Git のサブモジュールで困ったら読む本

mochikoAsTech　著

2020-09-12 版　　mochikoAsTech 発行

はじめに

2020 年 9 月 mochikoAsTech

本書を手に取ってくださったあなた、こんにちは！ あるいは、はじめまして。「Git のサブモジュールで困ったら読む本」の筆者、mochikoAsTech です。

本書は 2020 年 9 月にオンラインにて開催される「技術書典 9」に向けて書いた新刊です。8 月 24 日に「あと 20 日しかないのに新刊を書こうとしはじめました！」と Slack で宣言し、「デザインも必要になります…？ よね…？」と首をかしげるデザイナーさんに向かって元気よく「うん！」とこたえ、みんなで真夏のぎりぎり〆切に向かって楽しく駆け抜けた思い出の一冊です。

COVID-19 の影響で、2020 年の技術書典はすべて、オフライン（会場）からオンラインでの開催へと切り替わりました。「物理的なアクセスはなくなりますか？」と聞いた西乃園萌絵に対して、真賀田四季は「そうね、おそらく、宝石のように贅沢品になるでしょう。他人と実際に握手をすることでさえ、特別なことになる。人と人が触れ合うような機会は、贅沢品です。」と答えました。[*1] 私たちは情報をわざわざ紙に印刷し、大勢で一箇所に集まって本を手渡すという、今となっては得がたい宝石のような贅沢を楽しんでいたのだな、と感じます。

オンラインになったことで、場所や距離の制約はなくなり、東京から離れたところにお住まいの方にも本が届けやすくなりました。それはもちろん嬉しいことですし、失ったものより新たに得たものを数えた方が人は楽しく生きられます。けれど水の中では煙草は吸えないので、いつかまたオフラインで皆さんに本を届けられる日を楽しみに、日々を生き抜こうと思います。

本書が、サブモジュールでお困りになっているあなたのお役に立てば幸いです。

想定する読者層

本書は、こんな人に向けて書かれています。

[*1] 「すべてが F になる (講談社文庫) 森 博嗣」P21 より引用。https://www.amazon.co.jp/dp/4062639246

- 開発中の Git リポジトリにサブモジュールがある
- よく分からないままなんとなくサブモジュールを使っている
- サブモジュールで困った経験がある
- 何もしていないはずなのにサブモジュールが勝手に更新された
- `git pull` しただけでサブモジュールの差分が出た
- 別のブランチへ移動しただけでサブモジュールがなぜか更新された
- `git pull` したのにサブモジュールの中身が更新されない
- `git submodule update` したときと、サブモジュールのディレクトリで `git pull` したときの違いが分からない

マッチしない読者層

本書は、こんな人が読むと恐らく「not for me だった…（私向けじゃなかった）」となります。

- Git について何も知らないので 1 から学びたい
- サブモジュールを導入すべきか迷っていて判断材料が欲しい

本書のゴール

本書を読み終わると、あなたはこのような状態になっています。

- サブモジュールの仕組みが分かっている
- サブモジュールが意図せず更新されたときに対処できる
- サブモジュールのトラブルを自力で解決できる
- 読む前よりサブモジュールがちょっと好きになっている

免責事項

本書に記載されている内容は筆者の所属する組織の公式見解ではありません。また本書はできるだけ正確を期すように努めましたが、筆者が内容を保証するものではありません。よって本書の記載内容に基づいて読者が行なった行為、及び読者が被った損害について筆者は何ら責任を負うものではありません。不正確あるいは誤認と思われる箇所がありましたら、必要に応じて適宜改訂を行いますので GitHub の Issue や Pull request で筆者までお知らせいただけますと幸いです。

`https://github.com/mochikoAsTech/SubmoduleUpdated`

目次

第１章

サブモジュールとは？

ああサブモジュールよ、君の挙動が分からないんだ…！

1.1 Git とは

本書は Git のサブモジュールの挙動が分からずに苦しむ人のための本なので、Git とはなにか？ についてや、`git pull`、`git clone` といった初歩的な git コマンドについては解説しません。「Git について何も知らないので 1 から学びたい」という方には、湊川あいさん[*1]の書籍がお勧めです。

- わかばちゃんと学ぶ Git 使い方入門
  - `https://www.amazon.co.jp/dp/4863542178`

「お金をかけずにまずは無料で学びたい」という場合は、先ほどの本の元となったウェブ連載を読みましょう。書籍内容の序盤が体験できます。

- マンガでわかる Git 第 1 話「Git ってなあに？」
  - `https://next.rikunabi.com/journal/20160526_t12_iq/`

さらに最近は、SourceTree のように GUI を通して Git を使う方法だけでなく、コマンドで Git を使う方法が学べる「コマンド編」も連載されているそうです。

- マンガでわかる Git ～コマンド編～ 第 1 話「リポジトリを作ってコミットしてみよう」
  - `https://www.r-staffing.co.jp/engineer/entry/20190621_1`

Git は付け焼き刃の操作だけを学ぶよりも、どういう仕組みで、どんな理屈で動いているのかをしっかり学んだ方が、結果としては理解の速度が上がります。[*2]わかばちゃんと一緒にたくさん転んで、Git を楽しく学んでみてください。

1.2 Git のサブモジュールとは

サブモジュールとは、Git の機能のひとつです。サブモジュールを使えば、あるプロジェクトのリポジトリを、別のリポジトリのサブディレクトリとして扱えるようになります。

急に「サブディレクトリとして扱える」と言われてもピンとこないと思うので、サブモジュールの便利な使用例をご紹介しましょう。

[*1] `https://twitter.com/llminatoll`

[*2] Git に限らず、どんなこともそうですよね。分かってはいるけれど、困ったらつい「Git いつ更新調べる」のようにやりたいことベースで検索して、出てきたコマンドを叩いて、「なぜかは分からないけどできた！」みたいなことをしてしまうので、自分で書いていて耳が痛いです。

1.3 サブモジュールの便利な使用例

筆者は、技術書の原稿を Git のリポジトリで管理しています。この原稿リポジトリの中には、実際の原稿ファイルや画像ファイルだけでなく、prh[*3]という校正ツールがあり、その中には次のような「表記揺れを自動チェックするための正誤表」も含まれています。

```
- expected: 筆者
  pattern:  著者
- expected: 本書
  pattern:
    - この本
    - 本著
- expected: つたえる
  pattern:  伝える
- expected: 分かり
  pattern:  わかり
```

筆者は本を一冊書くたびに、少しずつこの正誤表に新しい内容を追記しています。そのため新しい本の原稿リポジトリを作るときには、毎回ひとつ前の原稿リポジトリから、正誤表を含む校正ツールのディレクトリ[*4]（`prh-rules`）をまるごとコピーしてくる必要がありました。

原稿リポジトリを作るたびに、いちいちコピーしてくるのは面倒です。さらに同時並行で色んな原稿を書いていると、あちらでの変更をこちらに持って来たり、今度はこちらでの変更をあちらに持って行ったりと、コピーペーストを繰り返す羽目になります。微妙に内容の違う正誤表が、古い原稿リポジトリに残っているのも気分的によくありません。ああ、**校正ツールのディレクトリだけ別リポジトリに切り出せたら**いいのに…！

そんなときに便利なのがサブモジュールです！ 校正ツールのディレクトリ（`prh-rules`）だけをひとつのリポジトリとして切り出しておき、それぞれの原稿リポジトリでサブモジュールとして指定してやればいいのです。

先ほど「サブモジュールを使えば、あるプロジェクトのリポジトリを、別のリポジトリのサブディレクトリとして扱えるようになります。」と説明しましたが、これを実態に即した形にすると「サブモジュールを使えば、校正ツールのリポジトリを、それぞれの原稿リポジトリのサブディレクトリとして扱えるようになります。」となり

[*3] ProofReading Helper の頭文字で prh です。`https://github.com/prh/prh`

[*4] フォルダのこと。Windows や Mac ではフォルダと呼ぶ方が馴染みがあると思いますが、本書では Git のドキュメントや Linux に倣ってディレクトリと呼びます。

ます。

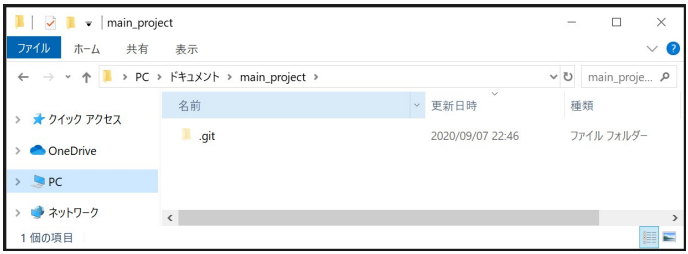
サブモジュールは便利な一方で、構造が複雑になるので、思わぬトラブルの元となることがあります。なんとなくではなく、前述のように「たしかに！ これは便利だ！」と思えるときに使いましょう。

1.4 サブモジュールを使ってみよう

サブモジュールを理解するには、使ってみるのがいちばんです。まずはサブモジュールを使うための、メインのリポジトリから作ってみましょう。次の `git init main_project` というコマンドを叩くと、`main_project` というリポジトリのディレクトリが生成されます。

```
メインのリポジトリ（main_project）を作る
$ git init main_project
Initialized empty Git repository
 in C:/Users/mochikoAsTech/Documents/main_project/.git/
```

main_project が生成されました。（図 1.1）[*5]



▲図 1.1　main_project というリポジトリのディレクトリができた

続いてメインリポジトリのサブモジュールとして、既に GitHub 上に存在している別のリポジトリ[*6]を追加してみましょう。

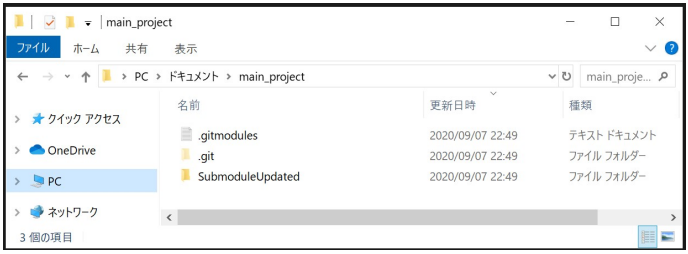
[*5] `.git` が表示されない場合は、おそらく「ドットからはじまるファイルやフォルダは非表示」という設定になっています。Windows なら「隠しファイルを表示する」にチェックを入れてください。Mac なら `Command+Shift+.`（ドット）を押すことで表示されるようになります。

[*6] サブモジュールとして追加したのは、あなたが今読んでいるこの本の原稿リポジトリです。`https://github.com/mochikoAsTech/SubmoduleUpdated`

```
作ったメインリポジトリのディレクトリに移動する
$ cd main_project

サブモジュールとして「SubmoduleUpdated」というリポジトリを追加する
$ git submodule add https://github.com/mochikoAsTech/SubmoduleUpdated
```

今回はサブモジュールとして、本書の原稿リポジトリを追加してみました。（図 1.2）[*7]



▲図 1.2　SubmoduleUpdated がサブモジュールとして追加された

サブモジュールを追加すると、自動的に追加した SubmoduleUpdated の中身をクローンしてくるため、こんな表示がされたと思います。

```
Cloning into
 'C:/Users/mochikoAsTech/Documents/main_project/SubmoduleUpdated'...
remote: Enumerating objects: 251, done.
remote: Counting objects: 100% (251/251), done.
remote: Compressing objects: 100% (213/213), done.
Receiving objects:  97% (244/251), 2.25 MiB | 458.00 KiB/s(delta 103),
 reused 76 (delta 26), pack-reused 0
Receiving objects: 100% (251/251), 2.29 MiB | 455.00 KiB/s, done.
Resolving deltas: 100% (103/103), done.
```

このときサブモジュール（SubmoduleUpdated）の中身はクローンしてきますが、サブモジュールのさらにサブモジュール（prh-rules）以下については再帰的にはク

[*7] 今回はサブモジュールとして GitHub の URL を指定しましたが、このような絶対パスの URL に限らず、サブモジュールには `git submodule add ../SubmoduleUpdated` のような相対パスでローカルのリポジトリを指定することも可能です。ただし相対パスで追加すると、サブモジュールのリモートリポジトリの URL（`remote.origin.url`）が `C:/Users/mochikoAsTech/Documents/SubmoduleUpdated` や `../SubmoduleUpdated` のようになります。特に理由が無ければリモートの URL で指定しておきましょう。

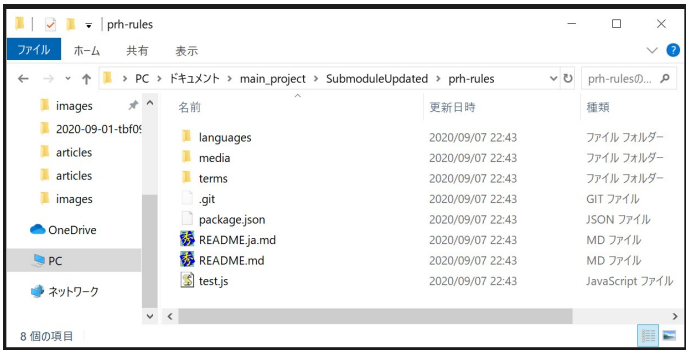
ローンしてきてくれません。実際に main_project/SubmoduleUpdated/prh-rules を見てみると、中身はまだ空っぽです。（図 1.3）

▲図 1.3　サブモジュールのサブモジュールはまだ中身が空っぽ

サブモジュールのサブモジュール以下についても、すべて中身を連れてきたい場合は、次のコマンドでサブモジュールを再帰的に初期化しておきましょう。このコマンドは、メインリポジトリである main_project ディレクトリで実行します。SubmoduleUpdated ディレクトリや prh-rules ディレクトリに移動して実行してはいけません。

```
サブモジュールを再帰的に初期化する
$ git submodule update --init --recursive
Submodule 'prh-rules' (https://github.com/mochikoAsTech/prh-rules)
 registered for path 'SubmoduleUpdated/prh-rules'
Cloning into
 'C:/Users/mochikoAsTech/Documents/main_project/SubmoduleUpdated/prh-rules'...
Submodule path 'SubmoduleUpdated/prh-rules': checked out
 'ec6d80a111881e28c6e8e5129cfa6a49b995830b'
```

サブモジュールを再帰的に初期化したことで、サブモジュールのさらにサブモジュール（prh-rules）の中身もクローンできました。（図 1.4）



▲図 1.4　サブモジュールのサブモジュールの中身も連れてこられた

なおサブモジュールを追加するときには、先ほどのように特にディレクトリ名を指定しなければ、サブモジュールのリポジトリ名（SubmoduleUpdated）がそのままディレクトリ名となります。ディレクトリ名を変えたいときは、次のように末尾でディレクトリ名（sub）を指定します。するとディレクトリ名を「sub」にした状態でサブモジュールを追加できます。

```
パターンA．サブモジュールとして「SubmoduleUpdated」を追加する
$ git submodule add https://github.com/mochikoAsTech/SubmoduleUpdated

パターンB．ディレクトリ名を「sub」にした状態でサブモジュールを追加する
$ git submodule add https://github.com/mochikoAsTech/SubmoduleUpdated sub
```

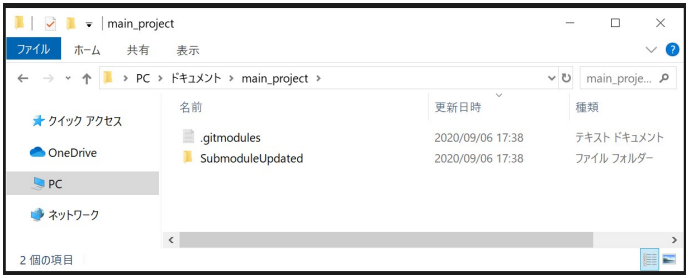
今回はパターン A のように「ディレクトリ名は特に指定しなかった」という前提で話を進めます。サブモジュールを追加してどうなったのか、git status でメインリポジトリの状態を確認してみましょう。

```
$ git status
On branch master

No commits yet

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

.gitmodules と SubmoduleUpdated の 2 つが、新しいファイルとして認識されています。（図 1.5）



▲図 1.5　.gitmodules と SubmoduleUpdated の 2 つが新しいファイルとして認識されている

サブモジュールを追加すると、このようにメインのリポジトリに.gitmodules というファイルが生まれます。これはテキスト形式の Git の設定ファイルです。テキストエディタで開くとこんなふうに、サブモジュールのディレクトリパスと、リモートの URL が書かれています。サブモジュールを複数追加した場合は、このファイルにサブモジュールの数だけ追記されていきます。

```
[submodule "SubmoduleUpdated"]
        path = SubmoduleUpdated
        url = https://github.com/mochikoAsTech/SubmoduleUpdated
```

あなた以外の誰かが、この「サブモジュールを使っているメインリポジトリ」（main_project）をクローンした場合、Git はこの.gitmodules というファイルに書かれた内容を元に、サブモジュールのパスや取得元を把握することになります。

続いて git diff コマンドでサブモジュール（SubmoduleUpdated）の変更前と変更後の差分を見てみましょう。まだコミットしていないファイルの差分が見たいので、--cached オプションを付ける必要があります。変更前の--- /dev/null は、このファイルが新たに作られたものであることを表しています。

```
$ git diff --cached SubmoduleUpdated
diff --git a/SubmoduleUpdated b/SubmoduleUpdated
new file mode 160000
index 0000000..6f47087
```

```
--- /dev/null
+++ b/SubmoduleUpdated
@@ -0,0 +1 @@
+Subproject commit 6f47087f1c9079ea6c677702da23ca040d0a13ed
```

実際は SubmoduleUpdated はディレクトリであり、その中にはたくさんの原稿ファイルがあります。ですがメインのリポジトリからは、サブモジュールの中身を 1 つ 1 つ追跡するようなことはしません。代わりにこのサブディレクトリを、+Subproject commit 6f47087f1c9079ea6c677702da23ca040d0a13ed という 1 つのコミットとして記録していることが分かります。

メインのリポジトリを作って、そこにサブモジュールを追加する、という一通りの流れを体験してみました。ここで作った main_project というディレクトリは、まるごと消してしまって構いません。

1.5 サブモジュールを含むリポジトリをクローンしてこよう

さっきは「ローカルでリポジトリを作って、そこにサブモジュールを追加してみる」という流れを体験しました。今度は prh-rules というサブモジュールを使っている、本著の原稿リポジトリを GitHub からクローンしてみましょう。ローカルでのディレクトリ名は sub_test にします。

```
サブモジュールを使っているメインリポジトリをクローンしてくる
$ git clone https://github.com/mochikoAsTech/SubmoduleUpdated sub_test
```

メインのリポジトリをクローンしてきたら、prh-rules というサブディレクトリを開いてみましょう。なんと中身は空っぽです。（図 1.6）

▲図 1.6　サブモジュールの中身は空っぽ！

実はサブモジュールを含むメインのリポジトリをクローンすると、「サブモジュールが入っているはずのディレクトリ」は取得できるのですが、その時点ではその中身は空っぽなのです。

サブモジュールを初期化する`--init`オプションを付けて、`git submodule update`というコマンドを叩くことで、中身を連れてこられます。

```
サブモジュールの状態を初期化する（中身をクローンしてくる）
$ git submodule update --init
Submodule 'prh-rules' (https://github.com/mochikoAsTech/prh-rules)
 registered for path 'prh-rules'
Cloning into 'C:/Users/mochikoAsTech/Documents/SubmoduleUpdated/prh-rules'...
Submodule path 'prh-rules': checked out
 'f126abf930039a23d5e6ea9f418451fe69277ddb'
```

これでサブモジュールである`prh-rules`の中身、つまり正誤表を含む校正ツールを一式持ってこられました。

1.6　最初からサブモジュールの中身も含めて全部連れてきたかった

メインのリポジトリをクローンしてきた直後に、空っぽの`prh-rules`を見ると「なんでサブモジュールの中身も一緒に連れてきてくれないの？！」という気持ちになります。[*8]そういうときは`--recursive`オプションを付けてクローンすることで、最初からサブモジュールの中身も含めて全部まるっと連れてこられます。

[*8] みなさんがなるかどうかは分かりませんが筆者はなりました。なんで！ 一緒に！！ 連れてきてくれないの？！？！ 判子と朱肉はセットでしょ？！

```
サブモジュールも含めて全部まるっとクローンしてくる
$ git clone --recursive https://github.com/mochikoAsTech/SubmoduleUpdated
```

筆者はいちいちサブモジュールの有無を確認するのが面倒なので、リポジトリをクローンするときは基本的に`--recursive`オプションを付けています。

1.7 メインリポジトリとサブモジュールは親子の関係

ところで 1 つのものをいろんな名前で呼ぶと混乱するので、ここから先の本書での名前を整理しておきましょう。

- 親
  - メインのリポジトリ（`SubmoduleUpdated`）のこと
  - 内部にサブモジュールを含んでいる
  - メインリポジトリやスーパープロジェクトと呼ばれることもある
- 子
  - 親から見たサブモジュールのリポジトリ（`prh-rules`）のこと
  - 子（`prh-rules`）は、親（`SubmoduleUpdated`）の中にある
  - つまり子は親のサブディレクトリである

メインリポジトリとサブモジュールは、親子のような関係になっています。親は常に最新の子を見ている訳ではなく、子の特定のコミットを「子」として認識しています。[*9]

たとえば、子のリポジトリではコミットが「A → B → C」と進んでいて C が最新ですが、親のリポジトリでは子の「B のコミット」を指定しているとします。この場合、親のリポジトリで`git submodule update`コマンドを叩くと、最新の C ではなく B の時点の子をクローンしてきます。

つまり`git submodule update`は、「子を最新にして！」というコマンドではなく、「親が認識している時点（コミット）の子にアップデートして！」なのです。

一方、親のサブディレクトリである子に`cd`して、`git pull`コマンドを叩くと、親の認識に関係なく最新の「C のコミット」をクローンしてきます。親の認識している「B のコミット」を連れてきたいのか、それとも親の認識に関係なく最新の「C のコミット」を連れてきたいのか、自分がしたいのはどちらなのか？ を把握して、適切な git コマンドを叩くことが大切です。

[*9] 「俺さ、大学進学のタイミングで家を出たから、うちの親の中ではハタチのままの俺で時間が止まってるんだよね。だからお盆に帰るとからあげとか山盛り出されてさ…俺もう 30 過ぎてるし、揚げ物そんなに食べられないんだけどなー」みたいな感じですね。あ、なんか急に切ない。

第 2 章

サブモジュールのトラブルシューティング

サブモジュールで困ったらトラブル事例を見てみよう！

2.1 【トラブル】サブモジュールを使っているか使っていないか確かめたい

「このプロジェクトってサブモジュール使ってたっけ？」と聞かれたけれど、「あの共通ライブラリはサブモジュールだったっけ…？ それとも Composer でパッケージ管理していたんだっけ…？」と分からなくなってしまった。そんなときは親のリポジトリで `git submodule status` コマンドを叩いて、サブモジュールを使っているか否かを確かめてみましょう。サブモジュールの、いまの状態が表示されます。

```
サブモジュールを使っている場合
$ git submodule status
 ec6d80a111881e28c6e8e5129cfa6a49b995830b prh-rules (heads/master)

サブモジュールはあるが、まだ初期化されていない場合
$ git submodule status
-ec6d80a111881e28c6e8e5129cfa6a49b995830b prh-rules  ←先頭にハイフンがある

サブモジュールを使っていない場合
$ git submodule status
            ←何も表示されない
```

2.2 【トラブル】git pull しただけなのにサブモジュールの差分が生まれた

あなたはいま親の master ブランチにいます。一緒に開発しているメンバーに「master ブランチを更新したから `git pull` してね！ サブモジュールも最新版になったよ」と言われました。幸い、手元でやりかけの作業はなかったので、「よし、じゃあ最新にするか！」と指示どおりに親で `git pull` コマンドを叩きました。

```
$ git pull
remote: Enumerating objects: 3, done.
remote: Counting objects: 100% (3/3), done.
remote: Compressing objects: 100% (1/1), done.
remote: Total 2 (delta 1), reused 2 (delta 1), pack-reused 0
Unpacking objects: 100% (2/2), 249 bytes | 14.00 KiB/s, done.
From https://github.com/mochikoAsTech/SubmoduleUpdated
   08bacf6..383f393  master     -> origin/master
Updating 08bacf6..383f393
Fast-forward
```

```
 prh-rules | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```

「よしよし、子（prh-rules）の更新を pull できたぞ！」と思って、確認のため git status を叩くと、なんとサブモジュールが更新されている、と出ます。

```
$ git status
On branch master
Your branch is up to date with 'origin/master'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   prh-rules (new commits)

no changes added to commit (use "git add" and/or "git commit -a")
```

「いま git pull して連れてきただけで、私は何のファイルも更新していないのに、なんで差分がでるの？」という気持ちになります。では git diff コマンドで、Git が「なにを更新したと言い張っているのか？」を見てみましょう。

```
$ git diff
diff --git a/prh-rules b/prh-rules
index 782af14..f126abf 160000
--- a/prh-rules
+++ b/prh-rules
@@ -1 +1 @@
-Subproject commit 782af14a4dae78d62b591f7dab818826f721ca70  ←10歳のサブモジュール
+Subproject commit f126abf930039a23d5e6ea9f418451fe69277ddb  ←8歳のサブモジュール
```

親を git clone したときのことを思い出してみましょう。あのときも、親を git clode しただけでは、子のディレクトリは空っぽで、git submodule update --init コマンドを叩いたことによって、初めてサブモジュールの中身を連れてくることができました。

それと同じで、git pull しただけでは、親が認識している子の年齢（コミット）が最新の 10 歳になるだけで、実際の子（つまりサブモジュールのディレクトリ）の中身は自動追従してこないので、8 歳のままなのです。つまり git pull したことによって、親の認識は「うちの子は 8 歳！」から「うちの子は 10 歳！」に変わったのですが、一方でそこにいる子供はまだアップデートされておらず 8 歳のままなので、親から見ると「10 歳だったうちの子が 8 歳になってる！ 更新したんだね！ 差分が

ある！」という状態になっているのです。

親の認識に合わせて、サブモジュールの中身も 10 歳になってほしいので、`git submodule update` コマンドを叩いて、そこにいるサブモジュールを、「親が認識している子の年齢」、つまり 10 歳にアップデートしましょう。

```
サブモジュールを「親が認識している子の年齢（コミット）」にアップデートする
$ git submodule update
Submodule path 'prh-rules': checked out
 '782af14a4dae78d62b591f7dab818826f721ca70'  ←10歳のサブモジュールを連れてきた
```

`git pull` したことで、親が認識している子の年齢（コミット）が変わったんだから、そこはちゃんと付いてきてよ！ と思いますが、そいうものなのです。

2.3 【トラブル】他のブランチへ移動しただけでサブモジュールがなぜか更新された

あなたはいま feature ブランチで、「親の認識する子の年齢（コミット）を最新にする」という作業を行っています。もともと親が認識している子（サブモジュール）の年齢は 8 歳でしたが、実際のサブモジュールは既に 10 歳を迎えていました。親のリポジトリでサブモジュールのディレクトリに移動した上で、`git pull` コマンドを叩いて子を 10 歳にしてやり、親の認識を「うちの子は 10 歳！」に改めた上で、feature ブランチをコミット＆プッシュをしました。いまは「更新したけど未コミットなもの」は何もない状態です。

この状態で `git checkout master` して、master ブランチへ移動します。するとチェックアウトしてブランチを移動しただけなのに、`git status` を見るとこんな差分が表示されます。

```
$ git status
(中略)
        modified:   prh-rules (new commits)

no changes added to commit (use "git add" and/or "git commit -a")
```

メイ、なにも変えてないもん！ master をチェックアウトしただけだもん！ ほんとだよ！ トトロいたもん！ と叫びたくなりますが、`git diff` コマンドを叩いて、Git が「なにを更新したと言い張っているのか？」を見てみましょう。

```
$ git diff
diff --git a/prh-rules b/prh-rules
index f126abf..782af14 160000
--- a/prh-rules
+++ b/prh-rules
@@ -1 +1 @@
-Subproject commit f126abf930039a23d5e6ea9f418451fe69277ddb  ←8歳のサブモジュール
+Subproject commit 782af14a4dae78d62b591f7dab818826f721ca70  ←10歳のサブモジュール
```

そうです。`git checkout master` しただけでは、子の中身は自動追従してこないので 10 歳のままなのです。ですが master ブランチでは、親が認識している子の年齢はまだ 8 歳です。そのため「うちの子が！！ 10 歳になってる！！！ 8 歳から 10 歳に更新したでしょ？！」となっているのです。

master ブランチで子の年齢を変えたい訳ではないので、こんなときは `git submodule update` を叩いて、そこにいるサブモジュールを、「親が認識している子の年齢」にアップデートしてあげましょう。

```
サブモジュールを「親が認識している子の年齢（コミット）」にアップデートする
$ git submodule update
Submodule path 'prh-rules': checked out
 'f126abf930039a23d5e6ea9f418451fe69277ddb'  ←8歳のサブモジュールを連れてきた
```

update というと、どうしても「古いものから新しいものにアップデートする」というイメージなので、「10 歳から 8 歳の状態に戻す」ために `git submodule update` コマンドをたたくのは不思議な感じがするかもしれません。でもサブモジュールを、「親が認識している子の年齢」にアップデートするのが `git submodule update` なので、これでいいのです。

2.4 【トラブル】サブディレクトリで git pull したらなぜか更新された

あなたはいま親の master ブランチにいます。一緒に開発しているメンバーから「master ブランチで子の年齢（コミット）を 9 歳にしておいたから、master ブランチを pull しておいてね」と言われたので、指示どおりに `git pull` コマンドを叩きました。

```
$ git pull
remote: Enumerating objects: 3, done.
remote: Counting objects: 100% (3/3), done.
remote: Compressing objects: 100% (1/1), done.
remote: Total 2 (delta 1), reused 2 (delta 1), pack-reused 0
Unpacking objects: 100% (2/2), 249 bytes | 14.00 KiB/s, done.
From https://github.com/mochikoAsTech/SubmoduleUpdated
   08bacf6..383f393  master     -> origin/master
Updating 08bacf6..383f393
Fast-forward
 prh-rules | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```

「よしよし、子（prh-rules）の更新を pull できたぞ！」と思って、確認のため git status を叩くと、なんとサブモジュールが更新されている、と出ます。

```
$ git status
On branch master
Your branch is up to date with 'origin/master'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   prh-rules (new commits)

no changes added to commit (use "git add" and/or "git commit -a")
```

git diff で確認したところ、どうやら親が認識している子の年齢（コミット）は 9 歳になったのに、そこにいる子は 8 歳のままなので、更新したと見なされて差分が出ているようです。

```
$ git diff
diff --git a/prh-rules b/prh-rules
index 4ac24b6..ec6d80a 160000
--- a/prh-rules
+++ b/prh-rules
@@ -1 +1 @@
-Subproject commit 4ac24b6fd2df37053900fe21f611b9b4131b1941  ←9歳のサブモジ
ュール
+Subproject commit ec6d80a111881e28c6e8e5129cfa6a49b995830b  ←8歳のサブモジ
ュール
```

「子を最新にすればいいんでしょ！」と思ったあなたは、子のディレクトリに cd prh-rules して、git pull を叩きました。再び git diff で確認してみたのですが、なんと差分はなくなっていません。親が認識している子の年齢（コミット）は 9 歳なのに、最新版の子を git pull して結果、9 歳を飛び越えていきなり最新の 10 歳に

なってしまったようです。

```
$ git diff
diff --git a/prh-rules b/prh-rules
index 4ac24b6..4690206 160000
--- a/prh-rules
+++ b/prh-rules
@@ -1 +1 @@
-Subproject commit 4ac24b6fd2df37053900fe21f611b9b4131b1941  ←9歳のサブモジュール
+Subproject commit 4690206e805b34e63580e0506b476dcdcc3d4c27  ←10歳のサブモジュール
```

親の認識に合わせて、サブモジュールの中身には 9 歳になってほしいので、この場合は親のディレクトリで `git submodule update` コマンドを叩きましょう。

```
サブモジュールを「親が認識している子の年齢（コミット）」にアップデートする
$ git submodule update
Submodule path 'prh-rules': checked out
 '4ac24b6fd2df37053900fe21f611b9b4131b1941'  ←9歳のサブモジュールを連れてきた
```

2.5 手間なくラクにサブモジュールを更新するには

話をまとめると、cd コマンドでサブモジュールに移動して、そこで `git pull` してはいけないのです。親の認識にかかわらず、子が最新版になってしまいます。最新版が 10 歳で、親の認識も 10 歳であれば問題ありませんが、いずれにしてもこの方法はお勧めしません。

親で `git pull` した後、メインプロジェクトが認識している、サブモジュールの状態にアップデートしたければ、`git submodule update` を使いましょう。サブモジュールの中にさらにサブモジュールがいる場合は、`--recursive` オプションを付けた `git submodule update --recursive` を叩けば、入れ子になっているサブモジュールも再帰的にアップデートしてくれます。

`git pull` と `git submodule update` で、いちいち 2 回叩くのは面倒だな、という場合は `git pull && git submodule update` でひとまとめにしてもかまいません。

ですが `git pull && git submodule update` を叩くくらいなら、代わりに、`git pull --recurse-submodules` でも同じ結果が得られますし、なんなら `git con`

`fig submodule.recurse true` を叩いて、`submodule.recurse`[*1]を有効にしてしまえば、以降は `git pull` するだけで同じ結果が得られます。

サブモジュールの挙動を理解して、サブモジュールを手間なくラクに「親が認識している子の年齢（コミット）」にアップデートしましょう。

[*1] ただし true にすると、`--recurse-submodules` オプションがあるすべてのコマンド (checkout、fetch、grep、pull、push、read-tree、reset、restore、switch) に適用されます。`https://git-scm.com/docs/git-config#Documentation/git-config.txt-submodulerecurse`

あとがき

本書を書いたことで、prh-rules（正誤表を含む校正ツールのディレクトリ）をサブモジュールにすれば、本の原稿リポジトリを作るたびに、前回のリポジトリからまるごと持ってこなくてよくなる！」と気づきました。さらに原稿を書く過程で、git pull したとき、サブモジュールを一緒にアップデートしてくれる--recurse-submodules オプションの存在も知ったので、技術書によっていちばん多く学びを得るのは著者自身だな、と改めて感じました。

数ある技術書の中から「Git のサブモジュールで困ったら読む本」を手に取ってくださったあなたに感謝します。

2020 年 9 月
mochikoAsTech

PDF 版のダウンロード

本書はどなたでも PDF 版を無料ダウンロードできます。ぜひあなたの周りにいるサブモジュールでお困りの方に、本書を知らせてあげてください。

- ダウンロード URL
  - https://techbookfest.org/product/4837403810332672
  - https://mochikoastech.booth.pm/items/2357642

Special Thanks:

- 深夜のジューシーピーチフラペチーノに付き合ってくれる旦那様
- ハムエッグを焼くのが得意な息子
- 電気を消しなさいよ、と小声でリクエストする猫
- Zoom のミーティングでかわいさをアピールしたい猫
- 目があうだけでごろごろの音量が 3 倍になる猫

- むっちりとたくましいのに遠慮がちな猫
- ソースコードとブロッコリーの守護を司る猫

レビュアー

- 旦那様

参考文献・ウェブサイト

- git-scm.com
  - `https://git-scm.com/`

著者紹介

mochiko / @mochikoAsTech

テクニカルライター。元 Web 制作会社のインフラエンジニア。ねこが好き。「分からない気持ち」に寄り添える技術者になれるように日々奮闘中。技術書典で頒布した「DNS をはじめよう」「AWS をはじめよう」「技術をつたえるテクニック」「技術同人誌を書いたあなたへ」は累計で 9,700 冊を突破。

- https://twitter.com/mochikoAsTech
- https://mochikoastech.booth.pm/
- https://note.mu/mochikoastech
- https://mochikoastech.hatenablog.com/
- https://www.amazon.co.jp/mochikoAsTech/e/B087NBL9VM

Hikaru Wakamatsu

表紙デザインを担当。

Shinya Nagashio

目次、章扉デザインを担当。

Git のサブモジュールで困ったら読む本

2020 年 9 月 12 日　技術書典 9 初版

著　者　　mochikoAsTech
デザイン　Hikaru Wakamatsu / Shinya Nagashio
発行所　　mochikoAsTech