Git 기본 설정

1. 사용자 등록

- `git config`: **사용자 정보**를 설정
- `--global`: 저장소를 생성할 때마다 동일한 설정을 하려면 번거러울수 있으므로, **로컬에 공통으로 적용**

```
git config --global user.name "(본인 이름)"
git config --global user.email "(본인 이메일)"
git config --list
```

2. git 관리하에 넣기 + 상태 확인

- `git init`: **빈 git 저장소가 초기화**되었다는 문구가 출력 (`Initialized empty Git repository`)
  - `.git`이라는 폴더가 생성됨 (앞으로 작성할 **프로그램의 저장 이력**이 기록되는 곳)
- `git status`: 저장소 내의 **파일 상태정보**를 출력
  - (1) git 분류 기준에 따른 파일의 **상태정보(untracked, modified, unmodified)**를 보여줌
  - (2) 파일의 **staging 여부**도 확인

```
git init
git status
```

3. `.gitignore` 사용 문법

```
# [1] (일부/전부) file.py 무시하기
file.py # file.py & A/file.py & B/C/file.py 모두 무시
/file.py # (root 경로 하의) file.py만 무시됨
#-----------------------------------------------------------------#
# [2] 특정 확장자 전부
*.py
#-----------------------------------------------------------------#
# [3] 해당 파일은 절대 무시 X
!do_not_ignore.py
#-----------------------------------------------------------------#
# [4] (모든 경로의) logs 파일 & 폴더 무시
# 4-1) 파일+폴더
logs
# 4-2) 폴더
logs/
# 4-3) 파일
logs
```

```
!logs/
#-------------------------------------------------------------#
# [5] 특정 폴더 하의 모든 파일
logs/certain_file.py # logs하의 파일 무시
logs/**/certain_file.py # logs하+하위 경로의 파일 무시
logs/*.py # logs하의 파일 무시
logs/**/*.py # logs하+하위 경로의 파일 무시
#-------------------------------------------------------------#
```

4. `.gitignore` 사용 문법 (심화: contain, 접두사)

```
# (1) "파일"명에 abc가 있는 파일 무시
*abc*
!*/

# (2) "폴더"명에 abc가 있는 폴더 무시
*abc*/

# (3) "파일 및 폴더"명에 abc가 있는 파일 및 폴더 무시
*abc*

#-----------------------------------------------------------------------#
# (4) "파일"명에 abc로 시작하는 파일 무시
abc*
!*/

# (5) "폴더"명에 abc로 시작하는 폴더 무시
abc*/

# (6) "파일 및 폴더"명에 abc로 시작하는 파일 및 폴더 무시
abc*
```

5. git add & commit

```
git add tigers.yaml
git commit -m "FIRST COMMIT"
```

6. 되돌리기

- **reset** : 원하는 시점으로 돌아간 뒤 이후 ***내역들을 "지웁니다".*** (사실상 복구 불가)
- **revert** : 되돌리기 원하는 시점의 커밋을 ***"거꾸로" 실행***합니다.

```
git log

git reset --hard (돌아갈 커밋 해시)

git revert (되돌릴 커밋 해시)
git revert --no-commit (되돌릴  커밋 해시) # 원하는 다른 작업을 추가한 다음 함께 커밋
```

7. 브랜치

```
# (1) 브랜치 확인
git branch

# (2) 브랜치 생성 & 전환
git branch new_branch
git switch new_branch

# (3) 브랜치 생성+전환
git switch -c new_branch
# git checkout -b new_branch

# (4) 브랜치 제거
git branch -d new_branch
git branch -D (강제삭제할 브랜치명) # 강제 삭제

# (5) 브랜치명 변경
git switch -m new_branch newname_branch
```

8. 브랜치 합치기

- `git merge` : 두 브랜치를 ***"한 커밋"***에 이어 붙임
  - ***브랜치 사용내역을 남길 필요가 있을 때*** 적합한 방식!
  - `merge`도 일종의 **"하나의 커밋"**이므로, `reset`을 통해 merge이전으로 되돌리기 가능!
  - 병합된 이후, 기존 브랜치는 삭제하기!
- `git rebase`: 브랜치를 ***"다른 브랜치"***에 이어붙임
  - ***한 줄로 깔끔히*** 정리된 내역을 유지하기 원할 때!
  - 이미 팀원과 공유된 커밋들에 대해서는 사용하지 않는 것이 좋습니다.

```
# git switch main
git merge add-coach
git branch -d add-coach
```

```
# git swith new-teams # 반대임!
git rebase main
git switch main # rebase로 끝난게 아니라, switch 이후 merge 해줘야!
git merge new-teams
```

9. 브랜치 합칠 때 충돌 해결

git merge 충돌

```
# git switch main

# 어라 오류?
git merge conflict1

# case 1) 일단 중단
git merge --abort

# case 2) 충돌 부분 해결 후,
git add .
git commit -m 'Merge'

#-----------------#
git branch -d conflict1
```

git rebase 충돌

```
# git switch conflict2

# 어라 오류?
git rebase main #

# case 1) 일단 중단
git rebase --abort

# case 2) 충돌 부분 해결 후,
git add .
git rebase --continue
git add .
git rebase --continue
git switch main
git merge conflict2

#-----------------#
git branch -d conflict2
```

10. github

- Git으로 관리되는 프로젝트의 원격 저장소
- 오픈소스의 성지
    - Git, VS Code, Tensorflow, React 등 살펴보기

11. github 실습 1

새로운 git repository 만들기

```
# [Step 1] 로컬의 Git 저장소에 원격 저장소로의 연결 추가
# 원격 저장소 이름에 흔히 origin 사용. 다른 것으로 수정 가능
git remote add origin (원격 저장소 주소)

# [Step 2] 기본 브랜치명을 main으로
git branch -M main

# [Step 3] 로컬 저장소의 커밋 내역들 원격으로 push(업로드)
# -u 또는 --set-upstream : 현재 브랜치와 명시된 원격 브랜치 기본 연결
git push -u origin main
```

`git push -u origin main`로 인해, 앞으로

- `git push origin main` 대신, `git push`만 해줘도되고
- `git pull origin main` 대신, `git pull`만 해줘도된다.

```
# 원격 목록 보기
git remote
```

12. github 실습 2

```
git clone (원격 저장소 주소)
```

13. pull을 한 후에 push하기!

- 사람 1: A 파일을 x로 수정
- 사람 2: A 파일을 y로 수정

이런 상황에서는??

```
# 사람1: x로 수정 & add & commit (cm-A)
# 사람2: y로 수정 & add & commit (cm-B) & push

# 사람1: push => Conflict 발생!
git push

# pull 부터 해줘야
git pull --no-rebase # merge 방식
git pull --rebase # rebase 방식 (cm-B => cm)

git push
```

14. 로컬의 내역 강제로 push

```
git push --force
```

15. 로컬에서 브랜치 만들어 "원격에 올리기"

```
git branch from-local
git switch from-local

git push -u origin from-local # 한번만 해주면, 이후에는 git push 로
```

```
# (원격 포함) 모든 브랜치 목록 확인
git branch --all
```

- 상황 요약: local & remote의 main & from-local이 각각 존재하는 상황 (총 4개)

16. 원격의 브랜치를 로컬에 받아오기

다른 유저가 새로운 브랜치를 생성한 뒤, 원격에 올렸음

(아래 코드: by 다른 유저)

```
git branch from-remote
git switch from-remote
git push -u origin from-remote
```

원격의 branch 받아오기

```
git fetch
```

- `git branch -a`를 하면, "origin/from-remote"가 생긴 것을 확인할 수 있다!

로컬에도, 해당 "origin/from-remote"와 연결되는 새로운 브랜치 생성하기

- switch까지도 해줌!

```
git switch -t origin/from-remote
```

- main / from-local / from-remote x 2 (원격&로컬) = 6개가 존재하는 것 확인 가능!

17. 원격의 브랜치 삭제하기

```
git push (원격 이름) --delete (원격의 브랜치명)
```

18. 세 가지 작업 영역 (요약)

- Working directory
  - untracked: Add된 적 없는 파일, ignore 된 파일
  - tracked: Add된 적 있고 변경내역이 있는 파일
  - `git add` 명령어로 Staging area로 이동
- Staging area
  - 커밋을 위한 준비 단계
    - 예시: 작업을 위해 선택된 파일들
  - `git commit` 명령어로 repository로 이동
- Repository
  - `.git directory`라고도 불림
  - 커밋된 상태

18. 세 가지 작업 영역 (상세)

- **Working directory** : 이력관리 대상(tracked) 파일들이 위치하는 영역
  - 지정된 디렉토리에서 `.git` 폴더를 제외한 공간
  - 작업된 파일이나 코드가 저장되는 공간
  - 주의: `.git` 폴더가 Repository에 해당한다.

        - 흔히 저장소(Repository)를 폴더 전체를 통칭하기도 하는데, 엄밀히 따지면 `.git` 폴더가 Repository이다!
        - 하지만 앞으로 저장소를 언급하면 상위 폴더 전체를 언급하는 것으로 이해하자!
- **Staging area** : 이력을 기록할, 다시 말해 `commit`할 대상 파일들이 위치하는 영역
    - `.git` 폴더 하위에 파일형태로 존재(`index`)
    - Q) 수정된 파일을 바로 commit하면 효율적일텐데 중간에 staging 과정을 추가한 이유는 무엇일까?

      A) **일부 파일만 commit**: 수정된 전체 파일들중 일부만 선별적으로 commit해야하는 상황이 발생할 수 있다. 이 때 원하는 파일만 선별하기 위해!
- **Repository** : 이력이 기록된(committed) 파일들이 위치하는 영역
    - `.git` 폴더에 이력관리를 위한 모든 정보가 저장, 관리됨

19. `git status -s` (--short)

```
km.yu99@DESKTOP-G0FB1FE MINGW64 /c/SimpleTest (master)
$ git status -s
?? test.txt
```

- `??` : untracked
- `M` : modified
- `MM` : 수정상태가 staged 된 후, 다시 modified
- `A` : 경로가 staged 된 후, 경로내에 untracked 파일 발생

20. `git status`의 출력 결과물 확인

`ReadMe.md` 파일 변경함.

`git status`의 결과

```
On branch master

No commits yet

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

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

- `On branch master` : 현재 master branch에 작업중이라는 의미
- `No commits yet` : 현재 commit 없음
- `Untracked files` : 추적(관리) 중이 파일 외에, 새로 추가된 파일을 표시

    - 해당 파일을 commit을 하려면 `git add` 명령을 실행하라고 힌트도 주고 있다.
- `nothing add to commit ~` : commit할 파일이 없다고 설명
    - 이유: 현재 Staging area에 위치한 파일이 없기 때문!

`git add ReadMe.md`

`git status`의 결과

```
On branch master

No commits yet

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

- `Changes to be committed` : Staged된 (commit을 진행할) 파일이 있음을 알려줌
    - 앞서 수정한 `ReadMe.md`파일에 해당됨
    - 해당 파일을 Unstage(파일을 modified 상태로 내리기 위한) 하기 위한 커맨드도 설명

`git commit -m "Add created date in ReadMe.md"`

```
[master (root-commit) f73cb01] Add ReadMe.md
 1 file changed, 1 insertion(+)
 create mode 100644 ReadMe.md
```

`git status`의 결과

```
On branch master
nothing to commit, working tree clean
```

- `nothing to commit, working tree clean` : commit할 파일이 없다!
    - Woring directory의 파일들이 unmodified 상태임!

21. 파일의 삭제와 이동

`git rm tigers.yaml` = Step 1+2

- Step 1) `tigers.yaml` 삭제하기
- Step 2) `git add .`

`git mv tigers.yaml tigers2.yaml` = Step 1+2

- Step 1) `tigers.yaml` 의 이름을 `tigers2.yaml`로 바꾸기
- Step 2) `git add .`

22. `git restore`

working directoy에서 staging area로 올렸는데..

어! 특정 파일은 나중에 commit하고 싶은데... staging area에서 내릴 수 없을까/

```
# case 1) unstage (파일 변화는 X)
git restore --staged file.py
# case 2) unstage + working dir에서도 제거 (즉, 변경사항도 되돌리기)
git restore file.py
```

23. `git reset`

복습) **reset** : 원하는 시점으로 돌아간 뒤 이후 ***내역들을 "지웁니다".*** (사실상 복구 불가)

reset도 세 가지 종류가 있음

- `--soft`: `repository`에서 `staging area`로 이동
- `--mixed (default)`: `repository`에서 `working directory`로 이동
- `--hard`: 수정사항 완전히 삭제 (아예 파일도 원래대로 바뀌게됨!)

24. HEAD

HEAD란? 현재 속한 브랜치의 가장 최신 커밋

`checkout`을 사용하여, 특정 브랜치의 앞/뒤 커밋으로 이동하기

```
# (1) HEAD 사용하기
git checkout HEAD^
git checkout HEAD^^
git checkout HEAD^2
git checkout HEAD~
git checkout HEAD~~
git checkout HEAD~2

# (2) Commit Hash 사용하기
git checkout (커밋해시)
```

이전 n단계로 돌아간 뒤, `git branch`를 쳐보면..

익명의 브랜치에 위치함을 알 수 있음!

```
git branch
```

기존 브랜치(의 맨 끝/최신)으로 다시 돌아오려면

```
git switch (브랜치명)
```

HEAD를 사용해서 reset할 수도 있음

```
git reset HEAD(원하는 단계) (옵션)
```

이전 n 단계 돌아간 뒤, 거기서 새로운 branch를 생성할 수도 있음

```
git checkout HEAD~
git switch -c new_branch
```

25. fetch vs. pull

- `fetch`: 원격 저장소의 최신 커밋 => 로컬로 가져오기
- `pull`: 원격 저장소의 최신 커밋 => 로컬로 가져오기 + 가져온 것을 `merge` 또는 `rebase`

그러면, 왜 굳이 fetch?

→ 내 main에 merge하기 이전에, 일단 살펴본 뒤 합치고 싶어서!

```
# 일단 fetch해온 것으로 넘어가서 점검해본 뒤
git checkout origin/main

# 그다음, 괜찮다 싶으면 pull하기
git pull
```

26. 원격의 새로운 브랜치 확인 + 받아오기

```
# 새로운 브랜치 받아오기 (pull까지는 아님)
git fetch

# 새로운 (원격) 브랜치가 생긴 것을 확인할 수 있음
git branch -a

# 새로운 브랜치로 일단 넘어가서 좀 살펴보고, 괜찮다 싶으면...
git checkout origin/new_branch

# OK! 내 로컬에도 받아오자!
git checkout main
git switch -t origin/new-branch

# 새로운 (원격) 브랜치와 동일한 (로컬) 브랜치가 생긴 것을 확인할 수 있음
git branch -a
```

27. `git log`

- 저장소에 **기록된 이력(commit history)**를 출력
- 맨 아래: **"최초"**의 commit ~ 맨 위: **"최신"**의 commit
- 구성:
  - commit ID: `5858e921....`
  - 작성자, 작성일자, commit 메시지
- 가장 최근에 작성한 commit: `(HEAD -> master)`라는 문구

  = 가장 최신 commit이라는 뜻!

```
km.yu99@DESKTOP-G0FB1FE MINGW64 /c/SimpleTest (master)
$ git log
commit 5858e921bc9294d7e2a7dc6b7b97ab3011b78c05 (HEAD -> master)
Author: User1 <User1@gmail.com>
Date:   Mon Oct 18 14:14:11 2021 +0900

    Add listbox and modify OnBnClickedMsgBtn function

commit cd9938f779744e6b9ac254d558deb5f2d5588d2a
```

```
Author: User1 <User1@gmail.com>
Date:   Fri Oct 15 14:29:47 2021 +0900

    Add OnBnClickedMsgBtn() function

commit e5b578b3c060fe26385c14a6cd9a57100e90d120
Author: User1 <User1@gmail.com>
Date:   Fri Oct 15 14:07:13 2021 +0900

    Apply .gitignore

commit 635225a9d36f99c4298a0de1ef4dfcdfa44cd16a
Author: User1 <User1@gmail.com>
Date:   Thu Oct 14 15:37:26 2021 +0900

    Create empty project
```