。

```
PS(develop) > git rebase -i HEAD~3
```

今度は pick を edit にします。

```
edit 262cd2f Listing 12-4
edit fd7a7a5 Listing 12-5
edit 87c6e08 amend
...
```

```
PS(develop) > git rebase -i HEAD~3
Stopped at 262cd2f... Listing 12-4
You can amend the commit now, with
```

```
git commit --amend
```

```
Once you are satisfied with your changes, run
```

```
git rebase --continue
```

この状態でログを確認してみましょう。

```
PS(develop) > git log --oneline
262cd2f (HEAD) Listing 12-4
15c7cf3 (main) add poem.txt
9ac6088 Listing 12-2
ae11c5e Reading the Argument Values
f9b54f3 add Cargo.lock
95dd157 first commit
```

リベース対象の3つのコミットのうち、1つ目のコミットがすでに実行されている状態になっています。ここで、`git commit --amend` で書き換えます。

```
PS(develop) > git commit --amend -m "read the contents of the file"
[detached HEAD 3cf680d] read the contents of the file
  Date: Sat Feb 11 10:51:32 2023 +0900
  1 file changed, 6 insertions(+)
```

```
PS(develop) > git log --oneline
3cf680d (HEAD) read the contents of the file
15c7cf3 (main) add poem.txt
9ac6088 Listing 12-2
ae11c5e Reading the Argument Values
f9b54f3 add Cargo.lock
95dd157 first commit
```

コミットメッセージが書き換わっていることがわかります。次のコミットに進めるために `git rebase --continue` をします。

Info

ちなみに、ここで `git rebase --abort` をするとリベースを中止できます。

```
PS(develop) > git rebase --continue
Stopped at fd7a7a5... Listing 12-5
You can amend the commit now, with
```

```
git commit --amend
```

```
Once you are satisfied with your changes, run
```

```
git rebase --continue
```

同様にコミットメッセージを書き換えます。

```
PS(develop) > git commit --amend -m "add parse_config()"
[detached HEAD 51eeec5] add parse_config()
Date: Sat Feb 11 10:55:58 2023 +0900
1 file changed, 8 insertions(+), 2 deletions(-)

PS(develop) > git rebase --continue
Stopped at 87c6e08... amend
You can amend the commit now, with

  git commit --amend

Once you are satisfied with your changes, run

  git rebase --continue

PS(develop) > git commit --amend -m "improve parse_config()"
[detached HEAD f76653f] improve parse_config()
Date: Sat Feb 11 11:05:48 2023 +0900
1 file changed, 14 insertions(+), 8 deletions(-)

PS(develop) > git rebase --continue
Successfully rebased and updated refs/heads/develop.
```

ログを確認してみます。

```
PS(develop) > git log --oneline
f76653f (HEAD -> develop) improve parse_config()
51eeec5 add parse_config()
3cf680d read the contents of the file
15c7cf3 (main) add poem.txt
9ac6088 Listing 12-2
ae11c5e Reading the Argument Values
f9b54f3 add Cargo.lock
95dd157 first commit
```

書き換わっていますね。

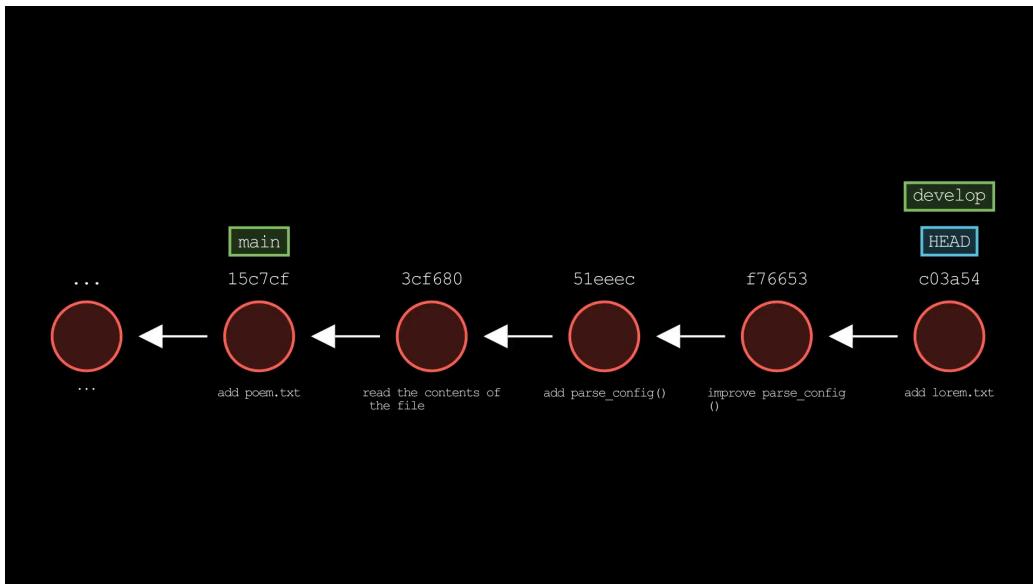
➤ コミットの並び替え

リベースを使えばコミットの順番を変えることもできます。試してみましょう。まず `lorem.txt` ファイルを追加してコミットしておきます。

*...
Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Donec quam felis, ultricies nec, pellentesque eu, pretium quis, sem. Nulla consequat massa quis enim. Donec pede justo, fringilla vel, aliquet nec, vulputate eget, arcu. In enim justo, rhoncus ut, imperdiet a, venenatis vitae, justo. Nullam dictum felis eu pede mollis pretium.*

```
PS(develop) > git add lorem.txt  
PS(develop) > git commit -m "add lorem.txt"  
[develop c03a54d] add lorem.txt  
 1 file changed, 1 insertion(+)  
 create mode 100644 lorem.txt
```

この時点でのコミットグラフは次のようにになっています。



リベースを実行します。

```
PS > git rebase -i HEAD~4
```

```
pick 3cf680d read the contents of the file  
pick 51eec5 add parse_config()  
pick f76653f improve parse_config()  
pick c03a54d add lorem.txt  
...
```

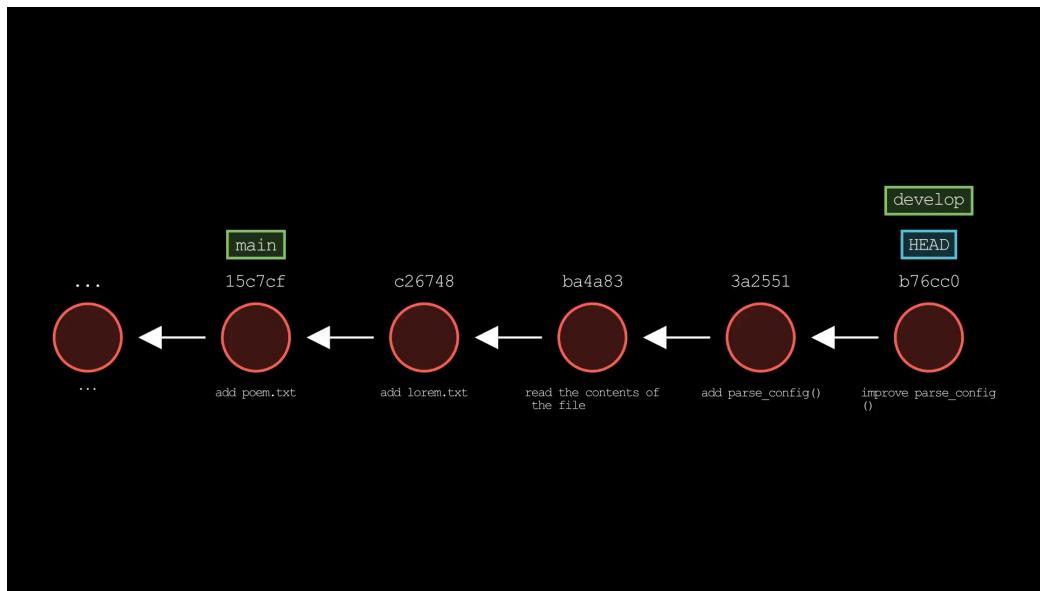
最後にコミットした add lorem.txt を最初に移動します。

```
pick c03a54d add lorem.txt  
pick 3cf680d read the contents of the file  
pick 51eec5 add parse_config()  
pick f76653f improve parse_config()  
...
```

```
PS(develop) > git rebase -i HEAD~4
Successfully rebased and updated refs/heads/develop.
```

```
PS(develop) > git log --oneline
b76cc0b (HEAD -> develop) improve parse_config()
3a2551a add parse_config()
ba4a836 read the contents of the file
c26748a add lorem.txt
15c7cf3 (main) add poem.txt
9ac6088 Listing 12-2
ae11c5e Reading the Argument Values
f9b54f3 add Cargo.lock
95dd157 first commit
```

コミットの順番が変わっていることが確認できます。



④ コミットをまとめる

リベースには複数のコミットをまとめる機能があります。試してみましょう。まずは、次のコミットをします。
src/main.rs を書き換えます。

```
use std::env;
use std::error::Error;
use std::fs;
use std::process;

fn main() {
    let args: Vec<String> = env::args().collect();

    let config = Config::build(&args).unwrap_or_else(|err| {
        println!("Problem parsing arguments: {}", err);
        process::exit(1);
    });

    println!("Searching for {}", config.query);
    println!("In file {}", config.file_path);

    if let Err(e) = run(config) {
        println!("Application error: {}", e);
        process::exit(1);
    }
}

fn run(config: Config) -> Result<(), Box<dyn Error>> {
    let contents = fs::read_to_string(config.file_path)?;

    println!("With text:\n{}", contents);

    Ok(())
}

struct Config {
    query: String,
    file_path: String,
}

impl Config {
    fn build(args: &[String]) -> Result<Config, &'static str> {
        if args.len() < 3 {
            return Err("not enough arguments");
        }

        let query = args[1].clone();
        let file_path = args[2].clone();

        Ok(Config { query, file_path })
    }
}
```

コミットします。

```
PS(develop) > git add -u  
PS(develop) > git commit -m "add error handling"  
[develop 01eba3d] add error handling  
1 file changed, 26 insertions(+), 8 deletions(-)
```

```
PS(develop) > git log --oneline  
01eba3d (HEAD -> develop) add error handling  
b76cc0b improve parse_config()  
3a2551a add parse_config()  
ba4a836 read the contents of the file  
c26748a add lorem.txt  
15c7cf3 (main) add poem.txt  
9ac6088 Listing 12-2  
ae11c5e Reading the Argument Values  
f9b54f3 add Cargo.lock  
95dd157 first commit
```

直近4つのコミットをまとめましょう。リベースを実行します。

```
PS(develop) > git rebase -i HEAD~4
```

最初のコミット以外を pick から squash に変えます。

```
pick ba4a836 read the contents of the file  
squash 3a2551a add parse_config()  
squash b76cc0b improve parse_config()  
squash 01eba3d add error handling  
...
```

次にコミットメッセージ (COMMIT_EDITMSG) がエディタで開かれます。

```
# This is a combination of 4 commits.
```

```
# This is the 1st commit message:
```

```
read the contents of the file
```

```
# This is the commit message #2:
```

```
add parse_config()
```

```
# This is the commit message #3:
```

```
improve parse_config()
```

```
# This is the commit message #4:
```

```
add error handling
```

```
...
```

最初のコミットメッセージだけ残しましょう。

```
# This is a combination of 4 commits.
```

```
# This is the 1st commit message:
```

```
read the contents of the file
```

```
...
```

```
PS(develop) > git rebase -i HEAD~4
```

```
[detached HEAD f17e83b] read the contents of the file
```

```
Date: Sat Feb 11 10:51:32 2023 +0900
```

```
1 file changed, 40 insertions(+), 4 deletions(-)
```

```
Successfully rebased and updated refs/heads/develop.
```

```
PS(develop) > git log --oneline
```

```
f17e83b (HEAD -> develop) read the contents of the file
```

```
c26748a add lorem.txt
```

```
15c7cf3 (main) add poem.txt
```

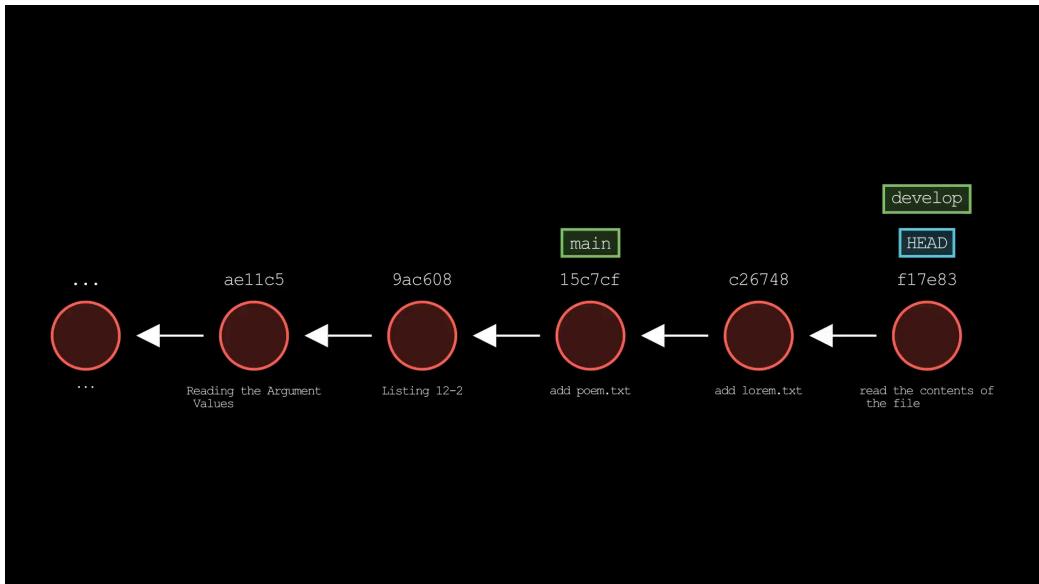
```
9ac6088 Listing 12-2
```

```
ae11c5e Reading the Argument Values
```

```
f9b54f3 add Cargo.lock
```

```
95dd157 first commit
```

最後にコミットしたスナップショットのまま、コミットがまとめられています。この時点のコミットグラフは次のとおりです。



今回のようにコミットメッセージを、最初の `squash` の前にあるコミットと同じでいいなら、`squash` ではなく `fixup` が使えます。試してみましょう。リベースを戻します。もうリベースを戻す方法はわかりますよね？ そろそろ `reflog` に表示される数も増えてきました。`git reflog -10` というように表示される数を制限できます。

リベースを戻したら、再度リベースを実行します。

```
PS(develop) > git rebase -i HEAD~4
```

今度は `squash` ではなく、`fixup` を指定します。

```
pick ba4a836 read the contents of the file
fixup 3a2551a add parse_config()
fixup b76cc0b improve parse_config()
fixup 01eba3d add error handling
...
...
```

```
PS(develop) > git rebase -i HEAD~4
Successfully rebased and updated refs/heads/develop.
```

```
PS(develop) > git log --oneline
c1b51d6 (HEAD -> develop) read the contents of the file
c26748a add lorem.txt
15c7cf3 (main) add poem.txt
9ac6088 Listing 12-2
ae11c5e Reading the Argument Values
f9b54f3 add Cargo.lock
95dd157 first commit
```

このように `squash` や `fixup` を使うことでコミットをまとめることができます。プルリクエストする際は、コミットをまとめることも大事です。ただし、コミットをまとめると1つのコミットに変更が多く含まれてしまうことがありますので、注意してください。プルリクエストする際のリベースについては各プロジェクトのポリシーに合わせましょう。

➡ autosquash

リベースはマージ以外にも squash や fixup など便利な機能があります。特に squash や fixup は上手く使えばコミットを整理することができて余計なコミットを減らすことができます。rebase のオプションには --autosquash というものがあります。これは、コミットメッセージが squash! や fixup! で始まるコミットに対してリベース時に処理を自動で設定してくれます。また、コミットするときに squash! や fixup! を追加するオプションも用意されています。試してみましょう。いくつかコミットします。まず、リファクタリングのため、src/lib.rs ファイルを作成します。

```

use std::error::Error;
use std::fs;
pub struct Config {
    pub query: String,
    pub file_path: String,
}

impl Config {
    pub fn build(args: &[String]) -> Result<Config, &'static str> {
        if args.len() < 3 {
            return Err("not enough arguments");
        }

        let query = args[1].clone();
        let file_path = args[2].clone();

        Ok(Config { query, file_path })
    }
}

pub fn run(config: Config) -> Result<(), Box<dyn Error>> {
    let contents = fs::read_to_string(config.file_path)?;

    println!("With text:\n{contents}");

    Ok(())
}

pub fn search<'a>(query: &str, contents: &'a str) -> Vec<&'a str> {
    vec![]
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn one_result() {
        let query = "duct";
        let contents = "\

Rust:
safe, fast, productive.
Pick three.";

        assert_eq!(vec!["safe, fast, productive."], search(query, contents));
    }
}

```

src/main.rs も書き換えます。

```

use minigrep::Config;
use std::env;
use std::process;
fn main() {
    let args: Vec<String> = env::args().collect();

    let config = Config::build(&args).unwrap_or_else(|err| {
        println!("Problem parsing arguments: {}", err);
        process::exit(1);
    });

    println!("Searching for {}", config.query);
    println!("In file {}", config.file_path);

    if let Err(e) = minigrep::run(config) {
        println!("Application error: {}", e);
        process::exit(1);
    }
}

```

コミットします。

```

PS(develop) > git add .
PS(develop) > git commit -m "refactor the main function"
[develop 58888c1] refactor the main function
 2 files changed, 49 insertions(+), 30 deletions(-)
 create mode 100644 src/lib.rs

```

しかし、このコードは実装が不十分で、テストに失敗します。テストが通るように修正します。`src/lib.rs` の `search` 関数を実装しましょう。

```

pub fn search<'a>(query: &str, contents: &'a str) -> Vec<&'a str> {
    let mut results = Vec::new();

    for line in contents.lines() {
        if line.contains(query) {
            results.push(line);
        }
    }

    results
}

```

これでテストが通ります。

```
PS(develop) > cargo test
Compiling minigrep v0.1.0 (F:\minigrep)
  Finished test [unoptimized + debuginfo] target(s) in 0.63s
    Running unitests src\lib.rs (target\debug\deps\minigrep-41749b9e62da306d.exe)

running 1 test
test tests::one_result ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s

  Running unitests src\main.rs (target\debug\deps\minigrep-bed007b557971686.exe)

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s

Doc-tests minigrep

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s
```

コミットしましょう。この時、この修正がリベース時に前のコミットに統合されうように設定しておきましょう。コミット時に `--fixup` を指定します。次のようにになります。

```
PS(develop) > git add -u
PS(develop) > git commit --fixup HEAD
[develop b414889] fixup! refactor the main function
  1 file changed, 9 insertions(+), 1 deletion(-)

PS(develop) > git log --oneline
b414889 (HEAD -> develop) fixup! refactor the main function
58888c1 refactor the main function
c1b51d6 read the contents of the file
c26748a add lorem.txt
15c7cf3 (main) add poem.txt
9ac6088 Listing 12-2
ae11c5e Reading the Argument Values
f9b54f3 add Cargo.lock
95dd157 first commit
```

ログを確認してみると、直前のコミットメッセージの前に `fixup!` が付与されたメッセージになっていることがわかります。

テストは通りましたが、実装した `search` 関数がテスト以外で使われていません。修正しましょう。

```
pub fn run(config: Config) -> Result<(), Box<dyn Error>> {
    let contents = fs::read_to_string(config.file_path)?;

    for line in search(&config.query, &contents) {
        println!("{}",<span style="color: green">{line}</span>);
    }

    Ok(())
}
```

実行してみます。

```
PS(develop) > cargo run -- the poem.txt
Compiling minigrep v0.1.0 (F:\minigrep)
    Finished dev [unoptimized + debuginfo] target(s) in 0.63s
        Running `target\debug\minigrep.exe the poem.txt`
Searching for the
In file poem.txt
Then there's a pair of us - don't tell!
To tell your name the livelong day
```

これも `--fixup` を指定してコミットします。ちなみに、`git commit` コマンドに `-a` を指定すると自動で変更されたファイルや削除されたファイルがステージングされます。

```
PS(develop) > git commit -a --fixup HEAD
[develop 44a9088] fixup! fixup! refactor the main function
 1 file changed, 3 insertions(+), 1 deletion(-)

PS(develop) > git log --oneline
44a9088 (HEAD -> develop) fixup! fixup! refactor the main function
b414889 fixup! refactor the main function
58888c1 refactor the main function
c1b51d6 read the contents of the file
c26748a add lorem.txt
15c7cf3 (main) add poem.txt
9ac6088 Listing 12-2
ae11c5e Reading the Argument Values
f9b54f3 add Cargo.lock
95dd157 first commit
```

それではリベースしましょう。`--autosquash` と `-i` を指定して `rebase` します。

```
PS(develop) > git rebase -i --autosquash HEAD~3
```

```
pick 58888c1 refactor the main function
fixup b414889 fixup! refactor the main function
fixup 44a9088 fixup! fixup! refactor the main function
...
...
```

自動で pick が fixup になっているのがわかります。そのまま実行します。

```
PS(develop) > git rebase -i --autosquash HEAD~3
Successfully rebased and updated refs/heads/develop.
```

```
PS(develop) > git log --oneline
e747550 (HEAD -> develop) refactor the main function
c1b51d6 read the contents of the file
c26748a add lorem.txt
15c7cf3 (main) add poem.txt
9ac6088 Listing 12-2
ae11c5e Reading the Argument Values
f9b54f3 add Cargo.lock
95dd157 first commit
```

リベースでまとめられていることがわかります。ちなみに、 fixup! fixup! ... と並んでしまっていますが、特に問題ないと思います。もし気になるなら --fixup の後ろのハッシュを pick となるコミットに指定してください。

Note

rebase.autosquash

リベースの度に --autosquash を指定するのが面倒な場合、 git config で常に有効にすることができます。

```
PS > git config --global rebase.autosquash true
```

➤ その他の機能

リベース時の編集では reword , edit , squash , fixup 以外にも指定できるものがあります。詳しくはドキュメントを参照してください。

リモートリポジトリ

Gitは分散型のバージョン管理システムです。バージョン管理のコアとなるリポジトリ(.gitディレクトリ)をサーバだけでなく、ローカルにも完全にコピーします。共有するサーバ(中央サーバ)にあるリポジトリはリモートリポジトリといい、**bare**リポジトリです。それに対して、個人がローカルに保持しているリポジトリをローカルリポジトリといいます。

スナップショットのバージョン管理は作業ツリーやインデックス、ローカルリポジトリで行うことができます。ローカルで作業した内容をリモートリポジトリにアップロードし、また他のユーザがアップロードした内容をローカルにダウンロードする必要があります。ここではリモートリポジトリに関するコマンドを解説します。

➤ git cloneコマンド

GitHubなど外部にあるリモートリポジトリからダウンロードしてローカルリポジトリを作成するには `git clone` コマンドを使います。

```
PS > git clone <repository_url>
```

通常はリポジトリ名のディレクトリが作成されます。ディレクトリ名を指定する場合は次のようにします。

```
PS > git clone <repository_url> <dirname>
```

他にもさまざまなオプションが用意されています。詳しくは `git clone` コマンドのドキュメントを参照してください。

クローンした直後の状態

例えば、次のプロジェクトをクローンしてみます。

```
PS > git clone git@github.com:mebiusbox/re-git-sample.git
```

この状態でローカルの作業ツリーやリポジトリがどのようにになっているか確認してみます。

```
PS > lat
.
├── .git
│   ├── info
│   │   └── exclude
│   ├── logs
│   ├── refs
│   │   ├── heads
│   │   │   └── main
│   │   └── remotes
│   │       └── origin
│   │           └── HEAD
│   └── HEAD
├── objects
│   ├── info
│   └── pack
│       ├── pack-d9a32a1ee5d4986e081d7f8d5782d6e4bc8734c1.idx
│       └── pack-d9a32a1ee5d4986e081d7f8d5782d6e4bc8734c1.pack
└── refs
    ├── heads
    │   └── main
    ├── remotes
    │   └── origin
    │       └── HEAD
    └── tags
├── config
├── description
├── HEAD
├── index
└── packed-refs
├── src
│   └── main.rs
├── .gitignore
├── Cargo.lock
├── Cargo.toml
├── poem.txt
└── README.md
```

初期はpackfileにまとめられていますね。中身を見てましょう。

```
PS > git verify-pack -v .\git\objects\pack\pack-d9a32a1ee5d4986e081d7f8d5782d6e4bc8734c1.idx
b3ed661567d7e58452689a3018e8aa9e0dc2bdd1 commit 224 154 12
0d22a3f04e0c11a5be123e36c34ac545dbd303d2 commit 84 95 166 1 b3ed661567d7e58452689a3018e8aa9e0dc2bdd1
c5e8b097c3eaba935d134fe21efd7d0115dc9a7d commit 251 166 261
16cfdd7208aff8a22ce54cb61e4590d3dcb1c6d7 commit 244 164 427
9cc0d7bda1f3c460e47993a394b1e8ed301debe6 commit 237 158 591
e386762b141def2bafcc12e093fd395c85cb94ec0 commit 240 164 749
77da8c11a1aca8b1c27d18bac1666142b62117ee commit 223 152 913
228d01e52845dcfc29f345dba4def49236b216fe commit 223 153 1065
88045a682a406f97e7fc7ddab3032680a8c4595a commit 238 162 1218
567c4c6b796d7d4c21a2590db94e2bf56737397c commit 225 153 1380
3142e1b2aa4b0a6af17f8c620760116f0a3bdd5 commit 175 118 1533
ca7b33d6167dcdcad5719ea549a6331de14cd096 tree 180 174 1651
2143293d2bce814d7b958246d9c49198fd7f77c9 tree 40 53 1825 1 ca7b33d6167dcdcad5719ea549a6331de14cd096
ea8c4bf7f35f6f77f75d92ad8ce8349f6e81ddba blob 8 17 1878
dcce6cf0e954c401a3869ed3c5f150e9f487462b blob 157 135 1895
d6733a5c0148389725f4f0f72dfce7f0f53d2092 blob 182 150 2030
c8d06d494eb4a93e574407e930f8b8616d8d8f0e blob 155 139 2180
87075273137fdea2814df14af10a6e6138de5986 blob 221 165 2319
372ec58f485d115da8fea8dbe1b04871e7680567 tree 35 46 2484
ae2fa7bb1bfb555f05411898d8ec66d316342b06 blob 218 156 2530
c8000128e0674e0ccbe1b7d0b2a151ec06d8129e tree 61 74 2686 1 ca7b33d6167dcdcad5719ea549a6331de14cd096
0f7869bc958cc28f5f6078e6106331373bc9568f blob 470 279 2760
ab7b5675904525570451278b07f78393880721a2 tree 69 75 3039
8e2013c4cc75a4d76189108d0662b7aeeb8af3c9 blob 1123 528 3114
d647344e1dfffb491e2525e33cc52971f2207a989 blob 1013 481 3642
381bc397c698f681d38a8b615bf5df4d95a093e7 blob 60 60 4123 1 d647344e1dfffb491e2525e33cc52971f2207a989
1d4463b9cb8f45c0e10324632bcd257c8dd6cdf tree 27 40 4183 2 c8000128e0674e0ccbe1b7d0b2a151ec06d8129e
77289c70187e4cd72cb24be3f5cf88a00950de0a tree 69 76 4223
67df13f6b4a9ce1576bae04711daa417f1ea55b6 blob 48 60 4299 1 8e2013c4cc75a4d76189108d0662b7aeeb8af3c9
058ff65b2a2a7e08d6b0b25421b56f5ff00fabbb9 tree 27 40 4359 2 c8000128e0674e0ccbe1b7d0b2a151ec06d8129e
f520183509c01d5374c3ec954a6d81573ae1506c tree 69 75 4399
747058fa2f66f97dff299bfa982ee4b3b406fd49 blob 19 31 4474 2 67df13f6b4a9ce1576bae04711daa417f1ea55b6
caaaec09785956388d103ddc7bbd1363eea381d9 tree 26 41 4505 3 058ff65b2a2a7e08d6b0b25421b56f5ff00fabbb9
f5bd16c8cddd8cc7487cf9ad9c21c196960c1cc8 tree 35 46 4546
173cfe58f3ab0d0a575bfb530e6f3ce48323afeb tree 27 40 4592 2 c8000128e0674e0ccbe1b7d0b2a151ec06d8129e
edc8cdae41f67d1b958f8c26018d72a1a5dbbdf2 tree 9 20 4632 1 ca7b33d6167dcdcad5719ea549a6331de14cd096
30a68996d7a203bdec0c997fdec403696be779e1 tree 27 39 4652 2 edc8cdae41f67d1b958f8c26018d72a1a5dbbdf2
4d92264f1a2ec60bd750fe99fb3093508465d03d tree 35 46 4691
ae7def53d662d6bf78b75cc11f9ce14ebb9ce0d4 blob 24 34 4737 1 ae2fa7bb1bfb555f05411898d8ec66d316342b06
362a7dca4d0c858e57567677e23c19746df6981e tree 27 40 4771 2 edc8cdae41f67d1b958f8c26018d72a1a5dbbdf2
305157a396c6858705a9cb625bab219053264ee4 tree 35 46 4811
e7a11a969c037e00a796aafeff6258501ec15e9a blob 45 53 4857
8a1d5d296b0b0b95c0b7a6c5768f5f7af5aac256 tree 8 19 4910 3 362a7dca4d0c858e57567677e23c19746df6981e
non delta: 28 objects
chain length = 1: 7 objects
chain length = 2: 6 objects
chain length = 3: 2 objects
.\git\objects\pack\pack-d9a32a1ee5d4986e081d7f8d5782d6e4bc8734c1.pack: ok
```

Gitオブジェクトは43個、28個は非差分(non delta)、他の15個は差分(deltaified)が格納されているようです。

他に、`.git/refs/remotes/origin/HEAD` がありますね。

```
PS > cat .git\refs\remotes\origin\HEAD
ref: refs/remotes/origin/main
```

ログを見てみると次のようになっています。

```
PS > git log --decorate --oneline --all
0d22a3f (HEAD -> main, origin/main, origin/HEAD) add README
c5e8b09 (origin/develop) fixup! fixup! refactor the main function
16cfdd7 fixup! refactor the main function
9cc0d7b refactor the main function
e386762 read the contents of the file
b3ed661 add lorem.txt
77da8c1 add poem.txt
228d01e Listing 12-2
88045a6 Reading the Argument Values
567c4c6 add Cargo.lock
3142e1b first commit
```

➤ リモートリポジトリの確認

リモートリポジトリを確認するには `git remote show` コマンドを使います。

```
PS > git remote show
origin
```

リモート名が確認できます。次に、リモートリポジトリの詳細を確認するには `git remote` コマンドに `-v` を指定します。

```
PS > git remote -v
origin  git@github.com:mebiusbox/re-git-sample.git (fetch)
origin  git@github.com:mebiusbox/re-git-sample.git (push)
```

また、`git remote show` コマンドにリモート名を指定して詳細を確認できます。

```
PS > git remote show origin
* remote origin
  Fetch URL: git@github.com:mebiusbox/re-git-sample.git
  Push URL: git@github.com:mebiusbox/re-git-sample.git
  HEAD branch: main
  Remote branches:
    develop tracked
    main     tracked
  Local branch configured for 'git pull':
    main merges with remote main
  Local ref configured for 'git push':
    main pushes to main (up to date)
```

➤ リモートリポジトリの追加

例えば、ローカルリポジトリを作成し、外部サーバに作成したリモートリポジトリを設定したい場合、`git remote add` コマンドを使います。

```
PS > git remote add <name> <url>
```

例えば、`git@github.com:hoge/hoge.github.com.git` というリモートリポジトリを `origin` という名前で追加する場合、次のようにします。

```
PS > git remote add origin git@github.com:mebiusbox/re-git-sample.git
```

➤ リモートリポジトリの削除

`git remote rm` コマンドを使います。

```
PS > git remote rm origin
```

Info

リモートリポジトリそのものを削除するわけではありません。ローカルにあるリモートリポジトリの情報を削除しているだけです。

➤ リモートリポジトリの名前変更

`git remote rename` コマンドを使います。

```
PS > git remote rename origin hoge
```

➤ フェッチ

リモートリポジトリからデータをダウンロードするには、`git fetch` コマンドを使います。

```
PS > git fetch
```

リモートリポジトリは複数追加できます。その場合、`git fetch` コマンドの後にリモート名を指定します。

```
PS > git remote add hoge git@github.com:mebiusbox/re-git-sample.git
```

```
PS > git remote -v
hoge    git@github.com:mebiusbox/re-git-sample.git (fetch)
hoge    git@github.com:mebiusbox/re-git-sample.git (push)
origin  git@github.com:mebiusbox/re-git-sample.git (fetch)
origin  git@github.com:mebiusbox/re-git-sample.git (push)
```

```
PS > git fetch hoge
From github.com:mebiusbox/re-git-sample
 * [new branch]      develop    -> hoge/develop
 * [new branch]      main       -> hoge/main
```

ブランチとタグ

リモートリポジトリでブランチやタグが削除されても、通常はフェッチしても、ローカルにあるリモート追跡ブランチは自動的に削除されません。削除するには `git fetch` コマンドで `--prune` や `-prune-tags`` を指定します。

```
PS > git pull --prune --prune-tags
```

プル

フェッチと同時に現在のブランチをマージする場合、`git pull` コマンドを使います。

```
PS > git pull
```

➤ プッシュ

ローカルリポジトリからリモートリポジトリにデータをアップロードするには `git push` コマンドを使います。通常はローカルリポジトリにあるすべてのデータをアップロードせずに、特定のブランチのみアップロードします。アップロードするとき、リモート名とブランチ名を指定します。

```
PS > git push <remote-name> <branch-name>
```

例えば、リモート名が `origin` で、ブランチ名が `main` なら次のようにになります。

```
PS > git push origin main
```

もし、`origin`のサーバに `main` ブランチがなければ作成されます。

上流ブランチ

`git push` コマンドではさまざまな指定方法があります。例えば、ブランチ名を省略して使うこともできます。

```
PS > git push origin
```

しかし、次のようなエラーが出力されるかもしれません。

```
PS > git push origin
fatal: The current branch main has no upstream branch.
To push the current branch and set the remote as upstream, use

  git push --set-upstream origin main

To have this happen automatically for branches without a tracking
upstream, see 'push.autoSetupRemote' in 'git help config'.
```

これは現在のブランチ(`main`)に上流ブランチ(`upstream branch`)がないという意味です。上流ブランチはリモート追跡ブランチのことです。通常はクローンしたときに自動で設定されます。

現在のブランチの上流ブランチを確認する場合、`git branch -vv` コマンドを使います。

```
PS > git branch -vv
* main 0d22a3f [origin/main] add README
```

`main` ブランチの上流ブランチが `origin/main` ということがわかります。もし、手動で上流ブランチを設定したい場合は `git branch` コマンドに `-u` を指定します。例えば、現在の `main` ブランチに `origin/main` リモート追跡ブランチを設定する場合は次のようになります。

```
PS > git branch -u origin/main
```

上流ブランチを設定したら、再度 `push` してみましょう。

```
PS > git push origin
Everything up-to-date
```

問題なさそうです。

タグ

ローカルで作成したタグをリモートリポジトリに追加するには、ブランチ同様にタグを指定します。

```
PS > git push origin v0.1
```

すべてのタグを追加する場合、`--tags` を指定します。

```
PS > git push origin --tags
```

リモートリポジトリからブランチやタグを削除するには

`git push` に `-d` を指定します。

```
PS > git push origin -d develop  
PS > git push origin -d v0.1
```

リセット

Gitはバージョン管理ソフトです。いつでも指定のバージョンに戻すことができます。現在のブランチに対して、スナップショットを指定のバージョンに戻す方法は `git reset` コマンドと `git restore` コマンドです。

Note

いろんなところで、`reset`と`checkout`を合わせて解説されているのを見ます。以前はこの組み合わせでしたが、`checkout`は機能が多く`switch`や`restore`コマンドが作られました。わざわざ分けたぐらいですから、やはり分かりづらかったり、混乱しやすかったのでしょう。今も`checkout`を使うことはできますが、`restore`の方が明解です。また、本書でも、`checkout`は使わずに`switch`や`restore`で解説しています。そして、gitが出力するコマンドも`checkout`ではなく、`switch`や`restore`に変わっています。

リセットの対象は何か？

`git reset` コマンドや `git restore` が何を対象としているのか理解することが大事です。何となく使っている人は`--hard` はともかく、`--soft` や `--mixed` もまだ曖昧だったりしていませんか。これらの違いは何を対象としているのかに過ぎません。ですから、これらのオプションがまだ曖昧だと思うのであれば、対象もまた曖昧のままであると考えられます。

では、対象は何でしょうか。まずは以下の3つの領域(エリア)が挙げられます。

- 作業ツリー
- インデックス
- コミット

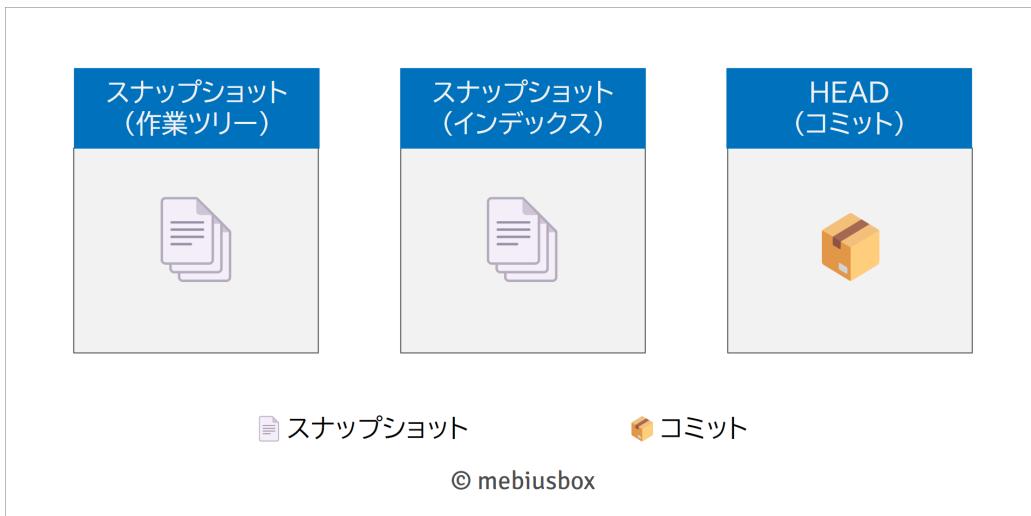
ですが、これらはまだ曖昧です。もっと明確にしていきましょう。最初に作業ツリーですが、作業ツリーの中でも対象となるのは追跡している(**Tracked**)ファイルです。追跡していない(**Untracked**)ファイルは対象ではありません。そして、コミットは `HEAD` が対象です。

Note

追跡しているファイルを確認したい場合、`git ls-files` コマンドを使います。また、追跡していないファイルは`-o` を指定します。

次に、作業ツリーとインデックスはスナップショットが変更されますが、`HEAD`は参照しているコミットが変更されます。

図で表すと次のようになります。

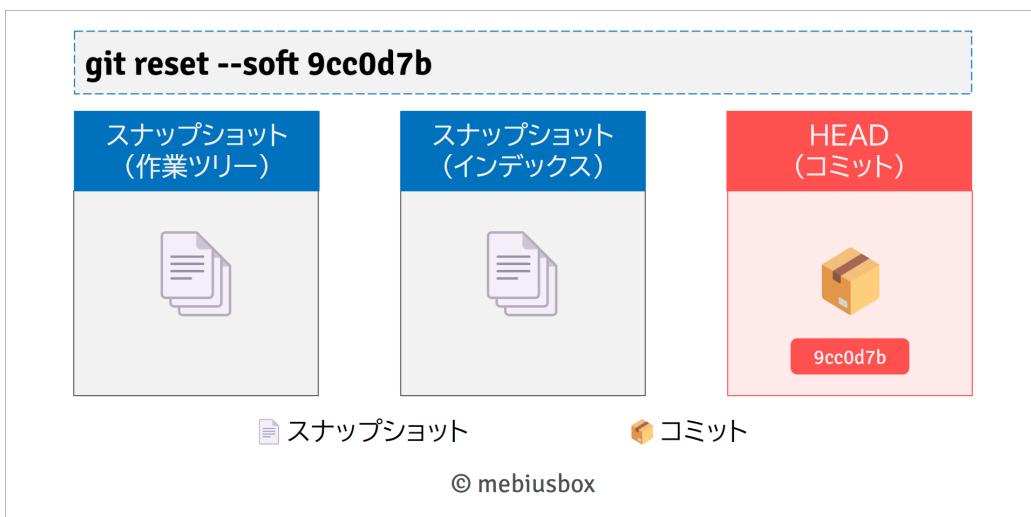


➤ git resetコマンド

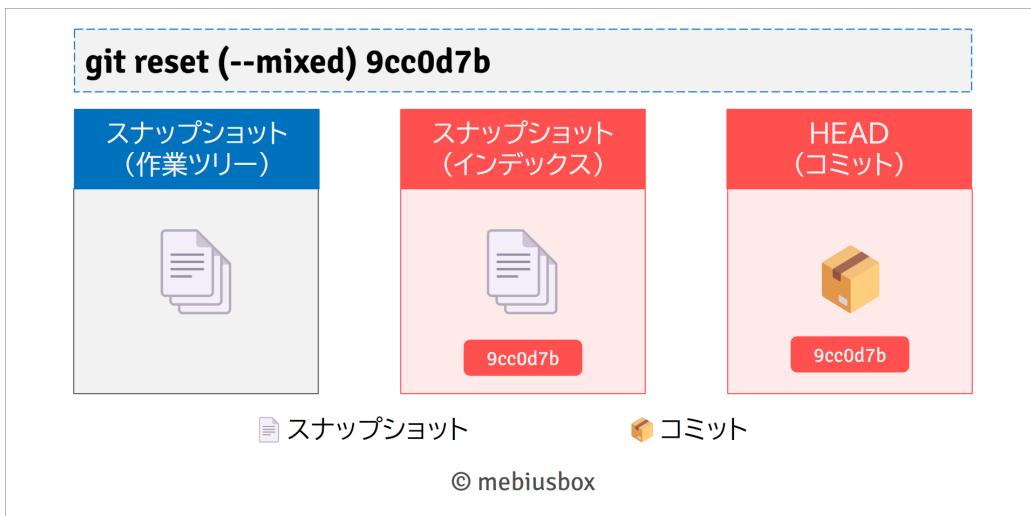
git reset コマンドの基本的な書式は次のとおりです。

```
git reset [--soft | --mixed | --hard] <commit>
```

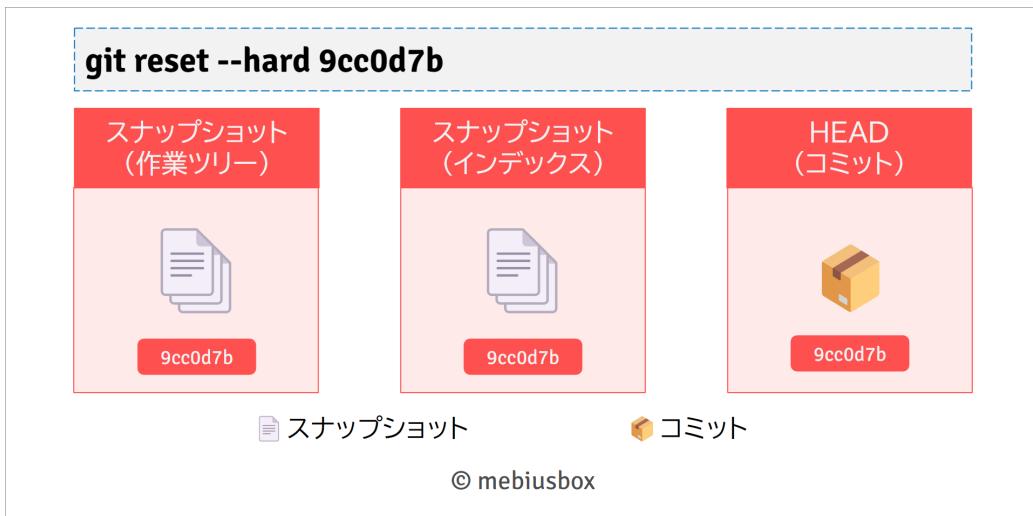
--soft , --mixed , --hard はそれぞれ次のような動作をします。



●●●



•••



>Note

あるコミットのファイルリストを確認したい場合、 `git ls-tree` コマンドで確認できます。 `-r` オプションを指定するとサブディレクトリも対象になります。

```
PS > git ls-tree -r HEAD
100644 blob ea8c4bf7f35f6f77f75d92ad8ce8349f6e81ddba    .gitignore
100644 blob dcce6cf0e954c401a3869ed3c5f150e9f487462b    Cargo.lock
100644 blob d6733a5c0148389725f4f0f72dfce7f0f53d2092    Cargo.toml
100644 blob 87075273137fdea2814df14af10a6e6138de5986    poem.txt
100644 blob ae2fa7bb1bfb555f05411898d8ec66d316342b06    src/main.rs
```

➤ git restoreコマンド

`git restore` コマンドの基本的な書式は次のとおりです。

```
git restore [--source=<tree>] [--staged] [--worktree] <pathspec>...
```

`--staged` , `--worktree` の組み合わせの動作は次のようにになります。

```
git restore --source=9cc0d7b .
```



📄 スナップショット

📦 コミット

© mebiusbox

•••

```
git restore --source=9cc0d7b --staged .
```



📄 スナップショット

📦 コミット

© mebiusbox

•••

```
git restore --source=9cc0d7b --staged --worktree .
```



📄 スナップショット

📦 コミット

© mebiusbox

➤ コミットの親

リセットなどのコマンドにはコミットからの相対参照を使用することがあります。例えば、HEADの1つ前などです。この場合、`^` を後ろにつけます。

```
PS > git show HEAD^
```

^ が複数ある場合、その回数分親を辿ったコミットを指します。

```
PS > git show HEAD^^^
```

^ の後ろに数字を付けると親の番号になります。例えば、マージコミットのように複数の親がいる場合に使います。

```
PS > git show HEAD^2
```

また、 ~ も親を指します。

```
PS > git show HEAD~
```

ですから、 HEAD^ も HEAD~ も同じコミットを指します。 ~ も複数並べられます。

```
PS > git show HEAD~~~
```

また、 ~ の後ろに数字も指定できます。この場合、数字分の親を辿ったコミットを指します。つまり、次のコマンドはどちらも同じです。

```
PS > git show HEAD~3
```

```
PS > git show HEAD~~~
```

数字を指定しなければ、 ^ と ~ は同じになります。

```
PS > git show HEAD^^^
```

```
PS > git show HEAD~~~
```

ですが、数字を指定した場合、意味が変わるので注意してください。以下は指しているコミットが違います。

```
PS > git show HEAD~~^2
```

```
PS > git show HEAD^^~2
```

➤ ユースケース

uncommit

最後に実行したコミットを取り消したい場合、次のようにになります。

```
PS > git reset --soft HEAD~
```

unstage

インデックスにあるスナップショットをすべて戻したい場合、次のようにになります。

```
PS > git reset HEAD
```

git reset や git restore には他にもさまざまなオプションがあります。 詳しくはドキュメントを参照してください。

参照ログ

Gitは何らかの理由でブランチのHEAD(参照)が更新されるたびにその情報を格納します。これらの参照ログは .git/logs ディレクトリにあります。

```
PS > lat .git\logs  
logs  
|   └── refs  
|       |   └── heads  
|       |       └── main  
|       |       └── v0.1  
|       └── remotes  
|           └── origin  
|               └── HEAD  
└── HEAD
```

参照ログを確認するには git reflog コマンドを使います。

```
PS > git reflog  
77da8c1 HEAD@{0}: checkout: moving from main to v0.1  
0d22a3f HEAD@{1}: reset: moving to HEAD~  
8346cd7 HEAD@{2}: cherry-pick: add lorem.txt  
0d22a3f HEAD@{3}: reset: moving to HEAD~  
ca3213f HEAD@{4}: cherry-pick: add lorem.txt  
0d22a3f HEAD@{5}: clone: from https://github.com/mebiusbox/re-git-sample.git
```

参照ログはローカルの情報なので、クローンしたときは参照ログはありません。参照ログを使えば、コミット以外で HEADが更新されたときに、その前の状態を参照できます。その場合、 @{n} 形式で参照します。

```
PS > git show HEAD@{5}  
commit 0d22a3f04e0c11a5be123e36c34ac545dbd303d2  
Author: mebiusbox <mebiusbox@gmail.com>  
Date:   Wed Feb 15 17:20:45 2023 +0900  
  
    add README  
    ...
```

ちなみに、PowerShellの場合、 @{...} は特別な意味を持つので、クオーテーションで囲む必要があります。

```
PS > git show 'HEAD@{5}'
```

参照ログをログ形式で確認したい場合、 git log コマンドに -g を指定します。

```
PS > git log -g
commit 77da8c11a1aca8b1c27d18bac1666142b62117ee
Reflog: HEAD@{0} (mebiusbox <mebiusbox@gmail.com>)
Reflog message: checkout: moving from main to v0.1
Author: mebiusbox <mebiusbox@gmail.com>
Date:   Wed Feb 15 16:49:24 2023 +0900

add poem.txt
...

```

また、`git reflog` は現在のブランチの参照ログが対象ですが、それ以外の参照ログを確認したい場合、`git log` コマンドに `--reflog` を指定します。

```
PS > git log --reflog --oneline
8346cd7 add lorem.txt
ca3213f add lorem.txt
0d22a3f add README
77da8c1 add poem.txt
228d01e Listing 12-2
88045a6 Reading the Argument Values
567c4c6 add Cargo.lock
3142e1b first commit
```

差分表示

差分を表示するコマンドは `git diff` コマンドです。

④ 作業ツリー、インデックス、コミット

まず、基本として、作業ツリー、インデックス、コミットをそれぞれ比較したい場合は次のようにになります。

作業ツリーとインデックス

```
PS > git diff
```

作業ツリーとコミット

```
PS > git diff HEAD
```

インデックスとコミット

```
PS > git diff --cached
```

④ ファイル単位

コマンドでファイルパスを指定します。

```
PS > git diff README.md  
PS > git diff HEAD README.md  
PS > git diff --cached README.md
```

ブランチとファイル名が被った場合

例えば、`test` ディレクトリを比較しようと次のコマンドを実行しました。

```
PS > git diff test  
fatal: ambiguous argument 'test': both revision and filename  
Use '--' to separate paths from revisions, like this:  
'git <command> [<revision>...] -- [<file>...]'
```

これは、`test` というブランチがあったため、ファイルパスなのかブランチなのか曖昧でエラーとなっています。指示にあるとおり、`--` の後にファイルパスを記述することで解決できます。

```
PS > git diff -- test
```

➤ その他のオプション

-U<n>, -unified=<n>

変更のあった行の前後を表示します。前後の行数を指定できます。

-numstat

追加した行と削除した行をファイルごとに表示します。

```
PS > git diff --numstat
1      1      README.md
1      1      test/test.rs
```

-shortstat

変更のあったファイルの数、追加した行と削除した行の統計を表示します。

```
PS > git diff --shortstat
2 files changed, 2 insertions(+), 2 deletions(-)
```

-name-only

作業ツリーで変更のあるファイル名を表示します。

```
PS > git diff --name-only
README.md
test/test.rs
```

-name-status

作業ツリーで変更のあるファイル名を状態とともに表示します。

```
PS > git diff --name-status
M      README.md
M      test/test.rs
```

-word-diff

通常は行単位での差分表示がされます。このオプションを有効にすると単語単位で差分が表示されます。追加箇所は `{+...+}`、削除箇所は `{-...-}` で表示されます。また、`--word-diff=color` とすることで、先ほどの表記ではなく色を付けて表示されます。標準ではマルチバイト文字の区切りはよくありません。その場合、`--word-diff-regex` で単語に一致する正規表現を指定できます。ただ、日本語に最適な正規表現を私は知らないので、現在模索中です。例えば、以下のように指定しています。

```
PS > git diff --word-diff=color --word-diff-regex="[^[:space:]]|[\xc0-\xff][\x80-\xbf]+"
```

`--word-diff=color --word-diff-regex=...` は `--color-words=...` と置き換えることができます。

```
PS > git diff --color-words="[^[:space:]]|[\xc0-\xff][\x80-\xbf]+"
```

長いのでエイリアスとして登録したり、`git config` で `diff.wordRegex` に指定したりします。直接 `.gitconfig` に追加する場合はエスケープする必要があるので注意が必要です。

```
[alias]
dw = diff --color-words="[^[:space:]]|[\xc0-\xff][\x80-\xbf]+"
```

Info

この指定では文字化けする場合があります。詳細をわかりませんが、例えば、マルチバイト文字で3バイト分を単語として一致したとします。しかし、比較ではバイト単位で行われているようで、1、2バイト目が不一致で3バイト目が一致だった場合、3バイト目を差分として認識しないようで、カラーコードなどが不正な位置に挿入されてしまい、文字化けするようです。ですので、上記の指定で文字化けする場合は、単語単位ではなく行単位で差分表示するように使い分けるといいかなと思います。

Note

`word-diff-regex` の初期値は `userdiff.c` を参照してください。

<https://github.com/git/git/blob/master/userdiff.c>

それ以外のオプション

コマンドのドキュメントを参照してください。

➡ コミットの範囲指定

..構文

git diff コマンドや git log コマンドで、コミットの範囲を指定したいことがしばしばあります。その場合、.. を使って指定します。

```
<commit-ish-1>..<commit-ish-2>
```

また、コミットはブランチをまたがって指定できます。この場合、1つのコミットからは辿れるけど、もう1つのコミットからはたどれないコミットの範囲を調べられます。そして、これは次のように書くこともできます。

```
^<commit-ish-1> <commit-ish-2>
<commit-ish-2> --not <commit-ish-1>
```

例えば、次のようなコミットグラフがあるとします。

- 9f32671 (HEAD -> main, test) add test/
- 0d22a3f (origin/main) [v0.2] add README
 - c5e8b09 (develop, origin/develop) fixup! fixup! refactor the main function
 - 16cfdd7 fixup! refactor the main function
 - 9cc0d7b refactor the main function
 - e386762 read the contents of the file
 - b3ed661 add lorem.txt
- 77da8c1 (v0.1) [v0.1] add poem.txt
- 228d01e Listing 12-2
- 88045a6 Reading the Argument Values
- 567c4c6 add Cargo.lock
- 3142e1b first commit

各コマンドの実行結果は次のとおりです。

```
PS > git log --oneline --decorate main..develop
c5e8b09 (origin/develop, develop) fixup! fixup! refactor the main function
16cfdd7 fixup! refactor the main function
9cc0d7b refactor the main function
e386762 read the contents of the file
b3ed661 add lorem.txt
```

```
PS > git log --oneline --decorate ^main develop
c5e8b09 (origin/develop, develop) fixup! fixup! refactor the main function
16cfdd7 fixup! refactor the main function
9cc0d7b refactor the main function
e386762 read the contents of the file
b3ed661 add lorem.txt
```

```
PS > git log --oneline --decorate develop --not main
c5e8b09 (origin/develop, develop) fixup! fixup! refactor the main function
16cfdd7 fixup! refactor the main function
9cc0d7b refactor the main function
e386762 read the contents of the file
b3ed661 add lorem.txt
```

...構文

両方のコミットグラフに存在するコミットを除外する場合、...を使います。

```
PS > git log --oneline --decorate main...develop
9f32671 (HEAD -> main, test) add test/
0d22a3f (tag: v0.2, origin/main, origin/HEAD) add README
c5e8b09 (origin/develop, develop) fixup! fixup! refactor the main function
16cfdd7 fixup! refactor the main function
9cc0d7b refactor the main function
e386762 read the contents of the file
b3ed661 add lorem.txt
```

git log コマンドに...を使う場合、--left-rightを使うと、どちらのコミットグラフに存在するコミットなのかわかりやすくなります。

```
PS > git log --oneline --decorate --left-right main...develop
< 9f32671 (HEAD -> main, test) add test/
< 0d22a3f (tag: v0.2, origin/main, origin/HEAD) add README
> c5e8b09 (origin/develop, develop) fixup! fixup! refactor the main function
> 16cfdd7 fixup! refactor the main function
> 9cc0d7b refactor the main function
> e386762 read the contents of the file
> b3ed661 add lorem.txt
```

git revert

コミットの取り消しや並び替えなどのコミットグラフの操作を行う場合、すでにプッシュしているコミットに対して行うべきではありません。どうしてもプッシュしたコミットを取り消したい場合、そのコミットの変更を戻すコミットを行って対応します。それが、`git revert` コマンドです。これは、指定したコミットの変更を戻すコミットを作成します。

```
PS > git revert HEAD~
```

このコマンドを実行するとコミットメッセージを編集するためにエディタが開かれます。つまり、このコマンドはコミットも行います。

git push -force (-f)

また、すでにプッシュしたコミットグラフをどうしても書き換える場合、`git push` コマンドに `--force (-f)` を使って強制的に書き換える方法があります。状況によってはやむを得ない場合があります。チームで開発している場合、必ず確認を取ってから行いましょう。

git blame

変更した箇所はログを見ることで確認できますが、例えば、編集しているときや不具合を見つける場合、ファイルを行単位で誰が、いつ変更したか確認できると便利です。それが git blame コマンドです。

```
PS > git blame src/main.rs
88045a68 (mebiusbox 2023-02-15 16:36:56 +0900 1) use std::env;
88045a68 (mebiusbox 2023-02-15 16:36:56 +0900 2)
^3142e1b (mebiusbox 2023-02-15 16:34:40 +0900 3) fn main() {
88045a68 (mebiusbox 2023-02-15 16:36:56 +0900 4)     let args: Vec<String> = env::args().collect();
228d01e5 (mebiusbox 2023-02-15 16:48:42 +0900 5)
228d01e5 (mebiusbox 2023-02-15 16:48:42 +0900 6)     let query = &args[1];
228d01e5 (mebiusbox 2023-02-15 16:48:42 +0900 7)     let file_path = &args[2];
228d01e5 (mebiusbox 2023-02-15 16:48:42 +0900 8)
228d01e5 (mebiusbox 2023-02-15 16:48:42 +0900 9)     println!("Searching for {}", query);
228d01e5 (mebiusbox 2023-02-15 16:48:42 +0900 10)    println!("In file {}", file_path);
^3142e1b (mebiusbox 2023-02-15 16:34:40 +0900 11) }
```

GitLens拡張機能

Visual Studio CodeならGitLens拡張機能を使うことで、エディタ上で行ごとの履歴を確認できます。

<https://marketplace.visualstudio.com/items?itemName=eamodio.gitlens>

git stash

作業している途中で、マージしたり、ブランチを切り替えたりする必要があるかもしれません。その場合、作業中の状態をスタックに退避できます。退避するには `git stash` コマンドを使います。

```
PS > git stash
```

スタックを確認したい場合、 `list` を指定します。

```
PS > git stash list
stash@{0}: WIP on master: 049d078 added the index file
stash@{1}: WIP on master: c264051 Revert "added file_size"
stash@{2}: WIP on master: 21d80a5 added number to log
```

スタックから復元するには `apply` を使います。

```
PS > git stash apply
PS > git stash apply "stash@{1}"
```

また、`stash` はステージされた内容も保持しますが、`apply`時に `--index` を指定しないとステージの状態が復元されません。

```
PS > git stash apply --index
```

復元してもスタックには残ったままになります。削除するには `drop` を使います。

```
PS > git stash drop "stash@{0}"
```

ステージされた内容を`stash`の対象にしたくない場合、 `--keep-index` を指定します。

```
PS > git stash --keep-index
```

`stash`が対象とするのは追跡されたファイルだけですが、追跡していないファイルも対象とする場合、 `--include-untracked (-u)` を使います。

```
PS > git stash -u
```

スタックからブランチを作成することもできます。

```
PS > git stash branch <name>
```

▣ サブモジュール

サブモジュールはプロジェクトの中に別のGitプロジェクトを追加することができる機能です。

サブモジュールが追加されるプロジェクトのことを `superproject` と呼びます。サブモジュールを追加すると、`superproject` にサブモジュールの Git ディレクトリ、`gitlink` ファイル、`.gitmodules` ファイルが追加され、`superproject` の `$GIT_DIR/config` ファイルが変更されます。

サブモジュールを追加するには `git submodule add` コマンドでリポジトリのURLを指定します。そうすると、`.gitmodules` ファイルが作成(すでに存在すれば追記)され、サブモジュールの初期化とともにプロジェクトがチェックアウトされます。チェックアウトされたサブモジュールディレクトリの中に `.git` ファイルが作成されます。その `.git` ファイルにはサブモジュールの Git ディレクトリの場所が記載されています。特に指定していなければ `$GIT_DIR/modules` 以下にサブモジュールの Git ディレクトリが作成され、そこを指します。`superproject` からはサブモジュールのディレクトリは `gitlink` ファイルとして登録されます。このファイルはサブモジュールのコミットを参照しています。

他のユーザーとサブモジュールを共有するために、`gitlink` ファイルと `.gitsubmodule` ファイルが必要です。サブモジュールを持っているプロジェクトを普通に `clone` しただけだと、サブモジュールはチェックアウトされず、フォルダだけ作成されます。この状態はサブモジュールが初期化されていない状態なので、`git submodule init` コマンドで初期化することができます。また、`clone` したときにサブモジュールも同時に初期化したい場合は `clone` 時に `--recursive` オプションを追加して実行します。

サブモジュールを削除するには、まず、`git submodule deinit` コマンドで実行します。`submodule deinit` コマンドを実行するとサブモジュールのディレクトリ内が空になります。また、`$GIT_DIR/config` ファイルから該当のサブモジュール情報が削除されます。しかし、このコマンドは未初期化状態に戻すだけなので、`gitlink` ファイルや `.gitmodules` ファイルには変更がありません。そこで `git rm` コマンドを実行してサブモジュールの `gitlink` ファイル(サブモジュールのディレクトリ)を削除します。そうすると `.gitmodules` から該当のサブモジュール情報も削除されます。この変更をコミットすればサブモジュールを削除することができます。しかし、サブモジュールのGitディレクトリは履歴のために削除されません。完全に削除するには手動でサブモジュールのディレクトリを削除します(通常は `$GIT_DIR/modules` ディレクトリにあります)。

▷ サブモジュールの作成

`git submodule` コマンドに `add` で追加するプロジェクトのURLを指定します。

```
PS > git submodule add https://github.com/hoge/foo
```

サブモジュールの情報は `.gitmodules` に追加されます。

▷ サブモジュールの初期化

サブモジュールを使用しているプロジェクトをクローンした場合、サブモジュールは自動でクローンされません。この場合、`git submodule` コマンドに `init` を指定します。

```
PS > git submodule init
```

または、クローン時に `--recursive` を指定するとサブモジュールも自動でクローンします。

```
PS > git clone --recursive https://github.com/hoge/project
```

④ サブモジュールの更新

サブモジュールは、それぞれ独立した環境で操作ができます。サブモジュールの更新は `gitlink` ファイルに反映されますので、このファイルのスナップショットを含めることで他の人と共有します。

また、`fetch`して`merge`した場合でも `.gitlink` ファイルが更新されただけで、サブモジュールの実体に変更はありません。その場合、`git submodule` コマンドに `update` を指定します。

```
PS > git submodule update
```

また、`--init` オプションを追加すると初期化を同時に行うことができます。

```
PS > git submodule update --init
```

さらに、サブモジュールが別のサブモジュールを持っていて、それらも含めたい場合は `--recursive` を追加します。

```
PS > git submodule update --recursive
```

```
PS > git submodule update --init --recursive
```

Gitはパスを指定した設定をサポートしています。ディレクトリに `.gitattributes` ファイルがあれば、そのディレクトリに対して適用されます。もちろん、ルートディレクトリに作成してすべてのファイルに対して適用させることもできます。

▷ バイナリファイル

Gitは改行を処理する機能が入っています。ですが、バイナリファイルに対して改行処理が入っては困ります。その場合、特定のファイルがバイナリファイルであることをGitに知らせなければなりません。例えば、`.bin` というファイルをバイナリファイルとして扱い場合、`.gitattributes` に追加します。

```
*.bin binary
```

▷ 差分ツール

特定のファイルに対して、通常のDiffとは違うDiffツールを使いたい場合、`diff` を指定します

```
*.docx diff=word
```

ここで指定している `word` はフィルターネームです。フィルターは `gitconfig` に設定を追加します。

```
PS > git config diff.word.textconv docx2txt
```

`.textconv` を指定した場合、テキストに変換するツールを指定します。そうすることで、通常のDiffツールを使って差分を表示します。

▷ フィルタ

チェックアウト時、ステージング時に行われるフィルターを個別に指定できます。それぞれ `smudge` (チェックアウト時)、`clean` (ステージング時) フィルタといいます。

まず、`.gitattributes` にフィルターを指定します

```
*.c filter=indent
```

そして、`gitconfig` にフィルターを追加します

```
PS > git config --global filter.indent.clean indent
PS > git config --global filter.indent.smudge cat
```

➡ アーカイブ

最新のスナップショットのアーカイブを作成するには `git archive` コマンドを使います

```
PS > git archive main -o output.zip
```

`git archive` コマンドはすべてのファイルが対象となります。一部を除外したい場合、属性で制御できます。

例えば、`test/` ディレクトリを除外したい場合、`.gitattributes` に以下を追加します。

```
test/ export-ignore
```

アーカイブ作成時に、`--worktree-attributes` を指定することで、`.gitattributes` の設定が読み込まれます。

```
PS > git archive HEAD --worktree-attributes -o output.zip
```

GitHub

GitHubはGitのホスティングサービスです。

README

リポジトリのトップページに表示されます。 README や README.md など。

CONTRIBUTING

CONTRIBUTING.md のファイルにはプルリクエストに関するガイドラインを明記します。

GitHub CLI

プルリクエストは Git の機能ではなく、GitHubなどのホスティングサービスが提供しています。GitHub CLIはコマンドラインでGitHubにたいしてさまざまな操作が行えるものです。インストールは WinGet がお手軽です。

```
PS > winget install -e --id GitHub.cli
```

GitHub CLI は gh コマンドです。

プルリクエストの確認

gh pr コマンドでプルリクエストの関する操作が行えます。 list を指定すれば、プルリクエストを確認できます。

```
PS > gh pr list
Showing 1 of 1 open pull request in mebiusbox/re-git-sample

#1 Develop develop less than a minute ago
```

特定のプルリクエストをチェックアウトしたい場合、次のようにします。

```
PS > gh pr checkout 1
```

この場合、#1のプルリクエストをチェックアウトします。

プルリクエストされた内容をテスト

プルリクエストされた内容をローカルで確認したいことがあります。まずは、プルリクエストをフェッチします。特定のプルリクエストをフェッチしたい場合は次のようにします。

```
PS > git fetch origin pull/<ID>/head:<name>
```

<ID> は #123 の数字の部分を指定します。<name> は任意の名前を指定します。

```
PS > git fetch origin pull/1/head:P1
From https://github.com/mebiusbox/re-git-sample
 * [new ref]          refs/pull/1/head -> P1
```

ブランチが作成されますので、作成したブランチに切り替えてテストできます。マージも確認したい場合、まず対象となるブランチからテスト用のブランチ(testing)を作成します。

```
PS(main) > git switch -c testing
```

次に、プルリクエストの内容をマージしてテストします。

```
PS(testing) > git merge P1
```

確認が終わってブランチが不要になったら削除します。

```
PS > git branch -D P1
PS > git branch -D testing
```

その他

➤ git describeコマンド

git describe コマンドは、そのコミットに最も近いタグの名前とそのタグからのコミット数、そしてコミットのSHA-1値の一部を使った名前を出力します。

```
PS > git describe main  
v0.2-1-g9f32671
```

➤ 対話的ステージング

git add コマンドに -i を指定すると対話形式で操作できます。すべてではなく特定のファイルのみステージングしたい場合などに使います。

```
PS > git add -i  
staged unstaged path  
1: unchanged +0/-1 TODO  
2: unchanged +1/-1 index.html  
3: unchanged +5/-1 lib/simplegit.rb  
*** Commands ***  
1: status 2: update 3: revert 4: add untracked  
5: patch 6: diff 7: quit 8: help  
What now>
```

対話形式を使えば、部分的な箇所だけをステージングできます。これは 5 の patch で行います。変更箇所(ハンク)単位でどうするかを決めることができます。

```
Stage this hunk [y,n,a,d,/,,j,J,g,e,?] ?  
y - stage this hunk  
n - do not stage this hunk  
a - stage this and all the remaining hunks in the file  
d - do not stage this hunk nor any of the remaining hunks in the file  
g - select a hunk to go to  
/ - search for a hunk matching the given regex  
j - leave this hunk undecided, see next undecided hunk  
J - leave this hunk undecided, see next hunk  
k - leave this hunk undecided, see previous undecided hunk  
K - leave this hunk undecided, see previous hunk  
s - split the current hunk into smaller hunks  
e - manually edit the current hunk  
? - print help
```

また、対話形式でなくとも、`git add -p` または `git add --patch` で同様のことができます。また、`git reset --patch` や `git checkout --patch` も可能です。

➤ 作業ツリーのお掃除

作業ツリーにある変更内容やファイルをすべて取り除きたい場合、`git clean` コマンドを使います。

```
PS > git clean -f
```

`dry-run (-n)` を使えば、削除されるものを事前に確認できます。また、実行する前に `git stash --all` で退避しておくと安全です。

`git clean` コマンドが対象とするのは追跡されておらず、かつ除外もされていないものです。`.gitignore` に合致するものは削除されません。それらも強制的に削除したい場合、`-x` を指定します。

➤ refspec

`refspec` はローカルのブランチをリモートブランチにマッピングします。

```
[+] <src>:<dst>
```

➤ Commit-ish/Tree-ish

Commit-ish/Tree-ish	Examples
1. <sha1>	dae86e1950b1277e545cee180551750029cf735
2. <describeOutput>	v1.7.4.2-679-g3bee7fb
3. <refname>	master, heads/master, refs/heads/master
4. <refname>@{<date>}	master@{yesterday}, HEAD@{5 minutes ago}
5. <refname>@{<n>}	master@{1}
6. @{<n>}	@{1}
7. @{-<n>}	@{-1}
8. <refname>@{upstream}	master@{upstream}, @{u}
9. <rev>^	HEAD^, v1.5.1^0
10. <rev>~<n>	master~3
11. <rev>^<type>	v0.99.8^commit
12. <rev>^{} 13. <rev>^{/<text>}	v0.99.8^{} HEAD^/fix nasty bug
14. :<text>	:fix nasty bug
Tree-ish only	Examples
15. <rev>:<path>	HEAD:README.txt, master:sub-directory/
Tree-ish?	Examples
16. :<n>:<path>	:0:README, :README

<https://stackoverflow.com/questions/23303549/what-are-commit-ish-and-tree-ish-in-git>

➤ detached HEAD

通常 HEAD は何かしらのブランチの先頭を参照します。しかし、例えば特定のコミットに切り替えた(switch)した場合、ブランチから切り離されます。この時のHEADの状態を detached といいます。git switch コマンドで特定のコミットに切り替える場合、--detach を指定する必要があります。

```
PS(main) > git log --oneline --decorate
9f32671 (HEAD -> main, test) add test/
0d22a3f (tag: v0.2, origin/main, origin/HEAD) add README
77da8c1 (tag: v0.1, v0.1) add poem.txt
228d01e Listing 12-2
88045a6 Reading the Argument Values
567c4c6 add Cargo.lock
3142e1b first commit
```

```
PS(main) > git switch --detach 228d01e
HEAD is now at 228d01e Listing 12-2
```

この状態からコミットをしたい場合、ブランチを作成します。

```
PS(228d01e) > git switch -c <name>
```

ブランチを作成しなくてもコミットは可能です。もしコミットしてブランチを元に戻してしまった場合、detached状態でコミットした最後のコミットに対してブランチを作成できます。また、特定のコミットに切り替えることが一時的なもので、前のHEAD状態に戻したい場合、次のコマンドで戻れます。

```
PS(228d01e) > git switch -
Previous HEAD position was 228d01e Listing 12-2
Switched to branch 'main'
```

➤ aliases

```
[alias]  
s = status -s  
co = checkout  
sw = switch  
cm = commit -m  
cma = commit --amend -m  
l = log --date=short --decorate=short --pretty=format:'%Cgreen%h %Creset%cd %C(yellow)%cn %C(cyan)%d  
%Creset%'  
la = log --all --graph --date=short --decorate=short --pretty=format:'%Cgreen%h %Creset%cd %C(yellow)%cn  
%C(cyan)%d %Creset%'  
ll = log --date=short --name-status --pretty=format:'%C(green)%h %Creset%cd %C(cyan)%an %Creset%  
%C(yellow)%d%Creset'  
lp = log -p  
lpp = log --word-diff -p  
a = add  
au = add -u  
aa = add -A  
ai = add -i  
az = archive HEAD --worktree-attributes -o  
b = branch  
unstage = reset HEAD  
uncommit = reset --soft HEAD~  
cp = cherry-pick  
d = diff -U0  
dw = diff -U0 --word-diff  
dc = diff -U0 --color-words="[^[:space:]]|[\u00c0-\uff][\u00e0-\u00ff]+\u00e0-\u00ff]+"  
ds = -c delta.side-by-side=true diff  
dd = diff --cached -U0  
dn = diff --name-only  
pu = push origin  
m = merge  
f = fetch  
su = submodule update  
sub = submodule
```