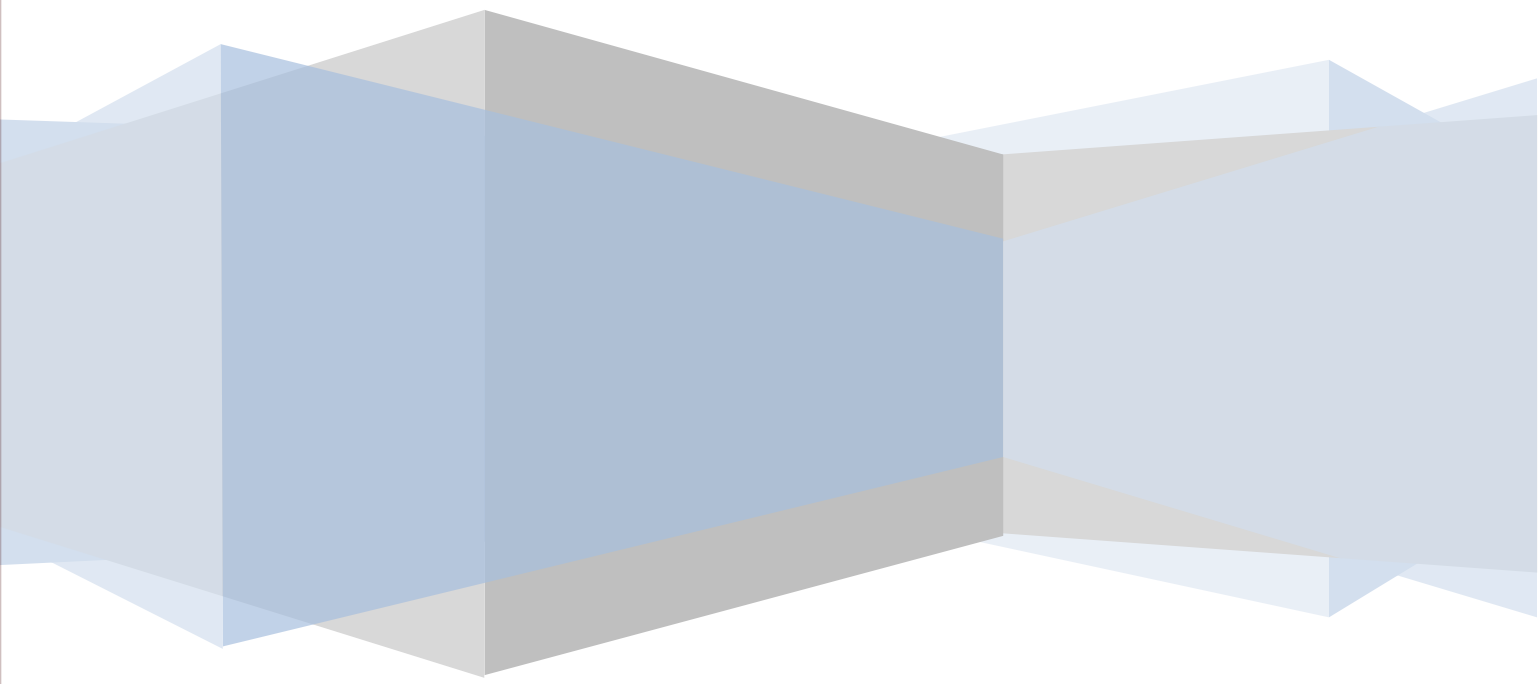
MEGAZONE CLOUD

Git & GitHub



1. Git 설치

- https://git-scm.com/download/win

2. Git Config

깃 환경설정파일 : C:/사용자/[User-Name]/.gitconfig

· git config --list

· User settings

git config --global user.name "name"
git config --global user.email "email"

git config user.name
git config user.email

· Set Auto CRLF (줄바꿈)

git config --global core.autocrlf true
git config --global core.autocrlf input

· git config --global -e

```
[diff]
    tool = vscode   # Visual Studio Code
[diff "vscode"]
    cmd = code --wait --diff $LOCAL $REMOTE
```

3. 깃 초기화

git init

4. 깃 저장소의 현재 상태 확인

git status

5. .gitignore 파일 추가

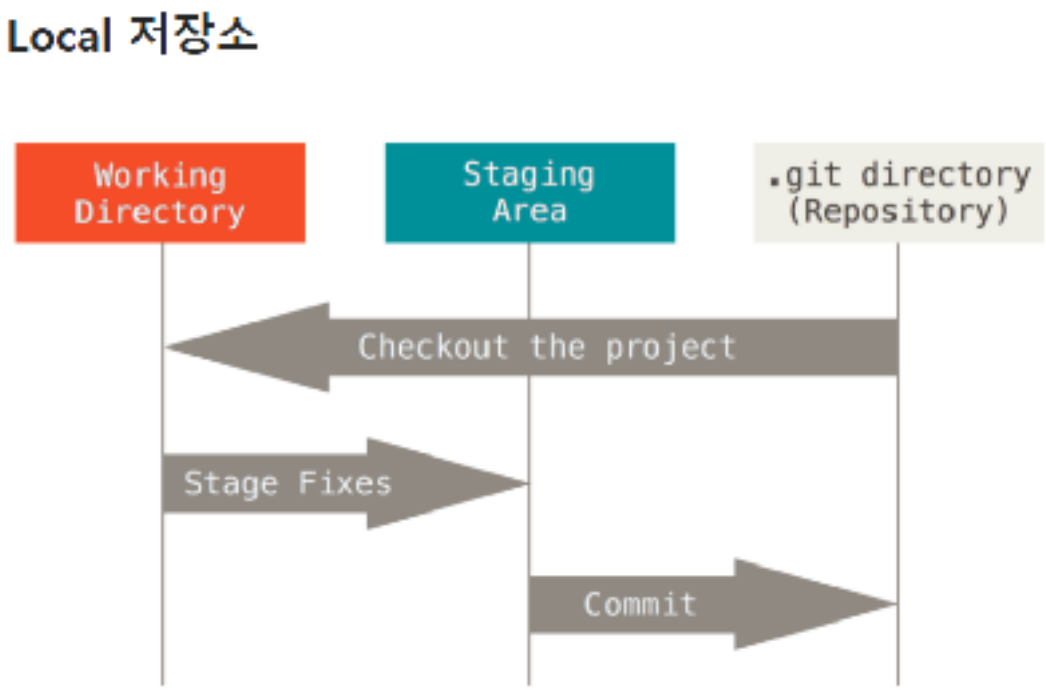
※ Gitignore 란?

깃에서 특정 파일 혹인 디렉토리를 관리대상에서 제외할 때 사용하는 파일이다.

자동으로 생성되는 로그파일, 프로젝트 설정 파일등을 관리 대상에서 제외할 수 있다.

· https://www.toptal.com/developers/gitignore 사이트 참조

● Git 은 Git Directory, Working Directory, Staging Area 세가지 영역으로 분리된다.



● Working Directory

1. untracked

· 아직 tracking 이 되지 않은 파일이다.

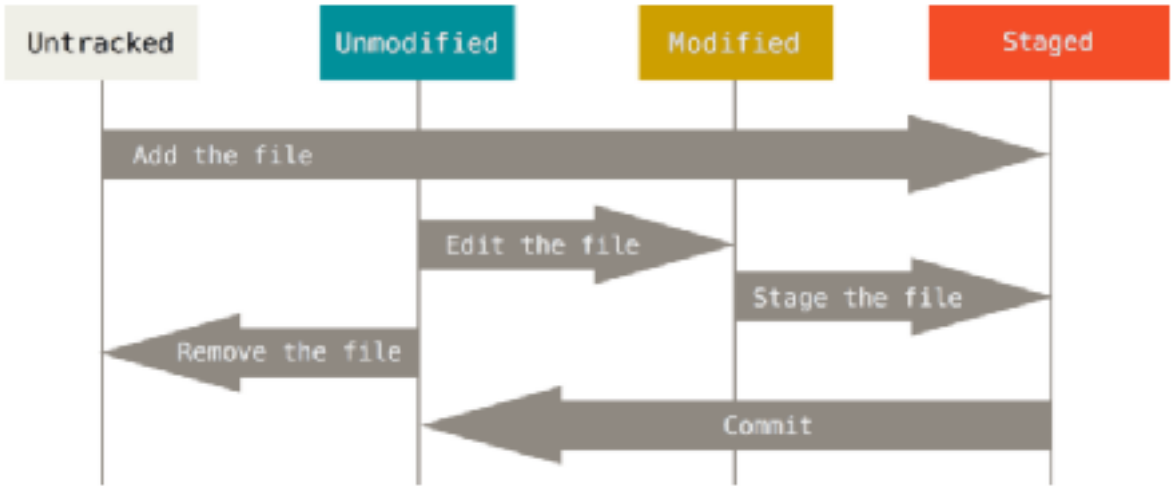
· 기존에 존재하던 프로젝트에서 git 을 초기화하거나 파일을 새로 만들면(또는 처음 저장소를 clone 하면) untracked 상태이다.

2. tracked

· unmodified/modified 로 나눌 수 있다.

· checkout 된 이후 수정사항이 있지만 stage 되지 않으면 modified 된 상태이다.

· modified 된 파일만 staging area 로 옮겨갈 수 있다.



● Staging Area (Index)

tracked & staged 상태

add 명령어를 통해 파일을 Staging Area 에 올릴 수 있다.

git rm --cached 를 이용하면 unstage(Staging Area-> Working Directory)가 가능하다.

git diff --cached 는 index 와 HEAD 사이의 변화를 보여준다.

● .git directory

commit 명령어를 실행하면 Staging Area 에 있는 파일들이 하나의 버전으로서

git directory 에 저장된다.

commit 후에는 파일 상태가 staged 에서 unmodified 로 변경된다.

※ 깃 명령어

$ git add

· 작업 디렉토리(working directory) 상의 변경 내용을 스테이징 영역(staging area)에 추가하기 위해서 사용하는 Git 명령어이다.

```
git add a.txt          # stage a.txt file
git add a.txt b.txt    # stage a.txt, b.txt files
git add *.txt          # stage all files ends with .txt
git add .              # stage all files
```

● 파일 삭제

rm a.txt //a.txt 파일을 삭제한다.

git rm atxt

작업 디렉토리나 스테이징 영역에 있는 파일을 삭제 할 때 사용한다.

git rm --cached a.txt

스테이징 영역에 있는 파일만 삭제 할 때 사용한다.

git clean -fd

untracked 파일들을 삭제할 때 사용한다.

$ git mv 기존 파일명 새 파일명

working directory 와 staging are 에 있는 파일의 이름을 변경 할 때 사용한다.

명령어를 실행하면 tracked 되어 staging 영역에 올라가며 commit 대기 상태가 된다.

commit 을 하면 파일 이름이 변경된다.

$ git diff - 파일 내용 비교

working directory 와 staging area 간의 비교도 가능하고 commit 간의 비교, branch 간의 비교도 가능하다.

· git diff

working directory 에 있는 파일 중에서 변경된 파일의 내용 비교한다.

· git diff --staged

staging area 에 있는 파일 중에서 변경된 파일의 내용 비교한다.

git diff [비교할 commit 해쉬 1] [비교할 commit 해쉬 2]

commit 간의 상태 비교하기 - commit hash 이용

commit 간의 상태 비교하기 - HEAD 이용

git diff [브랜치명] [브랜치명]

ex) git diff feature/test origin/master

local 의 feature/test 브런치와 remote 의 master branch 비교한다.

● Visual Studio Code 를 Git diff tool 로 설정하기

https://medium.com/@hohpark/vs-code%EB%A5%BC-git-diff-tool%EB%A1%9C-%EC%84%A4%EC%A0%95%ED%95%98%EA%B8%B0-88baa1d9f2b3

· git config --glabal -e

```
[diff]
    tool = vscode
[difftool "vscode"]
    cmd = code --wait --diff $LOCAL $REMOTE
```

$ git commit - 버전 등록

· git commit

staging area 에 있는 파일을 커밋한다.

· git commit -m "Commit message"

staging area 에 있는 파일을 커밋 메시지와 함께 커밋한다.

· commit -am "Commit message"

working directory (tracked)에 있는 파일들을 스테이징 절차(add)를 생략하고 바로 add 와 commit 을 동시에 하는 것도 가능하다.

$ git log - 버전(커밋) 히스토리 보기

· git log

저장소의 커밋 히스토리를 시간순으로 보여준다

· git log --oneline --graph --all

· git log --patch

각 커밋의 diff 결과를 보여준다. 다른 유용한 옵션으로 `-2`가 있는데 최근 두 개의 결과만 보여주는 옵션이다.

· git log --oneline #oneline

· git log --oneline --reverse

● Log Format

$ git log --graph --all --pretty=format:'%C(yellow)[%ad]%C(reset) %C(green)[%h]%C(reset)

| %C(white)%s %C(bold red)(%an)%C(reset) %C(blue)%d%C(reset)' --date=short

%ad : author date

%h : commit hash

%an : author

%d : ref names

%s : commit subject

// 별칭

git config --global alias.hist ” ”

● 브랜치 기본 사용법

브랜치란?

· 각자 독립적인 작업 영역(저장소)

  즉, 여러 사람이 동일한 소스코드를 기반으로 서로 다른 작업

· 분리된 작업 영역에서 마음대로 소스코드 변경

· 작업이 끝난 사람은 머지를 사용하여 메인 브랜치에 자신의 브랜치의 변경사항을 적용한다.

● Git-flow 브랜치 전략

·

https://velog.io/@jinuku/Git-%EB%B8%8C%EB%9E%9C%EC%B9%98-%EC%A0%84%EB%9E%B5

master : 제품으로 출시될 수 있는 브랜치

develop : 다음 출시 버전을 개발하는 브랜치

feature : 기능을 개발하느 브랜치

release : 이번 출시 버전을 준비하는 브랜치

hotfix: 출시 버전에서 발생하는 버그를 수정하는 브랜치

● 브랜치 생성 (Local Repository)

$ git branch {브랜치명}

Local Repository 에 새로운 브랜치를 생성한다.

$ git checkout -b {브랜치명}

Local Repository 에 새로운 브랜치를 생성과 동시에 체크아웃도 가능하다.

$ git swich -C {브랜치명}

Local Repository 에 새로운 브랜치를 생성과 동시에 브랜치로 이동한다.

● 브랜치 생성 (Remote Repository)

$ git remote add origin https://github.com/javaenjoy/repo

원격저장소를 지정한다.

$ git push origin 브랜치명

● 브랜치 목록 조회

$ git branch (로컬 브랜치 목록 조회)

$ git branch -r (원격 브랜치 목록 조회)

$ git branch -a (모든 브랜치 목록 조회)

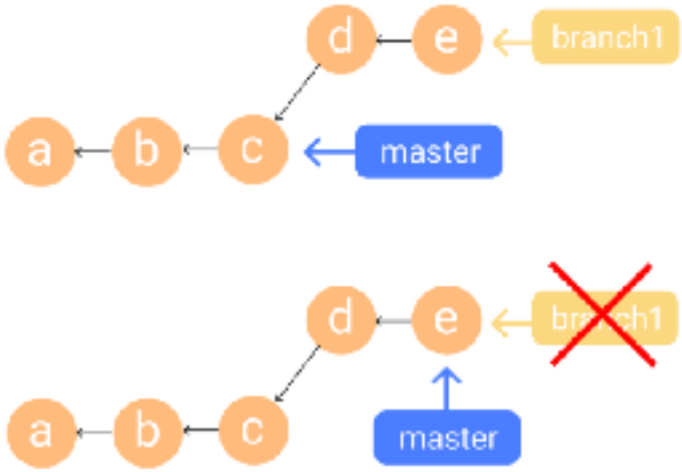
● 브랜치 병합 전략

1. Merge

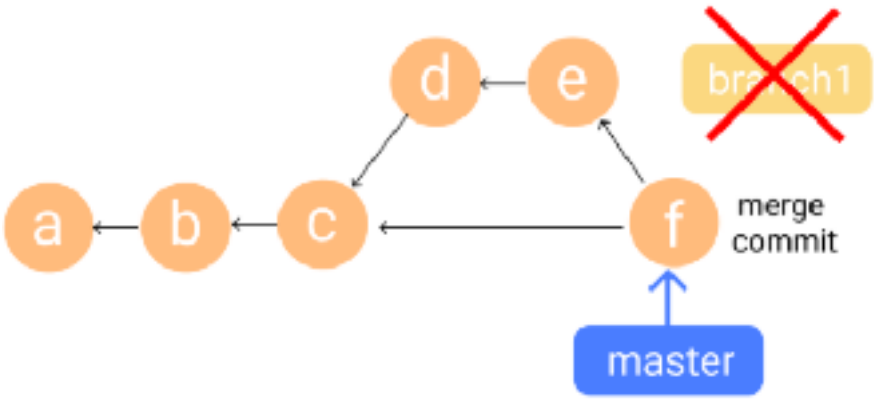
2. Rebase

● Merge

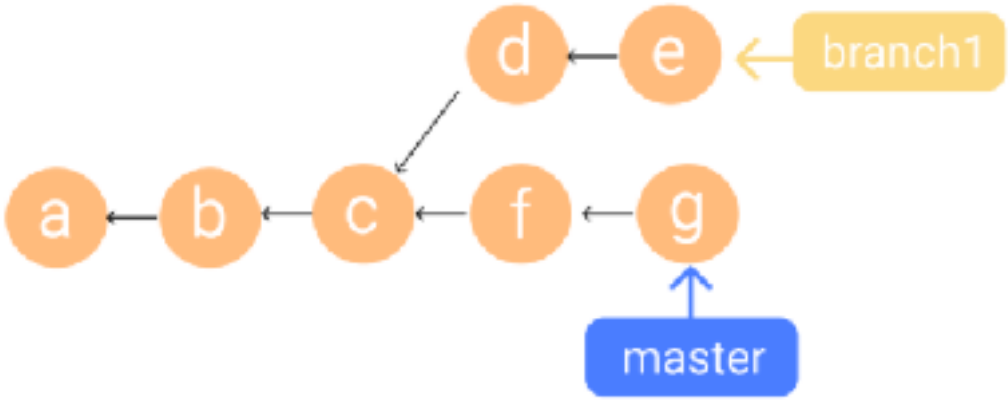
1. fast-forward merge



2. 3-way-merge



3. fast-forward merge 가 불가능한 경우



● 브랜치 병합(merge)

$ git merge {브랜치명}

● 브랜치 병합(merge) 취소

$ git merge --abort : 머지 이전 상태로 돌아간다.

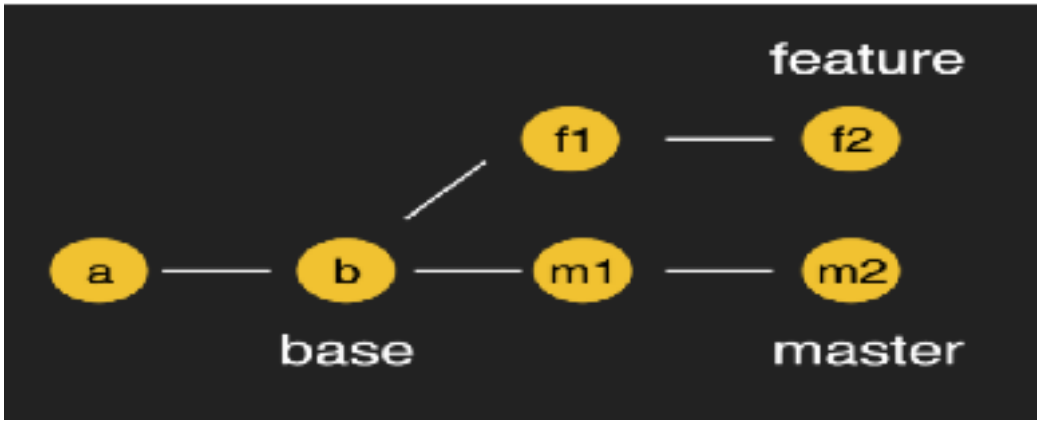
● Visual Studio Code 를 git merge tool 로 사용하기

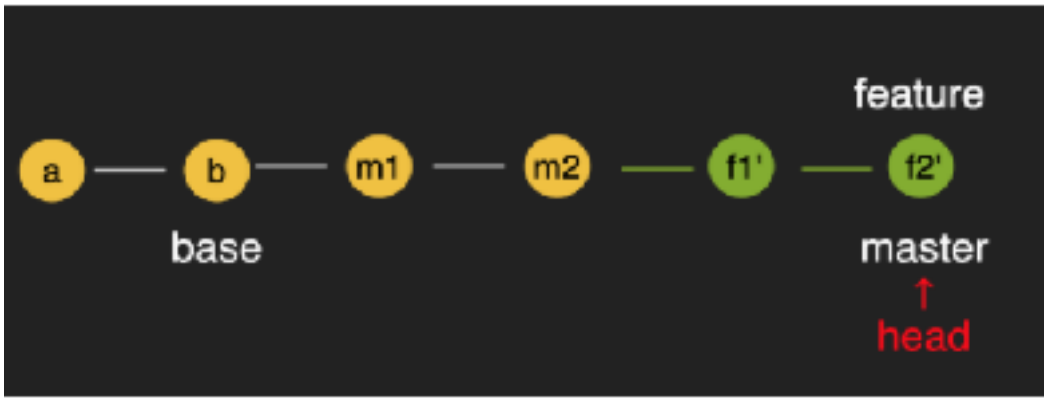
```
[merge]
  tool = vscode
[mergetool "vscode"]
  cmd = code --wait $MERGED
```

```
$ git config --global mergetool.keepBackup false
```

● 브랜치 병합(merge) conflict

● Rebase



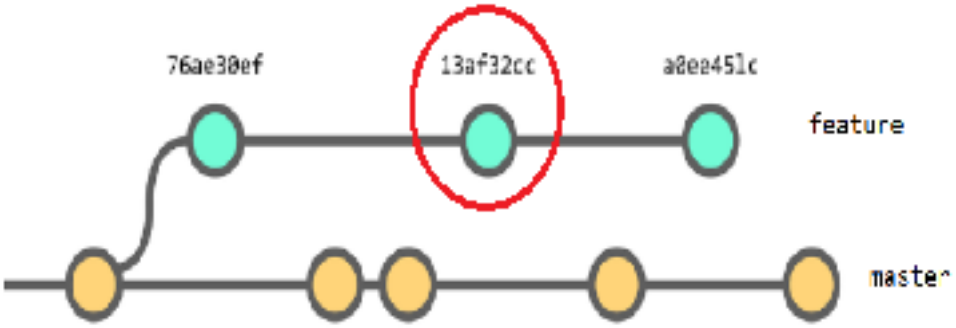


● Rebase 의 위험성

이미 원격 저장소에 Push 된 커밋은 Rebase 를 하지 않는 것이 일반적이다.

동일한 브랜치에서 다른 개발자와 작업을 하는경우 Rebase 를 해서는 안된다.

● cherry-pick 으로 원하는 commit 가져오기



$ git cherry-pick <commit_hashcode>

- 다른 브랜치에 있는 커밋을 선택적으로 내 브랜치에 적용시킬때 사용한다.

팀으로 협업할 때

사이드 프로젝트를 진행하고 있다. 나는 백엔드다. 현재 내가 맡은 기능이 프론트까지 더해졌을 때 어떻게 돌아가는지 확인하고 싶은데 프론트엔드를 맡은 팀원이 지금 한창 작업 중이라고 한다. 아까 CSS 파일을 완성해서 commit 해 뒀다고 한다. 그럼 난 cherry-pick 을 통해 그 commit 하나만 찾아서 내 branch 에 가져오면 된다. CSS 만 입혀볼 수 있게 됐다.

버그 수정

나는 며칠 전 A 기능을 완성한 개발자다. A 가 끝나고 B 기능을 개발하는 중이었는데, 알고 보니 A 기능에 버그가 있다고 한다. 더 많은 유저들이 이 버그로 인한 현상을 겪기 전에 빠르게 버그 패치를 해서 내 branch 에 commit 한다. 내가 방금 commit 한 이 패치는 git cherry-pick 을 통해 main branch 에 바로 반영되었다.

반영되지 않은 pr

실수로 pull request 를 merge 하기 전에 닫아버렸다. 당황하지 않고 git cherry-pick 을 통해 해당 commit 을 가져옴으로써 살릴 수 있게 됐다.

● Git 으로 실행취소(Undo) 하는 방법

$ git restore test.txt

working directory 에서 수정한 파일 되돌리기(undo) 명령어이다.

$ git restore --staged test.txt

staging area 에서 수정한 파일 되돌리기(undo) 명령어이다.

● reset

reset 은 아예 현재가 없었던 것 처럼 원하는 과거로 돌아갈 수 있다. 정말 말 그대로 '리셋'이다. reset 은 이력을 남기지 않는다. 따라서 현재까지의 commit 이력을 남기지 않고 원하는 시점으로 완전히 되돌아가고 싶을 때 사용할 수 있다.

$ git reset --soft [commit ID]

$ git reset --mixed [commit ID]

$ git reset --hard [commit ID]

$ git reset HEAD~10

$ git reset HEAD^

· soft : commit 된 파일들을 staging area 로 돌려놓음.

· mixed(default) : commit 된 파일들을 working directory 로 돌려놓음.

· hard : commit 된 파일들 중 tracked 파일들을 working directory 에서 삭제한다.

(Untracked 파일은 여전히 Untracked 로 남는다.)

$ git reset --hard HEAD^

· HEAD~취소할커밋수 : 현재로부터 원하는 만큼의 커밋이 취소된다.

· HEAD^ : 가장 최근의 커밋이 취소된다. (기본옵션 mixed)

● revert

현재에 있으면서 (히스토리 정보는 그대로 두고) 새로운 커밋을 만들면서 과거의 커밋 내용을 변경(undo)하는 것이다.

※ 이미 깃 허브 마스터 브랜치에 push 가 된 경우는 revert 를 사용해야 한다

● Git 복구

$ git reflog

git rebase 또는 git reset 등으로 커밋이 삭제될 수 있다.

하지만, git 이력은 보관되고 있는데 이러한 이력을 볼 수 있는 명령어가 git reflog 이다.

1. commit 복구하기

git reflog 명령어로 삭제된 commit id 확인 후

git reset --hard <커밋해시 id>

2. branch 복구하기

git reflog 또는 git reflog |grep 브랜치명 으로 log 확인

git switch -C <삭제한 브랜치명> <커밋해시 id>

● git rebase -i [수정을 시작할 커밋의 직전 커밋]

커밋한 히스토리를 변경하거나 또는 삭제하거나, 내용을 추가할 때 사용하는 명령어이다.

· pick

pick 은 커밋을 사용하겠다는 의미이다.

이를 이용하여 커밋의 순서를 바꿀수도 있고, 커밋의 해쉬값을 이용하여 특정 커밋을 가져올 수도 있다.

· reword

reword 는 커밋 메시지를 변경하는 명령이다.

· edit

edit 은 커밋 메시지 뿐 아니라 커밋의 작업 내용도 변경할 수 있는 명령이다.

· squash

squash 는 해당 커밋을 이전 커밋과 합치는 명령어이다.

· drop

drop 은 커밋 히스토리에서 커밋을 삭제하는 명령어입니다.

주의사항

$ git rebase 는이전의 커밋 히스토리를 변경하기 때문에 항상주의하여 사용해야 한다.