Git によるバージョン管理入門

田中 健策（株式会社ぺあのしすてむ）

第五回（最終回）

前回までのあらすじと今回のあらすじ

前回までのあらすじ

- プルリクエストを使ったブランチのレビューとマージの自動化と記録
- Github Actions を使ったコンパイルとリリースの自動化

今回は Git の内部構造といくつかの細かい機能について説明する。これを知っておくと Git におけるトラブルシューティングに役に立つからだ。

Git の内部構造　1

Git の内部構造は全て.git ディレクトリにある。
hooks ディレクトリには以前説明した git フックのスクリプトが入っている。
config ファイルにはこのリポジトリ独自の設定が入っている。
index ファイルには git add されて、これから記録されるファイルが登録されている。
特に重要なのは

- objects ディレクトリ
- refs ディレクトリ
- HEAD ファイル

である。

Gitの内部構造　2

objectsにはgitに記録された全てのコミットやフォルダやファイルの情報が圧縮されて保存されている。それぞれのobjectにはそのデータのhash値が名前として割り当てられている（フォルダ名とファイル名になっている）。
refsディレクトリには、branchの先頭やtagとして登録されたコミット等のobjectsが収納されている。
HEADファイルは、最新のコミットのobjectが収納している。
git switch（checkout）によってHEADの位置が変わり、その位置のコミットの指し示すフォルダのobjectがgitで管理されたフォルダ（ワーキングツリーと呼ぶ）に、展開される。

Git の内部構造　3

git commit をすると、index に登録されたファイルやフォルダの情報が圧縮されて git の objects に加えられる。またフォルダやコメント等のコミット情報も圧縮されて objects に加えられるが、その際、現在の HEAD がそのコミットの親として登録される。そして、refs や HEAD が動かされて、コミットが完了する。

失われた時を求めて　1

HEAD が branch の先頭を見ていれば、git commit をしても、branch の先頭の ref が動き、HEAD はそのブランチの ref を見ているだけである。しかし、git switch（checkout）で、branch の途中を HEAD が指している時（このようにブランチの先頭以外を HEAD が見ることを detached HEAD と呼ぶ）に、commit すると、HEAD はその加えられたコミットの object を見るが、ブランチの先頭の ref は動かない。
その状態で、他のブランチに移動してしまうと、先ほどのコミットはどの ref からも届かない object になってしまう。つまり歴史が消えてしまったのだ。
その他にも git reset –hard（HEAD の位置、インデックス、ワーキングツリーなどが全て戻される）などをして、意図的に歴史を消去することもできる。

失われた時を求めて　2

しかし消えてしまった記録も、しばらくは objects の中に残っている。そしてかつて HEAD で指し示されていた object の記録を git reflog で調べられる。
これを使えば、消えてしまった記録を復元できる。
復元するためのコマンドは git cherrypick である。これを使えば、あるコミットの変更を自分のワーキングツリーに反映できる。

変更の一時待避

git がファイルやフォルダの情報を保存する仕組みを使えば、コミットせずにその状態を一時的に記録し、あとで再現することができる。
それが git stash である。
急に対応をしなくてはいけない変更などがある場合に便利だ。
コミットしていない変更がある状態で、

git stash

とコマンドすると、変更が退避される。

git stash list

とコマンドすることで退避した作業の一覧が見られ、

git stash apply <stash の番号>

で退避した作業を元に戻すことができる。

git worktree

.git の仕組みを使うと、面白いことができる。
例えば git worktree コマンドでは、git の特定のブランチ用のワークスペースを、現在のワーキングツリーとは別に用意できる。
git worktree add <作業フォルダ> <ブランチ名>
とすると、その場所にそのブランチを展開した作業フォルダができる。これにより、いちいちブランチを変更しなくても、複数のブランチで同時に作業ができる。
そのフォルダの中には.git フォルダではなく、.git ファイルがある。この.git ファイルは、単に元のワーキングツリーの.git フォルダを参照しているだけのファイルで、この新しい作業フォルダで git を操作すると、元の.git フォルダに影響与えられる。
これにより、混乱することなく、二つの作業フォルダで一つの git リポジトリを操作することができる。

git submodule 1

これを応用すると、さらに面白いことができる。git リポジトリの中の一部に、別の git リポジトリを取り込むことができる、git submodule という機能である。
git submodule add <git リポジトリの URL> <フォルダ名>
とすると、現在のリポジトリの中に、別の git リポジトリがサブモジュールとして取り込まれる。
そして、現在の git リポジトリには.gitmodules というファイルができており、これは、サブモジュールの.git ディレクトリを参照している。

git submodule 2

サブモジュールの.git ディレクトリが更新されると、外側の git リポジトリはその更新を感知することができ、.gitmodules を更新しようとする。
それによって、外側の git リポジトリはサブモジュールがどのコミットを指し示せばいいかを記録できるので、
複数のローカルリポジトリで同じサブモジュールのコミットを参照できるのだ（もしこの仕組みがないと、ローカルリポジトリごとにサブモジュールの別のコミットを取り込んでしまうかもしれない。そうすると、取り込んだ時期によって、挙動が異なったりすることがありうる）。
これは大きなプロジェクトでは、是非とも使いこなしたい機能である。

便利なログ

.git の log ディレクトリには、そのブランチのログが収納されている。git log は様々なオプションをつけることで、得たい情報を上手く収集することができる。

- 特定のファイルの変更を追いかける
- 特定のファイルがいつ消されたかを特定する

などができる。
欲しい機能は大体あるので、悩まずに検索してみよう。
また git bisect というコマンドを使うと、問題が起こったコミットを binary search を使って効率的に探すことができる。