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。

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。

- 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が更新されない
- `git submodule update` とサブモジュールのフォルダで `git pull` したときの差が分からない

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パッケージでいい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

ああサブモジュール、君のこと（挙動）が分からない

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 そのもの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 SourceTree のような 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。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」と言われてもピンとこないと思うので、便利な使用例をご紹介しましょう。

1.2.1 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[*1]という校正ツールがあり、その中には次のように「表記揺れを自動チェックするための正誤表」が含まれています。

```
- 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たびに、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フォルダ（`prh-rules`）をまるごとコピーしてくる必要がありました。

コピーは面倒くさいですし、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、Git を使っているとは思えないような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あちこちにあるのは気分的にもよくありません。ああ、校正ツールのフォルダ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！ 校正ツールのフォルダ（`prh-rules`）だけをひとつのリポジトリとして用意しておき、それぞれの原稿リポジトリでサ

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

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で説明すると「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。」となります。

サブモジュールは便利な一方で、挙動を理解せずに使うとトラブルの元になりやすいです。前述のように「たしかに！ これは便利だ！」と心の底から思えるときにだけ使うのがお勧めです。

1.3 サブモジュールの作り方

まず親を作ります。

```
メインのリポジトリ（main_project）を作る
$ git init main_project
```

続いてメインのリポジトリのサブモジュールとして、既に存在している別のリポジトリを追加します。

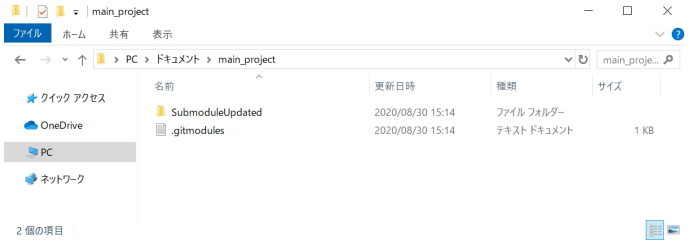
```
作ったリポジトリのディレクトリに移動する
$ cd main_project

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。[*2]追加した SubmoduleUpdated 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。このときサブモジュール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以下については再帰的にはクローンしてきてくれないので注意が必要です。

[*2] 今回は GitHub の URL を指定しましたが、このような絶対の URL に限らず、`git submodule add ../SubmoduleUpdated` のような相対パス、サブモジュールとして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` のようになってしまうので、特に理由が無ければリモートの URL で指定する方がお勧めです。

```
$ git submodule add https://github.com/mochikoAsTech/SubmoduleUpdated
Cloning into 'C:/Users/mochikoAsTech/Documents/main_project/SubmoduleUpdated'..
remote: Enumerating objects: 162, done.
remote: Counting objects: 100% (162/162), done.
remote: Compressing objects: 100% (143/143), done.
remote: Total 162 (delta 43), reused 26 (delta 7), pack-reused 0 eceiving objec
objects: 100% (162/162), 518.33 KiB | 772.00 KiB/s, done.
Resolving deltas: 100% (43/43), done.
```

なおサブモジュールを追加するとき、先ほどのように特にディレクトリ名を指定しないと、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。（図 1.1）



▲図 1.1　SubmoduleUpdated がサブモジュールとして追加された

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。

```
git submodule add https://github.com/mochikoAsTech/SubmoduleUpdated submodule
```

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。

サブモジュールを追加すると、このように親のリポジトリに.gitmodules というファイルが生まれます。これはテキスト形式の設定ファイルで、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すると、このファイルにその分だけ追記されていきます。

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

他の人が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すると、Git はこのファイルに書かれた内容を元に、サブモジュールの取得元を把握することになります。

続いて git diff で SubmoduleUpdated を見てみましょう。コミット前のファイルの差分を見たいので、--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、親から見た「子の年齢」のような +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。

1.4 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

今度は実際に prh-rules というサブモジュールを使っている原稿リポジトリをクローンしてみましょう。

```
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
```

prh-rules という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。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、最初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 することで、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（中身を連れてくる）
$ git submodule update --init
Submodule 'prh-rules' (https://github.com/mochikoAsTech/prh-rules) registered f
Cloning into 'C:/Users/mochikoAsTech/Documents/SubmoduleUpdated/prh-rules'...
Submodule path 'prh-rules': checked out 'f126abf930039a23d5e6ea9f418451fe69277d
```

これで正誤表を含む校正ツールを一式持ってこられました。

1.5 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

空っぽの prh-rules を見ると「使いたいからサブモジュールとして指定してるの！ 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。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。

```
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。

1.6 親子の関係

Git のメインリポジトリを親、サブモジュールを子だとします。

サブモジュールのリポジトリではコミットが「A → B → C」で進んでいて、けれど親のリポジトリではまだ B を参照しているとします。この場合、親のリポジトリで`git submodule update`したら、サブモジュールは B を引っ張ってきます。

`git submodule update`とは、「子のサブモジュールを最新にしておくれ」ではなく、「"親が指定している子のコミット"に更新しておくれ」なのです。

1.7 git pull しただけなのに差分が出た

あなたはいま master ブランチにいます。他の人から「master 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`git status`を叩いても、更新してあるファイルはありません。よし、じゃあ最新の状態にするか！ と`git pull`を叩いたところ、なんとサブモジュールが更新されたよ！！ 差分があるよ！！ コミットしなよ！！ という表示がわらわら出てきました。

いったい何が起きたのでしょう？ master ブランチで`git pull`を叩いただけで、なんのファイルも更新していないのに、なぜ更新されたファイルがわらわら出てくるのでしょうか？

`git clone`したときのことを思い出してください。あのときも、`git clode`しただけでは、サブモジュールのディレクトリは空っぽで、`git submodule update --init`を叩いたことで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。

実は`git pull`しただけでは、メインのリポジトリが指しているサブモジュールのコミットが最新版になるだけで、実際のディレクトリの中身は連れてこられないのです。`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！ 差分だ！」という状態になっているのです。

そんなときは深呼吸をして、`git submodule update`を叩きましょう。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くれます。

```
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する
$ git submodule update
```

1.8 git checkout しただけなのに差分が出た

あなたはいま feature ブランチで、親の認識するサブモジュールのバージョンを最新版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。メインのリポジトリでもサブモジュールを `git pull` して 10 歳にしてやり、親が認識しているサブモジュールの年齢を 10 歳に更新した上で、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！ ほんとだもん！ と叫びたくなりますが、`git diff` 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 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` しただけでは、サブモジュールの中身は自動追従してこないので、サブモジュールの中身は 10 歳のままです。でも master ブランチでは、親が認識しているサブモジュールの年齢はまだ 8 歳です。そのため「子供が！！ 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 ブランチで何か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」にアップデートしましょう。

```
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する
$ git submodule update
Submodule path 'prh-rules': checked out 'f126abf930039a23d5e6ea9f418451fe69277d
  ←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` 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。

1.9 サブモジュールで git pull しただけなのに差分が出た

あなたはいまメインリポジトリの master ブランチにいます。一緒に開発しているメンバーから「サブモジュールのバージョン上げたから、master ブランチを pull しておいてね」と言われました。

```
$ 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、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 して連れてきたのに、なんで差分がでるの？ 私は更新してないよ？？という気持ちになります。では **git diff** 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
```

そうです。**git pull** したことによって、親が認識している子供の年齢は 10 歳になったのですが、サブモジュールの中身は自動追従してこないので 8 歳のままなのです。それによって「子供が 10 歳から 8 歳になってる！ あなた更新したわね！」となっているのです。

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** 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。

```
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する
$ git submodule update
Submodule path 'prh-rules': checked out '782af14a4dae78d62b591f7dab818826f721ca
  ←10歳のサブモジュールを連れてきた
```

git pull したことで、メインリポジトリが認識しているサブモジュールのコミットが変わったんだから、そこはちゃんと付いてこいよ！ と思いますが、そいうものなのです。筆者は **git pull && git submodule update** を叩いたら楽なのかな、と思ったりします。

1.10 正しい差分の無くし方

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。

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。

筆者は git pull && git submodule update を叩いたら楽なのかな、と思ったりします。

あとがき

この原稿を書いたことで、「prh-rules（校正のルールを書いたファイルたち）をサブモジュールにすれば、本のリポジトリを作るたびに前回のリポジトリから prh-rules 持ってなくてもいいんじゃん！」と気づいてしまったので、本当に本を書くのは自分にとっていいことだなぁ、と思いました。まる。

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悩みの方に知らせてあげてください。

- ダウンロード URL
  - `https://mochikoastech.booth.pm/items/xxxxxxx`

Special Thanks:

- ねこ

レビュアー

- かしこいねこ

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

挿絵デザインを担当。

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