Git 唐揚げ

KA-FU-

目次

はじめに

お早うございます、KA-FU-です。今年中 3 になりました。時が流れるのは速いです...。未だに幼稚園児並の能力しかないことに悲しみを通り越して憤りを感じる、そんな年齢になってしまいました。完全に無能なので日頃できるようなこともなく就寝と睡眠を繰り返す毎日です。

みなさん、唐揚げは好きですか？私は大好きです。「唐揚げ」という言葉を聞くだけでお腹の音で教室の窓を割ってしまうほど好きです。(え？「初めからわけのわからないギャグを突っ込んでしかも完全にスベってるんですけど...、何この人...。」って？この記事を書いた人は日頃からそういうことをしている可哀想な人間なんだと思って我慢してください...。) この記事はそんな非常に美味な唐揚げという食べ物について研究し、最高の唐揚げを求め続けている 1 人の唐揚げ職人のお話です...。

第 I 章

唐揚げ職人 ギット=グル

あるところに、長年続く唐揚げ屋がありました。その店の店主をしているのは代々唐揚げ職人をしているギット家の長男、ギット=グルです。ギット家では料理に git というツールを使っています。そのツールはなんと、各工程における状態を保存しておき、万が一味付けを間違えたりした場合でも、問題がなかった時の状態に戻すことが出来ます。また、お客さんが多くて大変な時でも複数人で作業を分担し、後で組み合わせるということが出来ます。ギット=グルがこの素晴らしいツールを使う様子を見てみましょう！ (なおこの記事での出力は git version 2.8.1 によるものを、都合よく書き換えたりしたものです。)

1　唐揚げ作り

git は誰でも使うことが出来ます。まずは install してみましょう。ここ[*1]に Linux 等での install 方法が書かれています。Mac でも homebrew なりなんなりを使って簡単に install できると思います。

まずは下準備

まず、git を使うには下準備が必要です。いろいろと用意する必要があるのですが、これは唐揚げ作り自体には関係がありません。なのでギット家は git にそれを自動化する機能を埋め込みました。次のコマンドを実行してみましょう。[*2]

```
$ cd /path/to/dir

$ mkdir test
```

[*1] https://git-scm.com/download/linux

[*2] ”/path/to/dir” をお好みのディレクトリへの path に置き換えてください。

```
$ cd test

$ git init

Initialized empty Git repository in /path/to/dir/test/.git/
```

すると、.git というディレクトリができます。

```
$ ls -a

./    ../   .git/
```

この中にはいろいろなものが入っています。

```
$ ls .git

HEAD         description  info/        refs/
config       hooks/       objects/
```

ここで重要になるのは HEAD、objects、refs です。(今ないだけで後から追加されるものにも重要なものがありますが。) HEAD には今の状態、objects には保存した状態、refs には状態につけた名前が入ります。では、使い方と一緒にいったいどういうことが行われているのかを見ていきましょう。

唐揚げの作り方

まず、git には 3 つの場所があります。working directory、staging area、repository です。working directory は実際に私たちが作業しているディレクトリです。staging area は、次に git に保存する変更を指定する場所です。これがあることで実際にしている変更の一部のみ保存することができます。(例えば、機能 A を追加した後、機能 B を追加している最中でも、機能 A の分の変更だけを保存することが出来ます。) では、とりあえず状態を更新してみましょう。

```
$ echo "唐揚げよりラーメンのほうが旨くないですか？" > secret.txt

$ git status
```

```
On branch master

Initial commit

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

  secret.txt

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

git status を呼ぶことで今どういう状況なのか知ることが出来ます。出力からわかる情報は、「今 master ブランチにいて、最初の commit で、管理されていないファイル”secret.txt” がある」ということです。” ブランチ” や”commit” については少し後で説明します。この時点ではまだ何も起こりません。

では、次のコマンドを実行してみましょう。

```
$ git add secret.txt

$ find .git/objects -type f

.git/objects/b8/90a2a3832021404e126f5eca67580cb12cdd13
```

git add を使って working directory から staging area に指定したファイル (ここでは”secret.txt”) を登録することができます。また、”secret.txt” は新しく作ったファイルなのでこの時に git の管理対象に入ります。ファイルは管理対象に入った時に.git/objects の中にあるモノが作られています。これが blob オブジェクトと言って、ファイルを表すものです。

blob オブジェクト

blob オブジェクトの名前は、表すファイルの中身と管理に必要な情報をくっつけたものを SHA1 でハッシュ値にしたものです。SHA1 というのは 40 文字のハッシュ値を出力するハッシュ関数で、生成したハッシュ値は非常に重複しにくく、同じ内容なら同じハッシュ値なのはもちろんのこと、違う内容なら違うハッシュ値が帰ってくると考えて問題ありません。このことにより、内容が変更されたかどうかなどの判断がしやすくなります。また、blob オブジェクトの中身はさっき SHA1 に渡したのと同じものをを zlib で圧縮したものになっています。

git add は working directory から staging area にファイルを登録するものと言いました。では、staging area で”secret.txt” が指定されていることを確認してみましょう。

```
$ ls .git/

HEAD         description  index        objects/
config       hooks/       info/        refs/

$ git write-tree

706b8ddbf680cc8c043deb36b4e4912a801e38c4

$ git cat-file -t 706b8ddbf680cc8c043deb36b4e4912a801e38c4

tree

$ git cat-file -p 706b8ddbf680cc8c043deb36b4e4912a801e38c4

100644 blob b890a2a3832021404e126f5eca67580cb12cdd13  secret.txt
```

.git の中に index というファイルが追加されています。ここに staging area の情報が入っています。しかし、このままでは staging area に”secret.txt” が登録されているのかがわかりにくいので、git write-tree というコマンドを使って、staging area を tree オブジェクトという、ディレクトリを表すものに書き出しています。

tree オブジェクト

この tree オブジェクトや、後から出てくるものでも名前などの決め方は blob オブジェクトと同じです。しかし、tree はディレクトリを表すものなので、blob オブジェクトのように、ファイルの中身をそのまま使って、内容を生成するわけではありません。表すディレクトリが含むファイルやサブディレクトリのハッシュ値とヘッダー、ファイル名を列挙したものを tree オブジェクトの内容の生成に使います。

git cat-file というコマンドに-t を指定することで、渡した名前のオブジェクトが何を表すオブジェクトなのかがわかります。今回指定したのは、先ほど書きだした tree オブジェクトですので、”tree” と表示されています。また、-p を指定してオブジェクトの名前を渡すと、オブジェクトの内容をわかりやすく表示してくれます。出力を見てみると、「test ディ

レクトリはファイルのモード[*3]が”100644” の”secret.txt” というファイルを含んでていて、それの blob オブジェクトの名前が”b890a2a3832021404e126f5eca67580cb12cdd13” だ」ということがわかります。

さっき同じ内容なら同じハッシュ値が生成されるといいましたが、ほんとうかどうか確認してみましょう。

```
$ cat secret.txt > new.txt

$ git add new.txt

$ git write-tree

46a812a4779be5710d1d6bec02fbedc018080038

$ git cat-file -p 46a812a4779be5710d1d6bec02fbedc018080038

100644 blob b890a2a3832021404e126f5eca67580cb12cdd13  new.txt
100644 blob b890a2a3832021404e126f5eca67580cb12cdd13  secret.txt

$ find .git/objects -type f

.git/objects/46/a812a4779be5710d1d6bec02fbedc018080038
.git/objects/70/6b8ddbf680cc8c043deb36b4e4912a801e38c4
.git/objects/b8/90a2a3832021404e126f5eca67580cb12cdd13
```

最初に”secret.txt” と全く同じ内容の”new.txt” を作ります。そして、さっきと同じように staging area を tree オブジェクトに書き出してみて、中身を見てみると”secret.txt” と”new.txt” が同じ名前であることがわかります。また、.git/objects の中を見ても、”secret.txt” の blob オブジェクトと、さっき生成した tree オブジェクト、そして今生成した tree オブジェクトの分しかなく、”new.txt” の分は新しく生成されていないことがわかります。

では次に staging area に登録したファイルを repository に移してみましょう。これが状態を保存する作業になります。

```
$ git status
```

[*3] ここでは説明しません。追加の情報だと思ってください。

```
On branch master

Initial commit

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

  new file:   new.txt
  new file:   secret.txt

$ git commit -m "唐揚げとかコンビニで買えばいいじゃん..."

[master (root-commit) 3da9726] 唐揚げとかコンビニで買えばいいじゃん...
 2 files changed, 2 insertions(+)
 create mode 100644 new.txt
 create mode 100644 secret.txt

$ git log

commit 3da9726896338cbfbc370fd27c4ba31efb79d2b1
Author: name <email-address>
Date:   date

    唐揚げとかコンビニで買えばいいじゃん...

$ find .git/objects -type f

.git/objects/3d/a9726896338cbfbc370fd27c4ba31efb79d2b1
.git/objects/46/a812a4779be5710d1d6bec02fbedc018080038
.git/objects/70/6b8ddbf680cc8c043deb36b4e4912a801e38c4
.git/objects/b8/90a2a3832021404e126f5eca67580cb12cdd13

$ git cat-file -p 3da9726896338cbfbc370fd27c4ba31efb79d2b1

tree 46a812a4779be5710d1d6bec02fbedc018080038
author name <email-address> timestamp
```

```
committer name <email-address> timestamp
```

唐揚げとかコンビニで買えばいいじゃん...

git commit というコマンドを使って commit をすることができます。commit とは、staging area から repository に移すことで、commit をすると commit オブジェクトというものが作られます。

commit オブジェクト

commit オブジェクトには、保存する tree オブジェクト、一個前の commit、作者と commit した人の情報、commit した時間、commit メッセージが保存されています。 (もちろん、root commit つまり一番最初の commit には一個前の commit はありません。) git commit に-m をつけて文字列を渡すとそれを commit メッセージにすることができます。 (つけずに実行した場合、git に設定されているエディタが開かれて、そこで commit メッセージを書くことになります。) commit メッセージとは、その commit で保存された変更がいったいどういったものなのかを説明するために使うものです。これを読めば、実際にどんな変更があったか逐一みなくてもよくなります。 (上手い commit メッセージが書ければですが...。) また、一個前の commit の情報を保存しているので、これを順々にたどっていけば変更履歴を見ていくことが出来ます。

git log というコマンドを使えば変更履歴を表示することが出来ます。また、.git/objects の中を見るとちゃんと commit オブジェクトが追加されています。(”3da9726896338cbfbc370fd27c4ba31efb79d2b1”) そして git cat-file を使ってその内容を見てみると、git log で表示されたのと内容が同じであることがわかります。 (少し表示のされ方は違いますが。)

今までハッシュ値でオブジェクトを指定していましたが、あんな長くて人間から見たらデタラメな名前なんてきっとみんな嫌でしょう。なので、commit オブジェクトに別名をつけることが出来ます。この名前は.git/refs の中にに保存されます。では実際に別名をつけてみましょう。

別名にも種類があってその中で一番身近なのは、ブランチです。

ブランチ

これは.git/refs/heads の中に保存されます。また、今いる場所をブランチを使って指定すると、commit した時にブランチが自動で新しい commit の別名に変わってくれます。heads というディレクトリ名からもわかるように、作業の最新の場所を指すのに使うものなので自動で新しい commit の別名になってくれると嬉しいのです。

デフォルトは master ブランチで、先ほどの作業も master ブランチでしていたので、git status の出力に「master ブランチにいる」という情報が入っていました。

```
$ cat .git/refs/heads/master

3da9726896338cbfbc370fd27c4ba31efb79d2b1

$ git cat-file -p 3da9726896338cbfbc370fd27c4ba31efb79d2b1

tree 46a812a4779be5710d1d6bec02fbedc018080038
author name <email-address> timestamp
committer name <email-address> timestamp

唐揚げとかコンビニで買えばいいじゃん...

$ echo "とてもつらい" > kimochi.txt

$ git add kimochi.txt

$ git commit -m "締め切りなどなかったのだ..."

[master 98a946b] 締め切りなどなかったのだ...
 1 file changed, 1 insertion(+)
 create mode 100644 kimochi.txt

$ git log --oneline

98a946b 締め切りなどなかったのだ...
3da9726 唐揚げとかコンビニで買えばいいじゃん...

$ cat .git/refs/heads/master

98a946b8c1ac5f90f3737a0c0f398fef8e60d765
```

master ブランチのファイルに書いてあるハッシュ値を見てみると、確かにさっき commit した最新の commit が指されています。そして、もう一度 commit してみると新しい方の commit に自動で変わっているのがわかります。 (git log に--oneline を渡すと各 commit が一行で表示されます。また、ハッシュ値が先頭の何文字かだけになっていますが、意外とそれだけでどれかわかったりするので最初の何文字かだけで指定することが多いです。) で

は、さっき言った「今の状態」がどこなのか、git はどうやって覚えているのでしょうか？
実は.git/HEAD の中にその情報が書かれています。

```
$ cat .git/HEAD

ref: refs/heads/master

$ git log --oneline

98a946b 締め切りなどなかったのだ...
3da9726 唐揚げとかコンビニで買えばいいじゃん...

$ git checkout 3da9726

Note: checking out '3da9726'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -b with the checkout command again. Example:

  git checkout -b <new-branch-name>

HEAD is now at 3da9726... 唐揚げとかコンビニで買えばいいじゃん...

$ cat .git/HEAD

3da9726896338cbfbc370fd27c4ba31efb79d2b1

$ git checkout master

Previous HEAD position was 3da9726... 唐揚げとかコンビニで買えばいいじゃん...
Switched to branch 'master'
```

今は master ブランチにいるので master ブランチのファイルの場所が書かれています。

そして、HEAD が指すのはブランチでなくても良く、commit を指すことも出来ます。(ブランチは commit の別名なので当然といえば当然ですね。) git checkout というコマンドを使うと今いる場所を変更することが出来ます。実際に最初の commit を指定してみると、HEAD が変更されているのがわかります。 (commit に直接 checkout するのはあまり推奨されることではないので、git からたくさん文句を言われています。)

また、作業中なのに checkout してブランチを移動しなければいけなくなったりしたとき、一時的に作業内容を退避することができます。

```
$ git stash [save]

$ git stash list

$ git stash apply stash@{数字}

$ git stash drop stash@{数字}

$ git stash pop stash@{数字}
```

git stash というコマンドを使うと作業内容を退避できます。git stash に save を渡すと今の編集中の内容が保存されて変更が戻ります。list を渡すことで保存している変更の一覧がでます。list した際に、各行の最初にあった”stash@数字” を指定して apply を渡すことで、指定した変更を適用することが出来ます。しかし、適用した変更もまだ保存されたままなので、drop を使って消すことが出来ます。また、この 2 つのことを pop を使って一回で済ませることも出来ます。

そして、ブランチ以外の別名にはタグがあります。

タグ

タグもブランチと同じで commit を指します。タグには軽量タグと注釈付きタグと呼ばれる二種類があって、注釈付きタグでは付加できる情報が多いです。タグは.git/refs/tags の中に保存されます。commit と似た情報が保存され、タグを作った人、日付、指す commit、また、注釈付きタグではメッセージも保存されます。

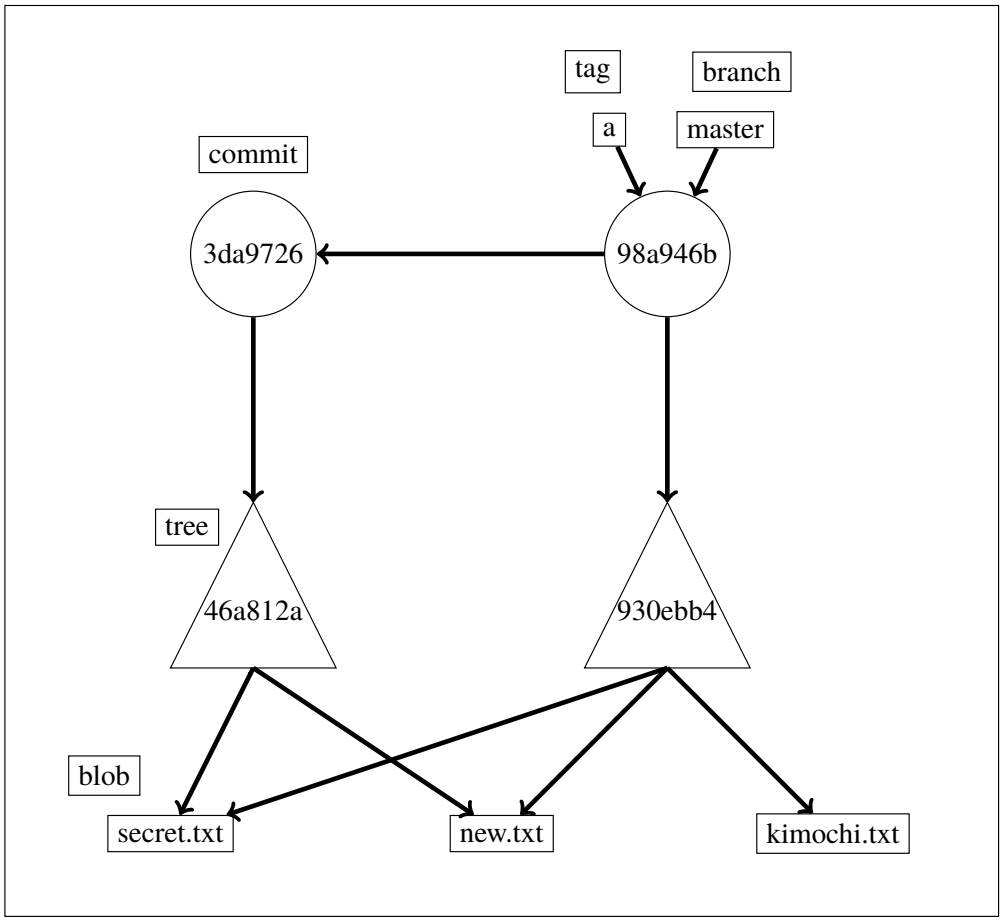
注釈付きタグを使うことによって、「この commit がバージョン 1.2.3 だ」のようなことををメッセージに書くことが出来ます。

```
$ git tag a
```

```
$ git show a

commit 98a946b8c1ac5f90f3737a0c0f398fef8e60d765
Author: name <email-address>
Date:   date

    締め切りなどなかったのだ...

$ git tag -a b -m "c"

$ git show b

tag b
Tagger: name <email-address>
Date:   date

c

commit 98a946b8c1ac5f90f3737a0c0f398fef8e60d765
Author: name <email-address>
Date:   date

    締め切りなどなかったのだ...

$ cat .git/refs/tags/b

fa7551f3f7fb5d9b6ce5ade3585a8d3abdb3ae29

$ git cat-file -p fa7551f3f7fb5d9b6ce5ade3585a8d3abdb3ae29

object 98a946b8c1ac5f90f3737a0c0f398fef8e60d765
type commit
tag b
tagger name <email-address> timestamp

c
```

git tag の後にタグの名前を渡せば、軽量タグを作ることが出来ます。git show コマンドで中身を見てみると、確かに指定した commit のハッシュ値が記されています。git tag に-a を渡してタグの名前を渡すと注釈付きタグが作れます。また、-m と文字列を渡すとメッセージを指定できます。注釈付きタグをつくると.git/objects の中にもタグオブジェクトが作られます。

ここまでの内容を図で見てみましょう。(タグ b は省略しました。) (色を付けたらもっとわかりやすいと思うのですが、面倒なので白黒で我慢してください. . . 。)



下位のファイルに変更があれば、そのファイルのハッシュ値が変わり、それを含むディレクトリのハッシュ値も変わり、となっていって変更があれば必ずルートディレクトリのハッシュ値が変わります。そのため、ルートディレクトリのハッシュ値を比較するだけで変更があったかがわかるのです。また、変更されていないものについては同じオブジェクトを使いまわしています。素晴らしいですね！！！

2　お客さんが待ってる！

今まで git の説明をするために開店時間を遅らせていたので、お客さんが行列を作っています。 (なんとギット=グルの店は行列ができるほどには繁盛していたのです！驚き！) 素

早くお客さんを捌くために手分けしましょう！ master ブランチ以外にもう一つブランチを使って作業します。

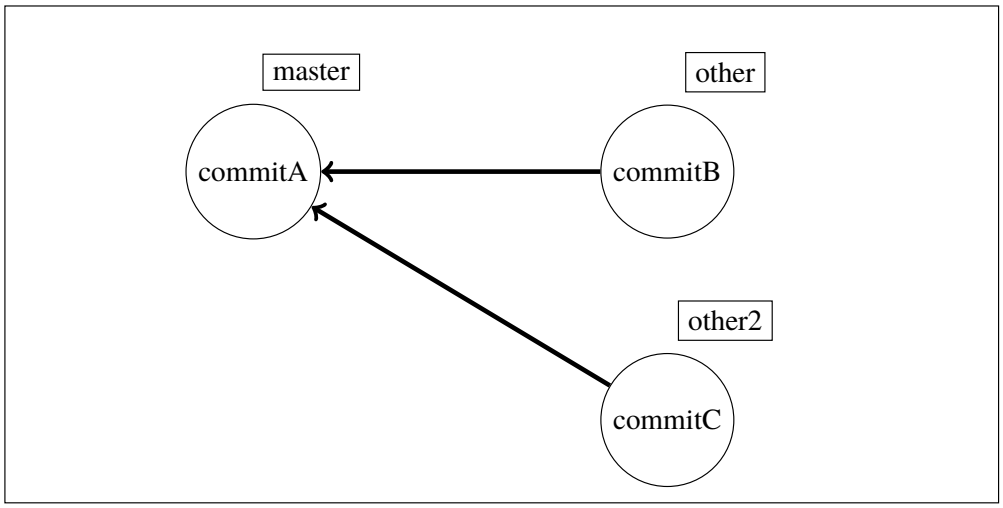
```
$ cd ../ && rm -rf test && mkdir test && cd test && git init

Initialized empty Git repository in /path/to/dir/test/.git/

$ echo "num: 0" > karaage.txt

$ git add karaage.txt && git commit -m "Initial commit"

[master (root-commit) 781f26e] Initial commit
 1 file changed, 1 insertion(+)
 create mode 100644 karaage.txt

$ git branch other && git branch other2
```

まず最初に、”karaage.txt” のなかに今できている唐揚げの個数を書き入れました。(もちろん 0 です。) そしてブランチ”other”、”other2” を作りました。では、作業を始めましょう！

```
$ git checkout other

Switched to branch 'other'

$ echo "kind: 2" >> karaage.txt

$ git add karaage.txt && git commit -m "スーパー A へ買いに行く"

[other 462fb63] スーパー A へ買いに行く
 1 file changed, 1 insertion(+)

$ git checkout other2

Switched to branch 'other2'
```

```
$ echo "num: 10" > karaage.txt

$ git add karaage.txt && git commit -m "スーパー B で唐揚げを買う"

[other2 04af820] スーパー B で唐揚げを買う
 1 file changed, 1 insertion(+), 1 deletion(-)
```

さて、とりあえず一段階唐揚げ店の作業をしました。(不正はなかった、いいね？) まず現在の状況を図にしてみました。



いやぁ非常に単純ですね、書いた人の手抜きが目に見えてますね〜。では、ここで merge をしてみましょう。

merge

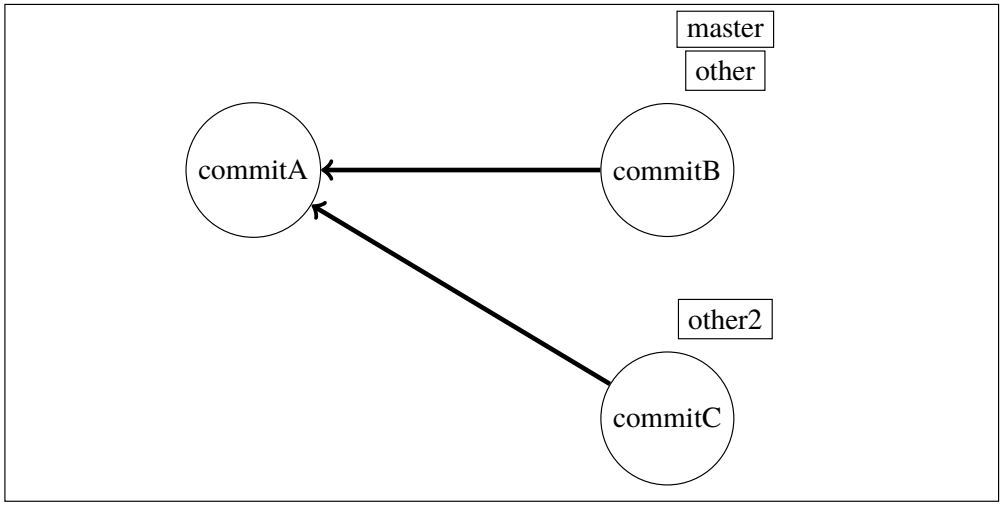
違うブランチの変更を合成します。git は賢いのでうまい感じに変更を合成してくれます、流石ですね。また、merge する 2 つのブランチが分岐しておらず、片方が進んでいるだけの時の merge は特別に fast-forward といいます。

では実際にしてみましょう。

```
$ git checkout master

Switched to branch 'master'

$ git merge --ff-only other

Updating 781f26e..462fb63
```

```
Fast-forward
 karaage.txt | 1 +
 1 file changed, 1 insertion(+)
```

git merge で merge が出来ます。指定したブランチの変更を今いるブランチに合成します。master と other は分岐しておらず、other が先に進んでいるだけなので、fast-forward merge になります。--ff-only を渡すことで fast-forward merge を git に強要することができます。今の状況の図を見てみましょう。



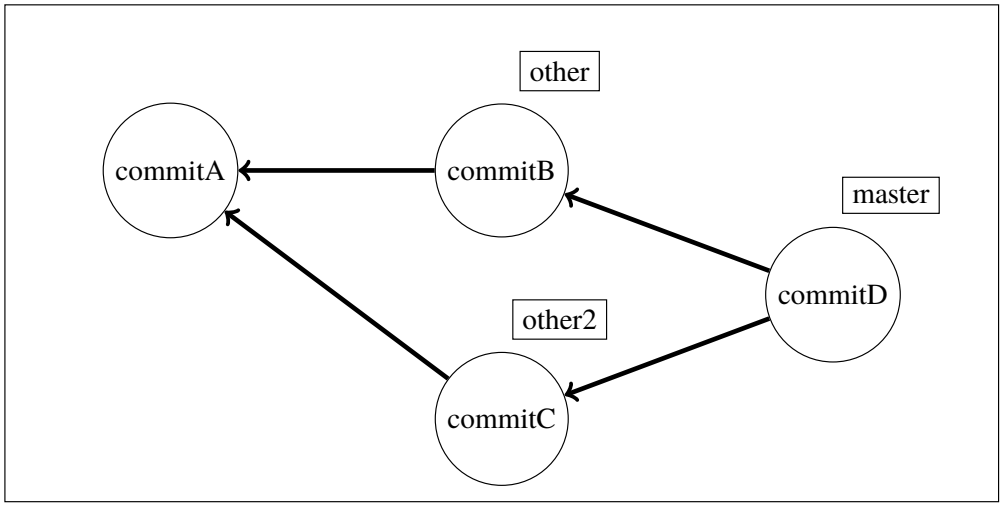
まぁ、変更が少しだけなのでそれだけですか...みたいな状態ですが、是非みなさんが開発する際に効果を実感してください！では一応 non-fast-forward もしておきましょう。

```
$ git merge other2

Auto-merging karaage.txt
CONFLICT (content): Merge conflict in karaage.txt
Automatic merge failed; fix conflicts and then commit the result.

$ cat karaage.txt

<<<<<<< HEAD
num: 0
kind: 2
=======
num: 10
>>>>>>> other2
```

おっと、何か出ました。これは conflict ですね。複数の commit で同じ箇所が修正されていた場合に git が検出して、どっちの変更にするか選べと言ってきます。ここでは”num: 10” の方を選びましょう。

```
$ echo "num: 10" > karaage.txt

$ git add karaage.txt && git commit

[master 3c9aecd] Merge branch 'other2'
```

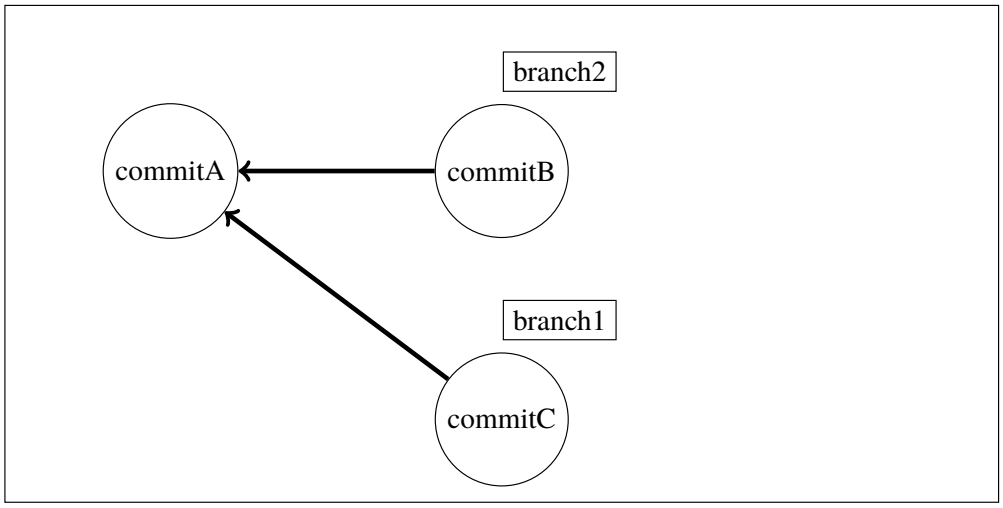
一応図でも見てみましょう。



実は merge 以外にも変更を合成する方法があります。rebase といいます。

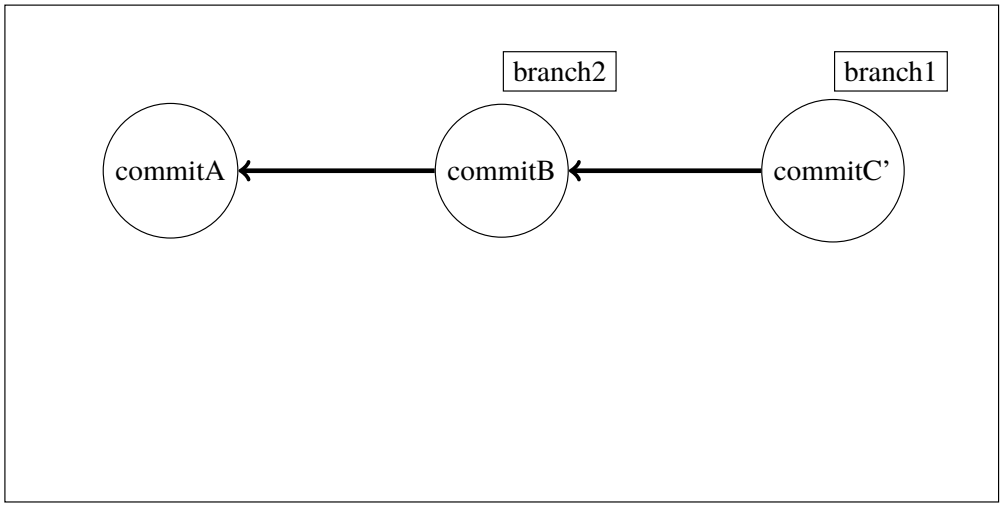
rebase

rebase は、相手のブランチと今のブランチが分岐した地点からの自分の変更点を、相手のブランチに適用して、そこに自分のブランチを移動します。言葉で聞いてもわかりにくいと思うので、とりあえず図を見てみましょう。



この状態で rebase を実行してみましょう。

```
$ git checkout branch1

$ git rebase branch2
```



こうなります。ここで気をつけて欲しいのは、最初の”commitC” と rebase 後の” commitC′” は違うものだ、ということです。特徴としては、merge と違って合成した後の履歴が綺麗になります。その時その時であっているほうを使いましょう。

この調子でギット=グルは唐揚げを調達作っていきました。あっと、重大なミスが発覚したようです！どうやら~~購入した~~作った唐揚げの量を間違えていたみたいですね。けどギット=グルは間違いがすごく嫌いなので git の履歴に間違いを残したくありません。さぁ！やっと git の過去改変の力を見せる時が来ました！親 commit 殺しのパラドックスなんて起こりません！なぜなら git は神のツールですから！もし違ったら木の下に埋めて貰っても構わないよ！ (ババーン)

```
$ git commit --amend

$ git reset --soft

$ git reset [--mixed]

$ git reset --hard
```

まず、git commit --amend は直前の commit を上書きします。git reset --soft は HEAD だけを指定した場所まで戻します。git reset --mixed は HEAD と staging area を、(これがデフォルトです。) git reset --hard は working directory も戻します。ちょっとした間違いならこれを使って直せたりします。しかし、もっと便利な道具があります。強力な過去改変ツールをご紹介します！

```
$ git rebase -i

pick abcdefg hogehogehoge
pick hijklmn foobarbar

# Commands:
# p, pick = use commit
# r, reword = use commit, but edit the commit message
# e, edit = use commit, but stop for amending
# s, squash = use commit, but meld into previous commit
# f, fixup = like "squash", but discard this commit's log message
# x, exec = run command (the rest of the line) using shell
# d, drop = remove commit
#
# 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.
#
# Note that empty commits are commented out
```

git rebase -i を使うと、指定したところまでの commit を編集することが出来ます。各 commit のハッシュ値、commit メッセージがあり、その前に”pick” と書いてあります。この”pick” のところを変更したり、行の順番、つまり commit の順番を変えたりすることで履歴を編集します。

入れる文字列 (省略形)	説明
pick (p)	そのまま commit を使う
reword (r)	commit 内容は変えずに commit メッセージ
edit (e)	commit 内容を変える (--amend を使う)
squash (s)	一個前の commit と結合する (commit メッセージも結合される)
fixup (f)	一個前の commit と結合する (commit メッセージは結合されない)
exec (x)	書いたコマンドを実行する

そして、git は書き換え終わったそのファイルを上から順番に実行していきます。全て pick を指定して行の順番だけ並び替えるとその順番に commit が並びます。

そして、過去の汚点を綺麗さっぱり消してしまうのにちょうどよいまたまた強力なツールがあります。

```
$ git filter-branch
```

git filter-branch は過去全ての commit から指定した条件を満たすものを消すことができ、履歴から完全に何かを消してしまう時に使います。 (これは詳細は省略します...。) ここまで過去改ざんの方法を紹介しましたが、そもそも過去を改変しないといけないような状態に陥ってはいけないのです...。

さて、これでお店はどうにかなりそうです。 (どうしてこれで経営が保っているのか...。) では、お客さんのもとに届けましょう！実は、ギット=グルの唐揚げ屋はネット販売なのです (~~あれ？じゃあさっきの行列というのは...？~~) 、できた唐揚げをネットに上げましょう。

分散バージョン管理システム

git は分散バージョン管理システムです。各クライアントのパソコンにリポジトリのデータがあって、他人の作業と合成してサーバーに上げそれを他人がサーバーから落とし、というふうな流れです。各人がデータを持っているので、それぞれがバックアップにもなります。

なのでもちろんサーバーにデータを上げるコマンドがあります。わざわざ用意するわけにもいかないので、コマンドの紹介だけします。

```
$ git push <remote> <source-branch(local)>:<dest-branch(remote)>
```

```
$ git pull <remote> <source-branch(remote)>:<tracking-branch(local)>

$ git fetch <remote>
```

git push に渡す必要があるのは、サーバーにある remote リポジトリと、上げるブランチ名とリモートでのブランチ名です。しかし、だいたいローカルとリモートでブランチ名は同じにしますから、":" の前後が同じ場合には":" 以降は省略できます。次に、git pull は名前からして git push の逆のようなものです。何をやっているのかというと、git fetch したあとに指定されたブランチとカレントブランチを merge しています。では先に git fetch は何なのか見てみましょう。これは、リモートリポジトリのブランチを追跡ブランチに落とします。

追跡ブランチ

これは.git/refs/remotes の中に保存されています。リモートのブランチをそのまま落としてきたものになっています。

git pull に指定するのは、サーバーにある remote リポジトリと、落としてくるリモートのブランチ名、追跡ブランチ名です。git fetch した後に指定した追跡ブランチとカレントブランチを merge します。

そして、さきほど過去改ざんの方法を紹介しましたが、過去の履歴を改ざんした際すでに push していた場合、そのままでは push できません。そこで--force を渡して強制的に上書きしてしまいます。

```
$ git push --force
```

しかし、これも通常していいものではありません、push する際には十分確認をしてからしましょう。

さて、ギット=グルもお客さんのために唐揚げを push することが出来たようです。これで一件落着ですね。

第 II 章

ライバル店を倒せ

ネット販売の店となるとこのご時世ライバルもたくさんいます！その中にはレシピをオープンな場所で公開してしまっている店もあるそうで、ギット=グルはそれを狙って他の店の情報を手に入れようと考えました。どうやら、そのような店は GitHub[*1]のような remote リポジトリを作り、git を使ってバージョン管理と共同作業ができるサービスを使ってレシピを公開してしまっているようです。

1　ライバル店のレシピを盗もう

では、GitHub のリポジトリから pull したり、push したりするために、どうやって remote リポジトリを指定するのでしょうか？

```
$ git remote

$ git remote -v

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

git remote を使って remote リポジトリに名前をつけることが出来ます。git remote を実行すると、名前をつけた remote リポジトリの一覧が表示されます。-v を付けることで remote リポジトリの URL の一覧が表示されます。git remote add を使うと、新しく remote リポジトリに名前をつけることが出来ます。基本的にこのようにつけた名前を使って remote リポジトリを指定します。GitHub などのサービスを使う際でも URL は簡単にコピーできるようにしてあります。また、GitHub などにすでに remote リポジトリがある時など、まだ手元にリポジトリがないときは git clone を使います。

[*1] https://github.com/

```
$ git clone <url>
```

指定した URL から repository を落としてきて、repository 名と同じ名前のサブディレクトリを作ってそこに入れてくれます。名前を渡せばその名前のサブディレクトリを作って入れてくれます。これを使って、ギット=グルはいくつかの店のレシピの情報を手に入れることができました。

2　ライバル店に文句をつけよう

手に入れたレシピのなかに、ギット=グルが唐揚げ職人としてどうしても許せないミスなどが見られ、これに文句をつけたいと思いました。しかし、どうやら他人の remote リポジトリに勝手に push することはできないようです...。そういう時は、GitHub では pull request を行います。(同様のサービスの GitLab[*2]では merge request と呼ばれます。私はこちらのほうが好みです。) まずは GitHub に自分がこうしたほうが良いと思う変更を上げるために、remote リポジトリをコピーして自分の remote リポジトリを作ります。この作業を fork と言って、GitHub などではボタン一つですることができます。そしてその remote リポジトリを clone して変更を加え push します。ここで pull request をします。これもまたボタン一つで可能で、是非みなさんが GitHub 上のプロジェクトに協力しようと思った際に使ってください。ギット=グルも pull request をしましたが、ギット=グルは口が悪く pull request の際につけたコメントがとても過激なものになってしまい、それを見た相手の店のお客さんから反感を買ってしまいました...！！！

3　そして...

ライバル店の予想以上の人気、そして迂闊な発言により店は経営不振に陥ってしまった...。自分の腕に完全に自信をなくしてしまったギット=グルは修行の旅に出ると言い出した。自分の唐揚げを楽しみに待っていてくれるお客さんを背にギット=グルは旅立って行く...。しかし、私たちは信じている。必ず彼は帰ってきて、私たちにより一層美味しくなった唐揚げを食べさせてくれるはずだ！そう！

[*2] https://gitlab.com/

ギット=グルは wwwwwww きっと来る wwwwwwwww

あとがき

これが、やりたかった、だけ。

参考文献

[1] Scott Chacon, Ben Straub. (2014). 『Pro Git』 Apress
[2] Jon Loeliger. (2010). 『実用 Git』 オライリージャパン
[3] これがやりたかっただけだろシリーズ (ニコニコ大百科)[*3]

[*3] http://dic.nicovideo.jp/a/これがやりたかっただけだろシリーズ