Git・Githubの中級者入門ゼミ

0. GithubとSSH

野口研のGithub組織内のリポジトリはデフォルトPrivateですので、cloneするにはSSHキーの設定が既にできていると思います。少しSSHの基礎から復習をします。
特定のユーザネームで特定のサーバへSSH通信する際、以下のコマンドを利用する。

```
$ ssh username@hostname
```

パスワード認証の後に、Shellが開きます。
パスワード認証の以外に、公開鍵認証の方法も利用されています。

```
$ ssh username@hostname -i <Private_Key_File_Path>
```

`<Private_Key_File_Path>`は秘密鍵のパスです。
GithubへSSH通信する際に、同じフォーマットでPCからGithubへの疎通を確認をすることができます。

```
ssh -T git@github.com -i <Private_Key_File_Path>
```

ここの`git`はユーザネームで、`github.com`はサーバです。この中の`-T`は「擬似ターミナルが要らない」という意味のオプションです。(擬似ターミナルとは何なのか、`tty`と何が違うのかについて、自分で調べて、または野口先生に聞いてください。)
様々な記事では、SSH公開鍵をGithubに登録した後、このコマンドでSSH通信ができているかどうかを確認していました。要するに、`-i`にGithubに登録した公開鍵のペアである秘密鍵を参照し、正規なユーザかどうかを認証していることです。GithubのSSH通信はユーザ認証のためのものです。

```
Github SSH通信の流れ
1. リポジトリからCloneする
2. git clone git@<Repository_URL>を叩くと、上記のSSH通信が発生する
3. 初回目のGit操作は、登録された公開鍵の情報・リポジトリのURLは使用者PCにknown_hostsという名前で保存される
4. 二回目以降のGit操作は、保存された情報を参照し、再び認証を行う
```

しかし、二つ以上のGithubアカウントを利用する場合、4番で一つの問題が発生してしまいます。それは、仮に二回目以降のGit操作が別のリポジトリで発生している場合、SSH通信がまた`known_hosts`で記録した情報を参照してしまいます。参照すべき公開鍵情報が違うため、以下のエラーが発生します。

```
$ git clone git@github.com:noguchi-lab-ibaraki/git-practice.git
Cloning into 'git-practice'...
git@github.com: Permission denied (publickey).
fatal: Could not read from remote repository.

Please make sure you have the correct access rights
and the repository exists
```

学生の頃に複数のGithubアカウントを使う場面があまりないのですが、皆さんはITエンジニアとして就職する際、マルチアカウントの運用問題がいずれ避けられないので、今のうちにやってみましょう。
（会社によって、Github/Gitlab/Bitbucketまたは自社用VCSを使うことがあります）

1. マルチアカウントの配置

Userフォルダの直下に、SSHフォルダ下のファイルたちを確認しましょう。

```
# Windowsの場合、C://Users/<UserName>に移動してください。
$ cd .ssh/
$ ls -l
total 56
-rw-r--r--  1 hinositamirai  staff   262 May 28 00:42 config
-rw-------  1 hinositamirai  staff  3381 May 25 16:42 firosstuart
-rw-r--r--  1 hinositamirai  staff   737 May 25 16:42 firosstuart.pub
-rw-------  1 hinositamirai  staff  3369 May 28 00:34 i_kumo
-rw-r--r--  1 hinositamirai  staff   732 May 28 00:34 i_kumo.pub
-rw-------  1 hinositamirai  staff   656 May 28 00:43 known_hosts
-rw-r--r--  1 hinositamirai  staff    92 May 28 00:43 known_hosts.old
```

私のPCでは、いくつかのSSHキーペアが存在しています。

- `config` SSHの設定ファイル、存在しない場合がある
- `firosstuart*` 今このGithub組織で使うアカウントのSSHキーペア
- `i_kumo*` 別のGithub組織で使うアカウントのSSHキーペア
- `known_hosts*` 接続したことがあるホスト・使用したキーの情報、

複数のGithubアカウントを利用している場合、`config` ファイルでキーペアごとに、通信するサーバに別名を追加する必要があります。
私の `config` ファイルを説明します。

```
# Private account
Host github.com-private
   HostName github.com
   User git
   IdentityFile ~/.ssh/firosstuart
   IdentitiesOnly yes

# Community account
Host github.com-community
   HostName github.com
   User git
   IdentityFile ~/.ssh/i_kumo
   IdentitiesOnly yes
```

`Host` 項目ごとに、`SSH` の設定を記述しています。設定ファイルに書き込むことで、指定した `Host` 名にアクセスする時点で、自動的に下の細かい設定を適用します。`Host github.com-private` が個人用アカウントで、`Host github.com-community` が別の組織で使うアカウントです。`IdentityFile` に秘密鍵を指定しています。`git@github.com-private` にアクセスする際、`~/.ssh/firosstuart` のファイルを使用していて、`git@github.com-community` にアクセスする際、`~/.ssh/i_kumo` のファイルを使用していることがわかります。
`git@github.com-private` にアクセスしてみると、成功です。

```
$ ssh -T git@github.com-private
Hi FirosStuart! You have successfully authenticated, but GitHub does not provide shell
access.
```

実際、IdentityFileやHostnameなどはSSHコマンドの固有オプションで、これまで使っていたオプションと互換性があります。
例えば、`ssh -T git@github.com -i ~/.ssh/firosstuart` コマンドは以下の長いコマンドと同じ効果です。

```
$ ssh git@github.com -o IdentityFile='~/.ssh/firosstuart' -o RequestTTY='no'
Hi FirosStuart! You have successfully authenticated, but GitHub does not provide shell
access.
```

もちろんですが、`git` じゃない場合でも、SSH設定ファイルを利用すれば、毎回長いコマンドを入力しなくても済みます。

2. ブランチの同期

ローカルブランチとリモートブランチの同期は、以下の二種類があります。

1. ローカルブランチの方が新しいので、リモートブランチにPushすることで、リモートブランチを更新する
2. リモートブランチの方が新しいので、ローカルブランチにPullすることで、ローカルブランチを更新する

`git push`は相対的に簡単です。現在HEADにいる位置からPushとか、権限がある場合の強制Pushとか、注意すべきところが少ないです。

```
# 現在HEADにいる位置からPush
$ git push origin HEAD
# 権限がある場合の強制Push
$ git push origin HEAD -f
```

しかし、Pullするのは実際のチーム開発の場合、ただの`git pull`で済みません。

環境を準備しながら、要点をまとめます。
まず、Git-practiceのリポジトリをCloneしましょう。

```
$ git clone git@github.com:noguchi-lab-ibaraki/git-practice.git
```

既にClone済みの場合、masterブランチを最新状態に更新しましょう。

```
$ git fetch && git checkout master && git reset --hard origin/master
```

このコマンドの解釈は以下の三つです。

1. リモートリポジトリから最新の情報を取得する
2. 今別のブランチにいる場合、まずmasterブランチに移動する
3. ローカルのmasterブランチをリモートのmasterブランチに同期する

なぜ`git pull`しなかった? `git pull`のコマンドは、実は二つの効果があります。まずリモートリポジトリから最新の構成をダウンロードし、そしてその更新を対応しているすべてのブランチにマージします。つまり、`git pull = git fetch && git merge`といういっぺんに実行する関係です。マージ操作が存在すると、コンフリクトの発生が可能です。
チーム開発の場合、コミットの数が数千個になることもよくあることですので、たった一個のコミットの違いで、リモートブランチと競合を起きる場合があります。その場合、コンフリクトが発生します。コンフリクトの解消は、ほぼすべてのGitユーザが直面する最大の問題だと言えるでしょう。
コンフリクトをできれば発生させないため、`git fetch`で更新を取り込んだ後、`git reset --hard <Source_Branch>`というコマンドで、**現在のブランチを強制的にソースブランチに変えます。**現在のブランチがリセットされてしまうので、masterブランチにいないと、別のブランチがリセットされてしまいます。master以外のローカルブランチを対応しているリモートブランチにリセットする場合、`git checkout <Branch> && git reset --hard origin/<Branch>`という操作でリセットすることができます。
`--hard`オプションを使っているが、実は`--hard`は極めて危険です。
本来、`git reset`は三つのレベルがあります。

- `--soft`　IndexとWorktreeを変更しない
- `--mixed`IndexとWorktreeに変更がない場合何も発生しない、StageされたIndexがあった場合、Worktreeに返す
- `--hard`IndexとWorktreeの変更を丸と消す

`--hard`を指定すると、リモートブランチに存在しない変更が本当に存在しなくなりますので、何かPushし忘れたものがあったら、壊滅的になるので、上記のブランチ同期する以外は慎重に使ってください。
上記の三つの組み合わせは長いと思った時に、`rebase`を使いましょう。

```
$ git pull --rebase
```

`git pull`は`git fetch && git merge`ですが、`--rebase`を指定すると、`git fetch && git rebase`のいっぺん実行になります。
rebaseはどういうものなのか、mergeと何か違うのか、文字で説明しづらいので、自分で調べてみてください。

3. ブランチの切り方

まず、ブランチはどこから**切れば**いいでしょうか。**ブランチを切る**というのは、現在のコミットツリーに一つブランチを作成することです。
実際の開発の流れを復習しましょう。masterブランチが保護され、直接にPushすることができないので、開発者はすべて最新のmasterから新しいブランチを作成し、変更をそのブランチにPushします。そして、変更内容やコミット履歴など、あとブランチが解決しようとする問題の説明も設定できるPull Requestを作成し、チームのコードレビュー者がレビューします。レビューが終わったら、担当者の承認を得て、masterブランチへのマージが許可されます。
常に最新のmasterから新しいブランチを作成することが基本です。

```
$ git checkout -b new_branch origin/master
```

開発方法により、最新のmasterじゃなく、別のブランチからさらに新しいブランチを作成することもあります。

```
$ git checkout -b new_branch origin/OtherBranch
```

さらに、特定のコミットからブランチを作ることもあります。

```
$ git checkout -b <new_branch_name> <start_point>
```

ブランチの起点(`start_point`)を選んでしまうと、そのブランチのベースコミットが定められます。その後の変更は、手間がかかります。
例えば、下のツリーがあります。masterブランチはA,B,C,D,Eのコミットで構成されています。branch1は最新のmasterから切ったものです。しかし、branch2はbranch1の先頭から切ってしまったから、最新のmasterブランチから切るように変更したいです。

```
# master    A - B - C - D - E
#                           |
# branch1                   F - G
#                               |
# branch2                       H - I - J
```

この場合、以下のコマンドが使用します。branch1から切ってしまったbranch2をmasterに移動するコマンドです。

```
$ git rebase --onto master branch1 branch2

# branch2                   H - I - J
#                           |
# master    A - B - C - D - E
#                           |
# branch1                   F - G
```

理解し難いと思いますが、`git rebase --onto どこへ どこから どのブランチ`のフォーマットです。`--onto`の後に指定できるパラメータは、ブランチだけではなく、コミットも使用できます。一般的な場合、ブランチの起点を注意して切ってください。`--onto`はできれば使わないようにしましょう。

4. マージ後の他ブランチへの影響

ここで以下の二つのシチュエーションがあります。

1. 開発者AとBが同時に開発を行なっていたが、Bがブランチを切る前にちょっと別のプロジェクトで二週間ぐらい忙しそうです。Bが戻ってブランチを切ろうとしたが、Aが既に3つぐらいのPull Requestがマージされました。この場合、Bがどうするべきでしょうか?
2. 開発者AとBが同時に開発を行なっています。AとBが同時にとある状態のmasterから各自のブランチを作成しました。AのPull Requestが承認され、変更がmasterにマージされました。その直後、Bも開発できたっぽいので、Pull Requestを作成しました。しかし、Bのブランチは古いmasterから切ったものなので、その中にAの先ほどマージされたコミットが入っていません。この場合、Bはどうするべきでしょうか?

シチュエーション1はよくあることです。要するにローカルリポジトリの情報がリモートリポジトリより古い状態で、どうするべきという問題です。この場合、リポジトリを更新してからブランチを切ればいいです。

```
$ git checkout master && git pull --rebase
$ git checkout -b <New_Branch_Name>
```

シチュエーション2は、実は何もしないのが正解です。しかし、現実ではそう単純ではありません。
以下のことが発生してしまいます。

- Aのブランチのコミットがないと、Bの機能は検証できないかもしれません。
- Aのブランチのコミットの変更に、Bも似ている変更をしたが、ちょっとやり方が違います。

正しいプロジェクトでは、疎結合なアーキテクチャであるべき、Aのコミットなくても、Bの機能が検証できるのが一般的な認識です。しかし、そういうまともなプロジェクトは中々ありません。
またはAとBが書こうとしたのは元々一つの機能だけど、一人で書けないので、二人のABに分けました。しかし、こんなに大きい機能を設計したのはPMと開発リーダーの責任です。
チーム開発はただの技術の問題ではなく、リーダーシップや人間性も色々な要素が入っているので、Gitを使う時にそういう問題も解決しないといけません。
Aのブランチのコミットがないと、Bの機能は検証できないかもしれません。の解決方法は以下となります。

```
$ git checkout <B_Branch>
$ git pull --rebase origin master
```

Aのブランチのコミットの変更に、Bも似ている変更をしたが、ちょっとやり方が違います。
この場合だと、大きな問題が発生します。それはコンフリクトです。みなさんは既にコンフリクトの概念についてわかったと思います。
経験上、正しいプロジェクトの進行かつ人為的なミスがない限り、コンフリクトが発生することがありません。つまり、上記の機能設計の問題がなければ、人為的なミスでコンフリクトを起こしているのです。
本来、AとBは自分の修正すべきところを承知しており、タスクが明白であれば、同じファイルの同じ行に同時に手を出したりしません。出した場合、後にPull Reqeust出した方がコンフリクトを解決しなければなりません。

5. コンフリクトの解消

以下で、コンフリクトの修正一例を出します。

```
$ git checkout <B_Branch>
$ git pull --rebase origin master # ここでAの最新のコミットを取り込むと、以下のコンフリクト警告が表示されます。
From github.com-private:FirosStuart/git-practice
 * branch            master       -> FETCH_HEAD
Auto-merging common.java
CONFLICT (content): Merge conflict in common.java
error: could not apply 0cb234c... B_Commit
hint: Resolve all conflicts manually, mark them as resolved with
hint: "git add/rm <conflicted_files>", then run "git rebase --continue".
hint: You can instead skip this commit: run "git rebase --skip".
hint: To abort and get back to the state before "git rebase", run "git rebase --abort".
Could not apply 0cb234c... B_Commit
```

現在の状態を確認してみましょう。

```
$ git status
On branch B_Branch
You have unmerged paths.
  (fix conflicts and run "git commit")
  (use "git merge --abort" to abort the merge)
Changes to be committed:

        new file:   A.java

Unmerged paths:
  (use "git add <file>..." to mark resolution)

        both added:  common.java
```

警告から、一部の情報が示されています。A.javaがAのブランチで追加された新しいファイルで、現時点のBのブランチに存在していないので、Aのコミットを取り込んだら、自動的にBのブランチにマージされました。しかし、common.javaはAのコミットでも、現在Bのコミット0cb234c... B_Commitでも修正されたので、コンフリクトが発生しました。
解決方法は、common.javaを開き、示されているコンフリクトしているところを修正することです。

```
5  <<<<<<< HEAD
6  var static Integer[]
7  =======
8  var static String[]
9  >>>>>>> 0cb234c (B_Commit)
```

<<<<<< HEADの下に示しているのは、最新のmasterブランチから来たAのコミットの変更です。========は二種類の変更の分割線です。分割線から>>>>>>>> 0cb234c (B_Commit)までの内容は、Bのコミットで行った変更です。どれを選ぶのか、Bが選ばなければなりません。例えば、Bが自分の変更を選ぶとしたら、5行から7行、9行の内容を消して、8行の内容だけを残します。そうすると、ファイルの第5行はこうなります。

```
5  var static String[]
```

この変更を保存し、rebaseを続行させましょう。

```
$ git add -u
$ git rebase --continue
```

Bのコミットの修正画面が出てきて、これを保存すれば、一つのコンフリクトが解決しました。
しかし、これはあくまで一つのコミットでのコンフリクト解消です。Aの変更を取り込んだら、Bのすべてのコミットにその変更を適用しなければなりません。もしBのいくつかのコミットで、またboth editedが発生したら、コンフリクト地獄が始まります。あまりよくない経験ですが、昔、最新developブランチの更新を取り込んだら、合計176件のコンフリクトがあって、コンフリクト解消だけで丸一日がかかりました。

6. ブランチの改名

実際の開発では、ブランチの名称は既にコード化されています。Github単独で利用するOSS創作の場合、<BranchType>/<BranchName>のように命名することが多いでしょう。例えば、新機能の追加ブランチではfeatures/add-new-something、コードの最適化ではrefactor/slim-codeのような名称が多いかと思います。多くの企業では、開発を一個ずつの課題として管理し、その課題に自動発行されたコードをブランチの名前として使用しています。例えば、NEW_PROJECT-205のような名称で、課題管理システムで、NEW_PROJECT-205wp検索すると、対応している詳しい内容が読めます。
しかし、うっかり名前を書き間違えることがたまにあります。それを備えて、ブランチの改名を慣れましょう。
ローカル・リモートのブランチを全部改名するには三つの手順があります。

```
git branch -m <old_name> <new_name>
git push <remote> <new_name>
git push <remote> --delete <old_name>
```

1. まずローカルのブランチを改名する
2. 現在のローカルブランチが既に新しいブランチになっているので、それをリモートにPushする
3. `git push` して、リモートの古いブランチを削除する

7. コミットの削除と移動

チーム開発の場合でも、個人で使う場合でも、たまに間違えたブランチにコミットしてしまうことがあります。
場合によって、いくつか対応できる方法があります。

- **そもそもコミットは既にPushしてしまって、自分にはコミット削除の権限がない**

`git revert` を使いましょう。`git revert` は実はマージ済みのブランチの内容まで取り消せるレベルのコマンドなので、Git管理員になるまで、謹慎的に使いましょう。
コミットのhashを取得し、以下のコマンドを実行し、間違えたコミットを消すコミットを追加します。

```
$ git revert 1a2b3d5
```

コミット修正画面に移動されるので、その内容を確認しましょう。
これで、リモートブランチでは、間違えて追加された変更が無くなります。

- **まだPushしてないけど、そのコミットはもう要らない**

簡単にリセットしましょう。

```
$ git reset --hard HEAD~
```

もしそのコミットの後ろに既に他のコミットがあった場合...次のRebaseの使い方を読んでください。

- **そのコミットは他のブランチで使うべきだった**

この場合は、まず正しいブランチに移動しましょう。移動する前に、Indexに追加されていない変更を保存しましょう。

```
$ git stash
$ git checkout <Branch>
```

先ほどのコミットのhashを指定し、以下のコマンドを実行します。

```
$ git cherry-pick 1a2b3d5
```

こうすると、別のブランチのコミットが現在にいるブランチに追加されました。その後、間違えたブランチに移動して、`git reset` でコミットを消せば問題ありません。

8. Rebase

`git rebase`と`git push origin master -f`の強制Pushの組み合わせは初心者から中級者への第一歩ですが、多くの会社で使用禁止されています。なぜなら、`git rebase`は現存のコミットツリーを改造することができ、少しやっちゃったら、プロジェクト全体に非常に大きい影響を与えます。ここで紹介するRebaseは、すべてまだPushしていないコミットに対する修正です。既にPushしたコミットの打ち消しや修正は、Revertコミットまたは他のコミットで直してください。

`git rebase`の種類は多いです。まず適当にHEAD一個前のコミットにRebaseし、Rebaseの一覧を出してみましょう。

```
$ git rebase -i HEAD~
pick b0d9311 This is a commit message.

# Rebase 600df77..b0d9311 onto 600df77 (1 command)
#
# Commands:
# p, pick <commit> = use commit
# r, reword <commit> = use commit, but edit the commit message
# e, edit <commit> = use commit, but stop for amending
# s, squash <commit> = use commit, but meld into previous commit
# f, fixup [-C | -c] <commit> = like "squash" but keep only the previous
#                    commit's log message, unless -C is used, in which case
#                    keep only this commit's message; -c is same as -C but
#                    opens the editor
# x, exec <command> = run command (the rest of the line) using shell
# b, break = stop here (continue rebase later with 'git rebase --continue')
# d, drop <commit> = remove commit
# l, label <label> = label current HEAD with a name
# t, reset <label> = reset HEAD to a label
# m, merge [-C <commit> | -c <commit>] <label> [# <oneline>]
# .       create a merge commit using the original merge commit's
# .       message (or the oneline, if no original merge commit was
# .       specified); use -c <commit> to reword the commit message
#
# 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.
```

ここの英文はすべてを話しました。1~2行までは、[操作] [Rebase対象コミットのHash] [Rebase対象コミットのメッセージ]のフォーマットで表示しています。
3行はRebaseの効果と作用範囲を説明しています。対象コミットから、HEADコミットの間のすべてのコミットが作用範囲となります。

4行以降は、Rebaseのコマンド種類を説明しています。1行目のpickを以下のpick,reword,edit,squash,fixup,exec,break,drop,label,reset,mergeに書き換えると、作用範囲内のコミットに色々なことができます。書き換えて、このコミットファイルを保存すると、rebase in progressになります。
一個の例を出します。
ちょっとlogを出してみます。b0d9311がHEADです。

```
$ git lg
* b0d9311 [2022-06-08] (HEAD -> test, origin/main, origin/HEAD, main) Update3
@FirosStuart
* 4ed3993 [2022-06-05] Update2 @FirosStuart
* f1efb79 [2022-06-04] Update1 @FirosStuart
* 78ce8fc [2022-06-02] Test2 @firosstuart
* 8f8e404 [2022-06-02] Test1 @firosstuart
* 4d524d5 [2022-06-02] first @firosstuart
```

8f8e404コミットの内容を修正したいです。追加で、78ce8fcのコミットメッセージを修正したくて、4ed3993のコミットを丸ごと消したいです。

```
$ git rebase -i 8f8e404~
  1 pick 8f8e404 Test1
  2 pick 78ce8fc Test2
  3 pick f1efb79 Update1
  4 pick 4ed3993 Update2
  6 pick b0d9311 Update3
  7
  8 # Rebase 4d524d5..b0d9311 onto 4d524d5 (6 commands)
  ...
```

操作したいコミットは8f8e404,78ce8fc,4ed3993なので、操作のタイプにより、それぞれコマンドを追加します。

```
  1 edit 8f8e404 Test1
  2 reword 78ce8fc Test2
  3 pick f1efb79 Update1
  4 drop 4ed3993 Update2
  6 pick b0d9311 Update3
  7
  8 # Rebase 4d524d5..b0d9311 onto 4d524d5 (6 commands)
  ...
```

保存すると、Rebaseに入ります。Rebaseの完成後、作用範囲のすべてのコミットが新しいコミットに変えられるので、新規コミット一連を4d524d5に追加されます。(--onto 4d524d5)

```
$ git status
interactive rebase in progress; onto 4d524d5
Last command done (1 command done):
   edit 8f8e404 Test1
Next commands to do (5 remaining commands):
   reword 78ce8fc Test2
   pick f1efb79 Update1
  (use "git rebase --edit-todo" to view and edit)
You are currently editing a commit while rebasing branch 'test' on '4d524d5'.
  (use "git commit --amend" to amend the current commit)
  (use "git rebase --continue" once you are satisfied with your changes)
```

これから8f8e404に一部のファイルを修正し、修正した内容をコミットに追加します。

```
$ [do something]
$ git add -u
$ git commit --amend --no-edit
```

現在のログを確認すると、8f8e404コミットが2b4902aになりました。
8f8e404の修正コマンドが完成すると、次の78ce8fcコミットのRewordに進みます。

```
$ git rebase --continue
```

Rewordはコミットメッセージの修正だけなので、出てきた修正画面で、メッセージを変えます。

```
 1 Test2 add new message!!
 2
 3 # Please enter the commit message for your changes. Lines starting
 4 # with '#' will be ignored, and an empty message aborts the commit.
 5 #
 6 # Date:      Thu Jun 2 12:45:54 2022 +0900
 7 #
 8 # interactive rebase in progress; onto 4d524d5
 9 # Last commands done (2 commands done):
10 #    edit 8f8e404 Test
11 #    reword 78ce8fc Test
12 # Next commands to do (4 remaining commands):
13 #    pick f1efb79 Update main.tf
14 #    drop 4ed3993 Update main.tf
15 # You are currently editing a commit while rebasing branch 'test' on '4d524d5'.
16 #
```

保存すると、rebase --continueをしなくても次のコマンドに進みます。次のコマンドはDropなので、コミットが自動的に削除されました。ここでRebaseが終了しました。
ログを確認してみましょう。

```
$ git lg
* d33f4ad [2022-06-08] (HEAD -> test) Update3 @FirosStuart
* 3d4f188 [2022-06-05] Update2 @FirosStuart
* 496708d [2022-06-04] Update1 @FirosStuart
* 6bd1367 [2022-06-02] Test2 add new message!! @firosstuart
* 2b4902a [2022-06-02] Test1 @firosstuart
* 4d524d5 [2022-06-02] first @firosstuart
```

4d524d5の後に、Rebase作用範囲[8f8e404...b0d9311]は5つの新規コミットに書き換えられました。本来のコミット78ce8fcのメッセージがTest2でしたが、修正後、Test2 add new message!!に無事に変えられました。これでRebaseの終了でした。
しかし、ここで一つの質問ですが、元々の[8f8e404...b0d9311]はまだ見れますか?

```
$ git show 8f8e404
commit 8f8e404baf4136c1ed012db6b135879a8f46e04a
Author: firosstuart <firosstuart@gmail.com>
Date:   Thu Jun 2 12:05:34 2022 +0900
```

まだ見れます。ローカルリポジトリのIndexファイルが存在している限り、どんなにRebaseして削除しても、コミットの情報は確認できます。

9. タイムマシン

Gitにはタイムマシンの機能がついています。厳密的に言えば、ただすべての操作とその操作時のIndex情報を記録したものです。

```
$ git reflog
b0d9311 (HEAD -> test, origin/main, origin/HEAD, main) HEAD@{0}: reset: moving to HEAD
b0d9311 (HEAD -> test, origin/main, origin/HEAD, main) HEAD@{1}: reset: moving to HEAD
b0d9311 (HEAD -> test, origin/main, origin/HEAD, main) HEAD@{2}: reset: moving to HEAD~
d5a0a54 HEAD@{3}: rebase (continue) (finish): returning to refs/heads/test
d5a0a54 HEAD@{4}: rebase (continue): Test
b0d9311 (HEAD -> test, origin/main, origin/HEAD, main) HEAD@{5}: pull --rebase origin
main (start): checkout b0d93111e524e7339e98b169791932932bfcc44a
0cb234c HEAD@{6}: commit: Test
600df77 HEAD@{7}: checkout: moving from main to test
b0d9311 (HEAD -> test, origin/main, origin/HEAD, main) HEAD@{8}: checkout: moving from
600df77c58236283385f8d1407c686e2ec47d602 to main
600df77 HEAD@{9}: checkout: moving from main to HEAD~
b0d9311 (HEAD -> test, origin/main, origin/HEAD, main) HEAD@{10}: pull --rebase: Fast-
forward
78ce8fc HEAD@{11}: commit: Test
8f8e404 HEAD@{12}: commit: Test
```

git stash listで示された内容と似ていて、すべての記録は、HEAD@{0},HEAD@{1}スタックのように保存されています。

どの状態に戻りたいのであれば、そのスタック番号を選べばよいです。その番号までのすべての操作が取り消されます。

```
$ git reset --hard HEAD@{5}
```

しかし、うっかり `git reset --hard` でクリアしてしまったWorkTreeの変更は、どう戻っても帰って来ませんので、気をつけてください。

10. 課題

1. リモートリポジトリ `git-practice` の `2948ccf` コミットから新しいブランチを切る。名前を `features/new-branch-自分の番号` に設定してください。
2. リモートリポジトリの状況を確認しましょう。マージ済みのブランチ、およびまだマージされていないリモートブランチの一覧を出してください。
3. マージ済みのリモートブランチ `remotes/origin/<何か>` を一つ選び、`remotes/origin/<何か>` ローカルブランチを作ってください。
4. `remotes/origin/<何か>` のローカルブランチで、以下のファイル `test.json` を作成し、変更をコミットしてください。

   ```
   {
     "Name": "testfile",
     "Value": "This is a test"
   }
   ```

5. 作成したコミットを `features/new-branch-自分の番号` に移動してください。
6. `remotes/origin/<何か>` のローカルブランチを消してください。
7. `test.json` の `Name` 項目をKeyに変更し、その変更をコミットしてください。変更されたファイルは以下となります。

   ```
   {
     "Key": "testfile",
     "Value": "This is a test"
   }
   ```

8. コミットのコミットメッセージを修正し、`CHANGE: json Name field` にしてください。
9. `features/new-branch-自分の番号` をPushしてください。
0. `features/new-branch-自分の番号` のブランチ名を `test/new-branch-自分の番号` に変更してください。
1. `features/new-branch-自分の番号` のベースコミットを `origin/master` の先頭にし、リモートブランチに更新してください。
2. `test.json` の作成コミットを修正し、`test.json` を以下のようにしてください。

```
{
  "Key": "testfile",
  "Value": "This is a test",
  "Tag": "new-json-tag"
}
```

3. 修正したコミットが含まれたまま、リモートブランチに更新してください。
4. `test.json` の `Tag` 項目の値を以下のように変更してください。

```
{
  "Key": "testfile",
  "Value": "This is a test",
  "Tag": "new-json-tag-test-value"
}
```

5. `test.json` の変更を `Index` と `Worktree` から削除してください。
6. 上記の操作を取り消してください。
7. `test.json` の変更を `Index` に追加し、そして変更を保ったままに `Index` から外してください。
8. `test.json` の変更を直前にコミットに追加してください。
9. 5番から追加されたコミットが3つになったはずです。その二つのコミットを一つに合併してください。