

Git Note

huyu-no-yozora

冬ノ夜空

2021 年 2 月



目次

～ Introduction ～	3
～ Notation & Assumption ～	3
1 Git とは	4
1.1 Git とは	4
1.2 環境設定	4
1.3 関連サービス	4
2 基礎概念	5
2.1 バージョン管理システム共通概念と Git 独自の概念	5
2.2 repository	5
2.3 working tree	5
2.4 commit	5
2.5 branch	6
2.6 index	7
3 基礎コマンド	8
3.1 branch の作成と checkout	8
3.2 リモートリポジトリへの変更の送信の流れ	8
3.3 その他の使用頻度の高いコマンド	10
3.4 コミット関連の整理	13
3.5 ブランチの整理	14

4	Deep Dive	14
4.1	役に立つコマンド	14
4.2	hunk 単位でのオペレーション	15
4.3	rebase の色々	15
4.4	もっと学びたい方へ	16

～ Introduction ～

Git とは、一言で言えば**ファイル群のバージョン管理を行うための仕組み**である。Linux の生みの親である Linus Torvalds によって作成された。現在でも、日々アップデートが続いている。現在では、主に、プログラムの開発者が共同でコーディングを行い開発作業を安定して進める際に使用される。このドキュメントでは、Git の使い方を簡単に学ぶためのガイド的な立ち位置のもとに、Git を使用するための基礎的な事項についてのみ記載する。Git の便利なコマンドやより深い内容などといった、より詳細なインフォメーションは、公式ドキュメントや本などで学習すると良いだろう。

～ Notation & Assumption ～

Notation に関する取り決めに先を明記しておく。

ユーザープロンプトについての notation ^{*1}

`[root]# command` : super user での実行

`[user]$ command` : normal user での実行

`command` : どちらも良い or software やインストール場所によって各々で判断

.[拡張子] 等の notation

言わずとも分かると思うが、これらの場合の `[]` はそれぞれの環境等を考慮したうえで各々で臨機応変に対応せよということである。

想定する読者としては、以下のような方を対象とする。

- バージョン管理システムとしての Git に興味があるが馴染みがない方
- これから業務で Git を使用しなければならない方
- 初めて学ぶ方
- Git の学習の仕方がわからないと悩んでいる方

このドキュメントでは主に、チームでの開発作業をできるようになることを目標とする。そのため、**リモートリポジトリを利用する想定**のもとに記述を行っていくものとする。

^{*1} 使用している Linux Distribution によっては記号が違ふことがある。

1 Git とは

1.1 Git とは

Git について

Git is a free and open source distributed version control system designed to handle everything from small to very large projects with speed and efficiency. (Git HP)

HP: <https://git-scm.com/>

1.2 環境設定

システムへのインストール後、以下コマンドを実行する。

global 設定

```
[user]$ git config --global user.name "[your_name]"
[user]$ git config --global user.email "[your_email_address]"
```

なお、特定の repository にのみ別の設定を反映したい場合には、以下のコマンドを実行する。

```
[user]$ cd "[path_to_a_local_repository]"
[user]$ git config --local user.name "[your_name]"
[user]$ git config --local user.email "[your_email_address]"
```

1.3 関連サービス

GitHub について

GitHub is a code hosting platform for version control and collaboration. It lets you and others work together on projects from anywhere.

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

GitHub Official: Feature

<https://github.com/features>

一言で言えば

ソースコードをリモートに保存し、必要に応じて公開することができるサービス

である。CI/CD を組んだりできるようになったことから、インフラ基盤として利用されることも多い。

GitHub のチュートリアル

<https://guides.github.com/activities/hello-world/>

2 基礎概念

2.1 バージョン管理システム共通概念と Git 独自の概念

2.1.1 共通概念

- repository
- commit
- working tree
- branch
- checkout

2.1.2 Git 独自の概念

Git の場合、working tree と repository の間に、index が存在する。これを利用して、ファイルの一部のみを index に登録し、commit できるというメリットがある。(4.2 を参照。)

2.2 repository

repository について

repository とは、管理したいソースコードファイルおよびその履歴の保管庫のことである。

ローカルリポジトリの中身のファイル群

- **.git** : git repository に関する設定ファイル等が格納されたディレクトリ
- **README.md**
- ソースコード

2.3 working tree

working tree について

working tree とは、local repository において、repository から取り出す際に展開されるための場所のこと。local repository においては、.git ディレクトリが存在するディレクトリのことを指す。ユーザは working tree 内で作業を行い、アップデート/修正の内容を repository に反映していく。

2.4 commit

commit について

commit とは、repository に修正内容を反映していくオペレーション、また、それによって repository へ反映された履歴のことを指す。

2.5 branch

2.5.1 What is branch?

branch について

ブランチは、主に最新の commit を追っていくための label のようなもの。branch は追っていった commit の commit 履歴を情報として持っている。

HP: <https://git-scm.com/book/en/v2/Git-Branching-Branched-in-a-Nutshell>

よく知られた branch には以下のようなものがある。

- **master** : 本番 (リリース) 用ブランチ
- **develop**: 開発用ブランチ (source branch: master)
- **feature**: 実際に (source branch: develop)
- **hotfix** : 緊急性の高い bug の修正を行う際に使用
- **release**: リリース準備用の branch (source branch: develop)

2.5.2 branch の種類

branch の種類

Git における branch には大きく、local branch, remote-tracking branch, remote branch の3種がある。

local branch

3.1 Git Branching - Branches in a Nutshell

<https://git-scm.com/book/en/v2/Git-Branching-Branched-in-a-Nutshell>

A branch in Git is simply a lightweight movable pointer to one of these commits. The default branch name in Git is master. As you start making commits, you're given a master branch that points to the last commit you made. Every time you commit, the master branch pointer moves forward automatically.

remote branch

3.5 Git Branching - Remote Branches

<https://git-scm.com/book/en/v2/Git-Branching-Remote-Branched>

Remote references are references (pointers) in your remote repositories, including branches, tags, and so on. You can get a full list of remote references explicitly with `git ls-remote <remote>`, or `git remote show <remote>` for remote branches as well as more information. Nevertheless, a more common way is to take advantage of remote-tracking branches.

remote-tracking branch

3.5 Git Branching - Remote Branches

Remote-tracking branches are references to the state of remote branches. They're local references that you can't move; Git moves them for you whenever you do any network communication, to make sure they accurately represent the state of the remote repository. Think of them as bookmarks, to remind you where the branches in your remote repositories were the last time you connected to them.

2.5.3 tracking branch と upstream branch

Tracking Branches

Checking out a local branch from a remote-tracking branch automatically creates what is called a "tracking branch" (and the branch it tracks is called an "upstream branch"). Tracking branches are local branches that have a direct relationship to a remote branch. If you're on a tracking branch and type `git pull`, Git automatically knows which server to fetch from and which branch to merge in.

... The simple case is the example you just saw, running `git checkout -b <branch> <remote>/<branch>`. This is a common enough operation that Git provides the `--track` shorthand:

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

上流ブランチ/下流ブランチ

upstream branch(上流ブランチ) とは、リモート側との～。

上流ブランチの確認

```
[user]$ git branch -vv
```

上流ブランチを登録しつつ、リモートリポジトリへ push

```
[user]$ git push --set-upstream origin feature/exampleBranch
```

2.6 index

index とは

index とは、cache のようなもの。次の commit 対象となるファイルの一時保管場所。
commit する場合には、まず index に管理対象のファイルを登録する。

3 基礎コマンド

3.1 branch の作成と checkout

checkout

checkout とは、作業フォルダの状態を特定の save point(commit) の状態に復元することである。checkout を実行することにより、repository から、branch のデータを index および working tree に展開される。

(基本的には branch に対して checkout を行うことが多い。)

まず、以下コマンドで branch を作成する。

```
[user]$ git branch [branch name]
```

作成した branch に checkout してみる。

```
[user]$ git checkout [branch name]
```

また、以下のコマンドで、上記のローカル branch の作成と checkout をまとめて行える。

```
[user]$ git checkout -b [branch name]
```

ここでは、develop ブランチを作成し、checkout することにする。

```
[user]$ git branch develop  
[user]$ git checkout develop
```

Column: Git の current branch を表示させる

どこに checkout しているのか知るために、以下の環境変数を変更しておくの良いかもしれない。シェル起動スクリプト (login script: ~/.bash_profile 等) に以下を設定。^a

```
PS1="\u@\h_\[\033[32m\]\W\[\033[00m\]]\[\033[35m\]\$ (parse_git_branch)\[\033[00m\]]\$_"  
export PS1
```

^a ここでは、Mac, Linux, RHEL 等を想定しかつ、デフォルトシェルとして bash の利用を想定。

3.2 リモートリポジトリへの変更の送信の流れ

リモートリポジトリへの変更のアップロード手順に関しては、主に以下の Step から成る。

Git Main Process

- Step1. git add によるインデックスへのファイルの追加等
- Step2. git commit によるインデックスへの変更の反映
- Step3. git push による変更内容のリモトリポジトリへのアップロード

以下のようなファイルを Git で管理したいとする。

```
1 #include <stdio.h>
2 int main() {
3     // printf() displays the string inside quotation
4     printf("Hello,_World!");
5     return 0;
6 }
```

3.2.1 Step1. add - index に登録する

以下コマンドを実行し、Git の管理下にあるファイルを index に追加する。

```
[user]$ git add .
```

3.2.2 Step2. commit - index に変更を反映する

変更内容を index に反映（保存）するには、以下のようなコマンドを使用する。

```
[user]$ git commit -m "[comment]"
```

ここでは、`\$ git commit -m "initial_commit"` として一度 commit したとしよう。
その後、以下のように編集したとする。

```
1 #include <stdio.h>
2 int main() {
3     // printf() displays the string inside quotation
4     printf("Good_Bye!");
5     return 0;
6 }
```

再度 `git add .` を実行し、index に追加し、その後、再度 commit を行う。その際、修正内容をコメントとしてつけて commit を行う。ここでは、`git commit -m "change_print_message"` として commit しておこう。

リモートに反映させたい修正の範囲になるまで、Step1, Step2 を繰り返す。

3.2.3 Step3. push - リモートへの変更内容の送信

リモートに変更内容を送信するには、以下のコマンドを実行する。

```
[user]$ git push origin [branch name]
```

ここでは、リモートリポジトリに develop ブランチを push するでしょう。

```
git push origin develop
```

異なる branch 名で push する場合

```
git push origin [local branch name]:[remote branch name]
```

3.3 その他の使用頻度の高いコマンド

3.3.1 リモートの変更内容の反映

```
[user]$ git pull
```

remote repository に変更内容をアップロードする際には、事前に上記コマンドを実行する。その後、origin/develop の最新 commit が、自分がこれから remote repository に対して送信/更新するブランチの source commit より先に進んでいるか確認する。

なお、pull は、裏側の処理としては fetch^{*2} と merge との組み合わせである。

3.3.2 状態の確認

ローカルリポジトリ、ワーキングツリーの状態の確認

```
[user]$ git status
```

コミット履歴等の確認

```
[user]$ git log
```

3.3.3 変更の取り消し（ローカルブランチに対してのみ推奨）

`git reset` コマンドでは、特定のコミットによる変更を戻すことができる。なお、デフォルトのオプションは `--mixed`（HEAD とインデックスのリセット）である。

ここでは、ローカルリポジトリのワーキングツリーやインデックス、ブランチを最新のコミットの内容にまで戻す場合について記載する。

```
[user]$ git reset --hard HEAD
```

または、ワーキングツリーの内容を最新のコミットの状態に戻すには、以下のコマンドでも良い。

```
[user]$ git stash
[user]$ git checkout HEAD
```

3.3.4 一時保存

```
[user]$ git stash (save [saving name])
```

^{*2} fetch については、remote repository から対象ブランチの最新までの変更内容をダウンロードするためのオペレーション（コマンド）である。

fetch: <https://git-scm.com/docs/git-fetch>

このドキュメントは introduction の範囲内での記載に留めるため、fetch については、このドキュメントでは取り扱わないものとする。

保存されているキューの一覧の表示

```
[user]$ git stash list
```

キューに保存された内容の表示

```
[user]$ git stash show -p stash@[queue number]
```

保存したキューの内容を適用

```
[user]$ git stash apply [stash@{stash[queue number]} or saved name]
```

保存されたキューの内容を削除

```
[user]$ git stash drop [stash@{stash[queue number]} or saved name]
```

3.3.5 Rebase

feature/example ブランチを develop ブランチに追従させる場合

```
graph TD
    18d3438((18d3438: (master) initial code was putted)) --> 561e128((561e128: add a Contents table))
    561e128 --> e5ad949((e5ad949: change a subsection name))
    e5ad949 --> be8634b((be8634b: (feature/example) add description about how to delete a branch))
    be8634b --> ef963ca((ef963ca: (HEAD -> feature/example) add description about how to delete a branch))
    69119b1((69119b1: add description of some useful command)) --> ef963ca
```

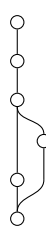
以下を実行する。

```
[user]$ git checkout feature/example
[user]$ git rebase develop
```

```
graph TD
    18d3438((18d3438: (master) initial code was putted)) --> 561e128((561e128: add a Contents table))
    561e128 --> 69119b1((69119b1: add description of some useful command))
    69119b1 --> cfb44f5((cfb44f5: (develop) add description about hunk))
    cfb44f5 --> bfb0835((bfb0835: change a subsection name))
    bfb0835 --> ef963ca((ef963ca: (HEAD -> feature/example) add description about how to delete a branch))
```

3.3.6 Merge

```
[user]$ git checkout [source branch]
[user]$
[user]$ # 3 way - merge
[user]$ git merge --no-ff [target branch]
[user]$
[user]$ # fast forward merge (default)
[user]$ git merge [target branch]
```



```
graph TD
    A((d764b48: added plaintext version in markdown))
    B((54ba4b2: release 2014-01-25))
    C((c589395: Merge branch 'master'))
    D((9f9c652: Remove holdover from kjh gh-pages branch))
    E((b3bd158: exclude font files))
    F((63268c1: micro-typography))
    A --- B
    B --- C
    C --- D
    C --- E
    E --- F
```

d764b48: added plaintext version in markdown
54ba4b2: release 2014-01-25
c589395: Merge branch 'master'
9f9c652: Remove holdover from kjh gh-pages branch
b3bd158: exclude font files
63268c1: micro-typography

3 way - merge と fast forward merge の使い分け

fast forward merge を統合ブランチに対して実施すると、そのブランチの commit log に対して、統合される branch で行なった一つ一つの細かい commit 履歴 (およびコメント) がそのまま取り込まれてしまう。一方、3 way - merge を使用すると、merge される branch 内部の commit 履歴の一つ一つが表示されないようにできる。

統合 branch との merge を行う際には、3 way - merge を使用することを推奨する。

rebase と merge の使い分け

Merge は、現在 checkout している branch に別の branch の変更内容を取り込むことができる。但し、merge 単体での運用だと、merge するまでの時間が長いほど、その branch で行なった作業の commit 履歴同士が離れていくので、導入した作業の内容や流れが追いにくなる。

Rebase は、リモート側の最新を取得 (fetch / pull) した上で、source branch が進んでいたら、その最新の commit に対して新しく自分の行なった変更を付け足していく。そのため、自身の行なった作業の commit (メッセージ) が連続して (作業内容が 1 箇所にまとまって) 表示されるため、commit graph が綺麗になる。

その時々で、「シンプルな commit graph を作るために、今の場合 merge と rebase のどちらを使うべきだろう」という視点で考えるようにすると良いだろう。

以下のような運用をお勧めする。

基本的には remote に branch を push する前に、remote の変更内容を取得し、rebase を行なってから、push するようにする。

3.4 コミット関連の整理

3.4.1 コミットメッセージの変更

直前の commit message を変更するには以下を実行すれば良い。

```
[user]$ git commit --amend
```

2 つ以上前の commit message を変更する場合には以下ようになる。

2 つ以上前の commit message の変更

n 個前の commit message を変更したいとする。

Step1. 変更対象の branch に checkout し、以下を実行する

```
[user]$ git rebase --interactive HEAD~n
```

Step2. 変更したい対象の commit の commit id の隣の「pick」という文字を「edit」に書き換え最初の edit の部分に戻っているので、以下を実行する

```
[user]$ git commit --amend
```

Step3. commit message を変更後、以下を実行する

```
[user]$ git rebase --continue
```

Step4. 以降、edit に書き換えた部分に対し、Step2, Step3 を繰り返す

Step5. git log を実行し、変更が反映されていることを確認する

3.4.2 コミット内容の確認

特定の commit の変更内容を表示するには、以下のような形で実行する。

```
[user]$ git show [commit id]
```

最新の commit の内容を確認・表示するには、以下のようにすれば良い。

```
[user]$ git show HEAD
```

3.4.3 コミットの統合

現在 checkout している branch にてコミットの統合を行う場合、git rebase コマンドを使用して統合する。

rebase を使用したコミットの統合

n 個前までのコミットをまとめたいとする。

Step1. 変更対象の branch に checkout し、以下を実行する

```
[user]$ git rebase --interactive HEAD~n
```

Step2. 統合先のコミット (初めのコミット) は「pick」のまま、統合対象の commit id の隣の「pick」を「s」に書き換え

Step3. 統合した際の commit message として、コミットメッセージを修正する

Step4. git log を実行し、変更が反映されていることを確認する

その他、merge 時にコミットを統合することもできる。

```
[user]$ git merge --squash [merged branch]
```

3.5 ブランチの整理

3.5.1 ブランチの確認

以下コマンドで、local repository のブランチを確認できる。

```
[user]$ git branch -a
```

3.5.2 ブランチの削除

branch の削除の仕方については以下の 2 種類がある。

元になっているブランチよりも新しい commit がある場合には、d オプションでは error を返す。

```
[user]$ git branch -d [target branch]
[user]$ git branch -D [target branch]    # by force
```

4 Deep Dive

4.1 役に立つコマンド

4.1.1 cherry-pick

他のブランチなどにある commit も取り込むことができる。

取り込みたい branch に checkout し、その後、取り込みたい commit id を羅列するだけで良い。

```
[user]$ git checkout [target branch]
[user]$ git cherry-pick [target commit 1] [target commit 2] ...
```

詳細は以下を参照。cherry-pick: <https://git-scm.com/docs/git-cherry-pick>

4.1.2 tag

特定の commit に対して v1.0 というタグをつけたい場合は以下のように実行する。

```
[user]$ git tag v1.0 [target commit id]
[user]$ git tag -m "version_1.0" v1.0 [target commit id] # with a comment
```

現在のタグを確認する場合は、以下を実行すれば良い。

```
[user]$ git tag
```

詳細は、以下を参照。tag: <https://git-scm.com/book/en/v2/Git-Basics-Tagging>

4.1.3 clean

```
[user]$ git clean -n
[user]$ git clean -f
```

4.1.4 archive

```
$ git archive --format=tar --prefix=[directory name] HEAD | gzip > [directory path]/[file name].tar.gz
```

prefix オプションで、展開後のディレクトリを指定できる。

4.2 hunk 単位でのオペレーション

hunk とは

管理対象のファイルに対し、Git はブロック単位でコードを認識している。hunk とは、コード内における Git が認識するブロックのことである。

以下では hunk 単位での扱いについて記載する。

hunk 単位での index への登録

```
[user]$ git add -p
```

hunk 単位での commit

```
[user]$ git commit --interactive
```

4.3 rebase の色々

4.3.1 merge commit を含めて rebase

単純な `git rebase -i` だとマージコミットを含めずに rebase を行うことになり、rebase 後には merge commit が消えてしまう。これに対処するためには、`-rebase-merges` オプションを使用する。

```
[user]$ git rebase -i --rebase-merges HEAD~n
```

但し、マージコミットのブランチ名などは失われてしまうことに注意。

4.3.2 commit, branch の移動

src から end-rev までの commit(但し、src の commit は含まれない) を destination に対して移動する。

```
[user]$ git rebase --onto=destination src end-rev
```

※ src, end-rev, destination は revision を指定

もとの commit を維持したい場合 (移動ではなく、付与) は、cherry-pick を使用する。

誤って、`\$ git rebase --onto=destination end-rev` とすると、src や、それ以前のコミットなども含まれてしまうので注意！

4.3.3 過去のコミットの改変

```
[user]$ git rebase -i HEAD~n
```

対象のコミットを pick → edit に編集し、変更したい作業を実施する。

```
[user]$ git add .  
[user]$ git commit --amend  
[user]$ git rebase --continue
```

4.4 もっと学びたい方へ

Git Branch Model

Git Branch Model というものを学習すると良いだろう。Git-Flow, GitHub-Flow などが存在する。

<https://nvie.com/posts/a-successful-git-branching-model/>

ProGit

ProGit という本が Git 本家から出ているので、より詳しく学びたい場合は一読すると良いだろう。

Git HP: <https://git-scm.com/book/en/v2> GitHub: <https://github.com/progit/>