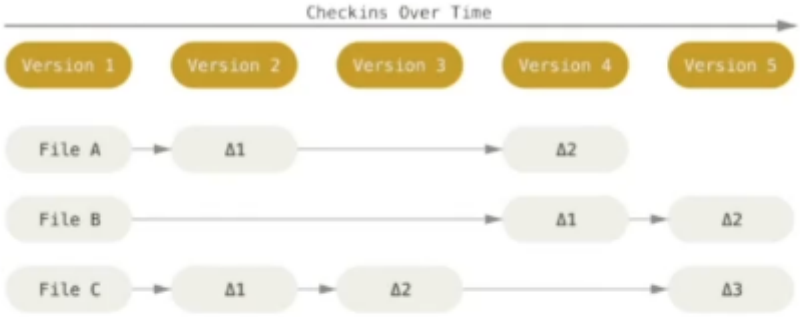
#git #github #coding #vcs

섹션 5. Git 보다 깊이 알기

S5-Lesson 1. 다른 VCS와의 차이점

- snapshot - 델타방식 vs 스냅샷 방식

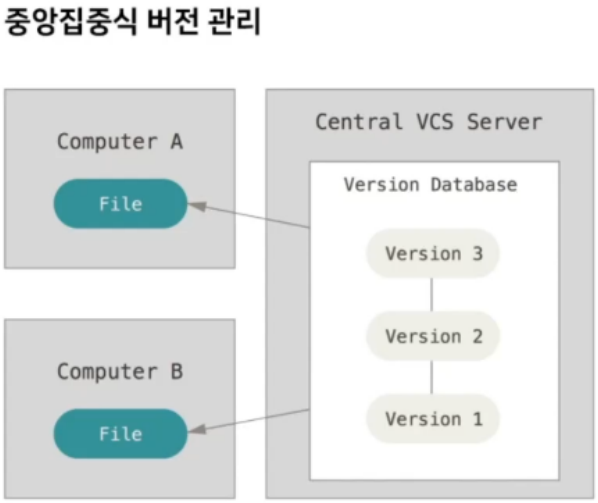
델타 방식

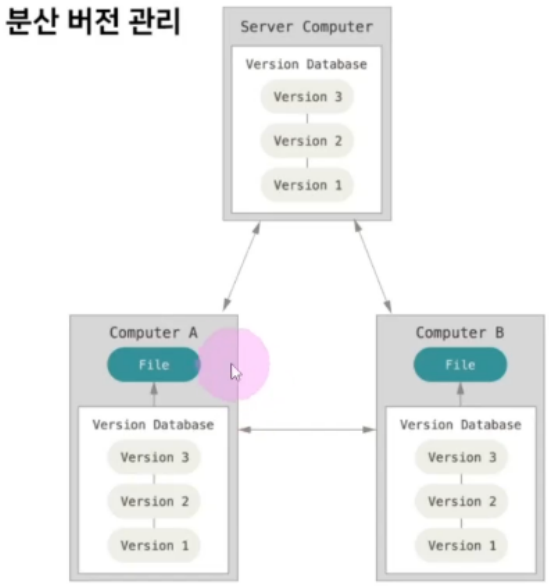


스냅샷 방식

버전 history가 길어질 수록 델타방식은 느려짐

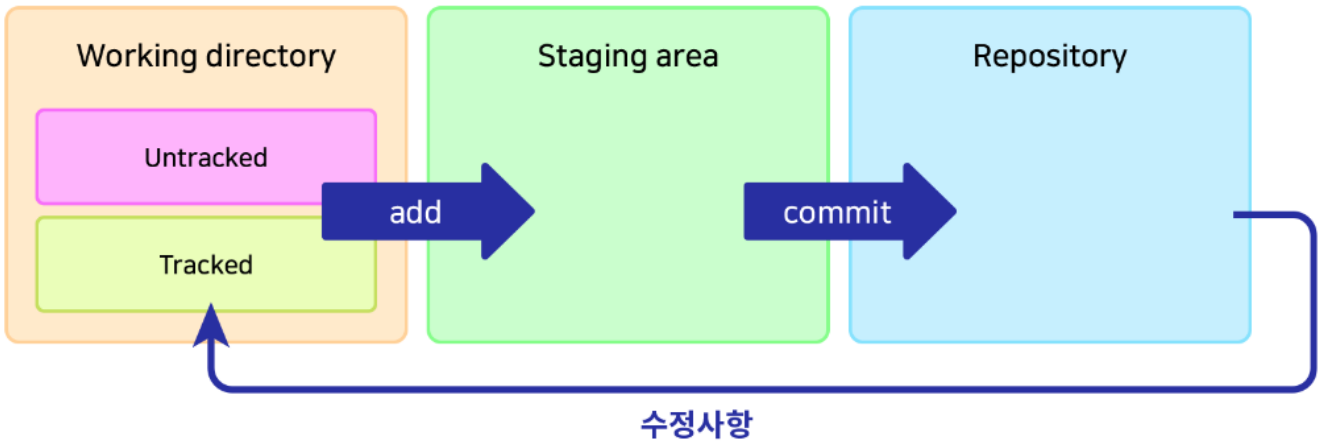
- 관리 방식





  인터넷 연결이 끊겨도 문제 없이 작동함

S5-Lesson 2. Git이 다루는 3가지 공간



- commit 되어 레포지토리에 들어간 후 수정사항이 발생하면 `tracked` 파일로써 스테이징을 기다린다.

1. Working directory

- Untracked : Add 된 적 없는 파일, ignore 된 파일
- Tracked : Add 된 적 있고 변경내역이 있는 파일
- `git add` 명령어로 Staging area로 이동

2. Staging area

- 커밋을 위한 준비 단계 (예: 작업을 위해 선택된 파일들)
- `git commit` 명령어로 repository로 이동

3. Repository

- `.git directory` 라고도 불림
- 커밋이 완료된 상태의 파일

어떤 파일을 **그릇**으로 비유한다면

상태	설명
untracked	식기세척기에 들어가 본 적이 없거나 식기세척기 사용이 불가(ignored)한 그릇
tracked	식기세척기에 들어가 본 적이 있고 식기세척기 사용이 가능한 그릇
add	식기세척기에 넣는 행위
staging area	식기세척기 안(에 들어간 상태)
commit	세척(식기세척기 가동)
repository	세척되어 깨끗해진 상태
파일에 수정이 가해짐	그릇이 사용되어 이물질(커밋되지 않은 변경사항)이 묻음
working directory	세척되어야 하는 상태

tracked가 된다는 건, Git의 관리대상에 정식으로 등록됨을 의미한다.
새로 추가되는 파일은 반드시 **add**해줌으로써, 해당 파일이 tracked될 것임을 명시해야 한다.
(Git이 새로운 파일을 무조건 다 관리하는 것을 방지하기 위하여)

4. 관련 명령어

A. git **rm** (파일 삭제)

- `tigers.yaml` 를 삭제해본 뒤 `git status` 로 살펴보기
- 파일의 삭제가 `working directory` 에 있음
- `git reset --hard` 로 복원
- `git rm tigers.yaml` 로 삭제하고 `git status` 로 살펴보기
- 파일의 삭제가 `Staging area` 에 있음
- `git reset --hard` 로 복원

B. git **mv** (이름바꾸기)

- `tigers.yaml` 를 `zzamtigers.yaml` 로 이름변경 뒤 `git status` 로 살펴보기
- 복원 후 `git mv tigers.yaml zzamtigers.yaml` 로 실행 뒤 비교

파일을 `staging area`에서 `working directory`로

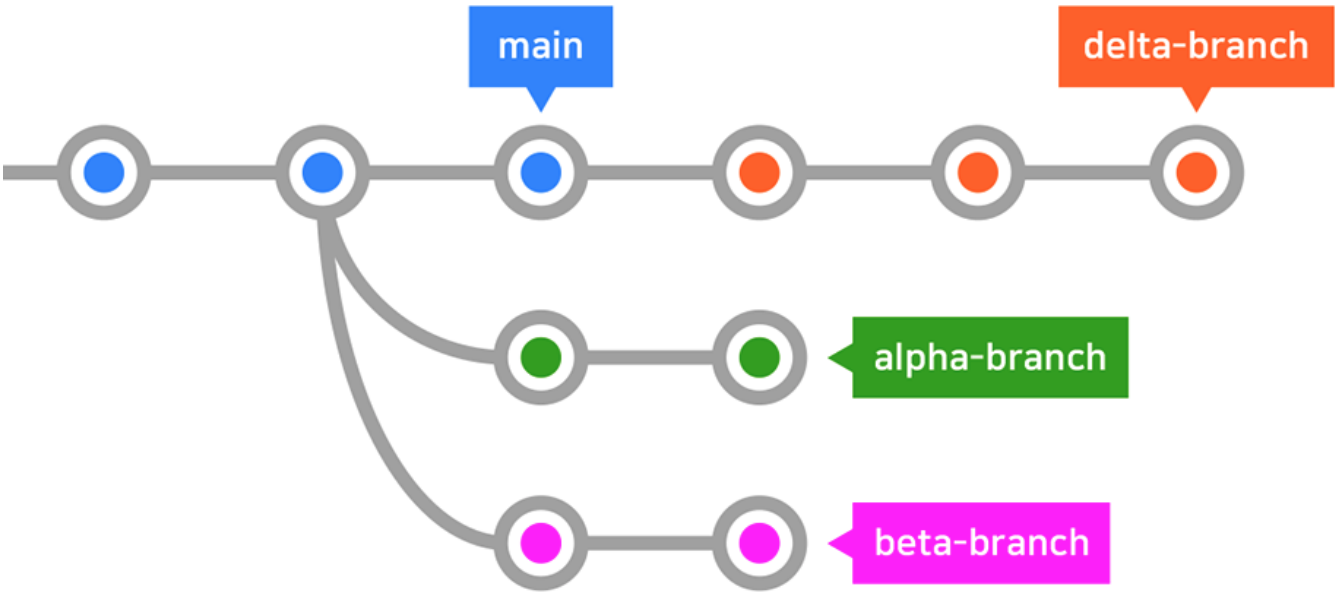
```
git restore --staged (파일명)
```

- `--staged`를 빼면 `working directory`에서도 제거
- 예전: `git reset HEAD (파일명)`

C. **reset**의 세 가지 옵션

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

S5-Lesson 3. HEAD



1. Git의 HEAD

현재 속한 브랜치의 가장 최신 커밋

- switch로 브랜치 이동해보기
- `main`과 `delta-branch`

2. `checkout`으로 앞뒤 이동하기

`git checkout HEAD^`

- `^` 또는 `~`: 갯수만큼 이전으로 이동
- `git checkout HEAD^^^`, `git checkout HEAD~5`
- **커밋 해시**를 사용해서도 이동 가능
- `git checkout (커밋해시)`
- `git checkout -` : (이동을) 한 단계 되돌리기

3. HEAD와 checkout 관계

익명의 브랜치에 위치함을 알 수 있음

- `checkout`으로 이전으로 돌아간 뒤
- 기존 브랜치로 돌아오기: `git switch (브랜치명)`
- 새 브랜치 만들어보기
- 새 커밋 만들어보기

4. HEAD 사용하여 reset하기

```
git reset (옵션) HEAD(원하는 단계)
```

S5-Lesson 4. Fetch vs Pull

1. fetch와 pull의 차이

- `fetch`: 원격 저장소의 최신 커밋을 로컬로 가져오기만 함
- `pull`: 원격 저장소의 최신 커밋을 로컬로 가져와 `merge` 또는 `rebase`

2. `fetch` 한 내역 적용 전 살펴보기

A. 원격의 `main` 브랜치에 커밋 추가
* `git checkout origin/main` 으로 확인해보기

B. 원격의 변경사항 `fetch`
`git checkout origin/main` 으로 확인해보기
`pull` 로 적용

C. 원격의 새 브랜치 확인

- `git checkout origin/(브랜치명)`
- `git switch -t origin/(브랜치명)`

S6-Lesson 1. git help

1. git help 사용법

A. Git 사용 중 모르는 부분이 있을 때 도움을 받을 수 있는 기능
`git help` : 기본적인 명령어들과 설명
`git help -a` : Git의 모든 명령어들

B. 명령어 설명과 옵션 보기

- `git (명령어) -h` : 해당 명령어의 설명과 옵션 보기
- `git help (명령어)` / `git (명령어) --help`
  - 해당 명령어의 설명과 옵션 웹사이트에서 보기
  - 웹에서 열리지 않을 시 끝에 `-w` 를 붙여 명시

Git 공식문서
자세한 사항은 아래의 Pro Git book에서도 확인 가능
Pro Git Book

S6-Lesson 2. git config 자세히 알아보기

1. global 설정과 local 설정

config를 **--global**과 함께 지정하면 전역으로 설정된다.

- 특정 프로젝트만의 `user.name`과 `user.email` 지정해보기

2. config 설정값 확인

현재 모든 설정값 보기
`git config (global) --list`

에디터에서 보기 *(기본: vi)*
`git config (global) -e`

기본 에디터 수정
`git config --global core.editor "code --wait"`

- 또는 `code` 자리에 원하는 편집 프로그램의 .exe파일 경로 연결
- `--wait` : 에디터에서 수정하는 동안 CLI를 정지

- `git commit` 등의 편집도 지정된 에디터에서 열게 됨

3. 에디터 설정을 되돌리려면

`git config --global -e` 로 편집기를 연 뒤 아래 부분을 삭제하고 저장

맥에서 `code` 로 VS Code가 실행되지 않을 시

- VS Code에서 `command` + `shift` + `p`
- `shell` 로 검색하여 `셸 명령: PATH에 code 명령 설치` 선택\
- 영문: `Shell Command: Install 'code' command in PATH`

4. 유용한 설정들

A. 줄바꿈 호환 문제 해결
`git config --global core.autocrlf (윈도우: true / 맥: input)`

B. `pull` 기본 전략 : `merge` 또는 `rebase` 로 설정

- `git config pull.rebase false`
- `git config pull.rebase true`

C. 기본 브랜치명
`git config --global init.defaultBranch main`

D. push시 로컬과 동일한 브랜치명으로
`git config --global push.default current`

5. 단축키 설정

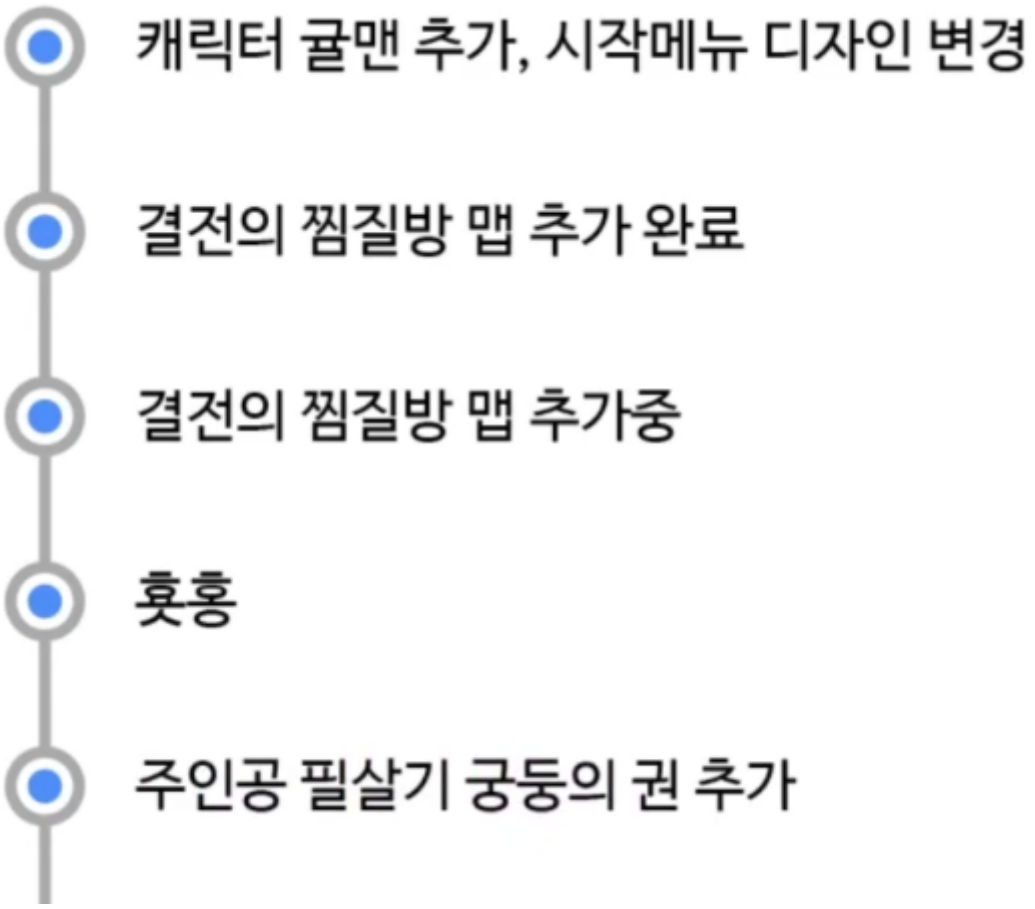
`git config --global alias.(단축키) "명령어"`
(예시: `git config --global alias.cam "commit -am"`)

S7-Lesson 1. 커밋하는 방법

1. 작업을 커밋할 때 권장사항

A. 하나의 커밋에는 한 단위의 작업을 넣도록 한다.

- 한 작업을 여러 버전에 걸쳐 커밋하지 않는다.
- 여러 작업을 한 버전에 커밋하지 않는다.



B. 커밋 메시지는 어떤 작업이 이뤄졌는지 알아볼 수 있도록 작성한다.

널리 사용되는 커밋 메시지 작성방식

```
type: subject
body (optional)
...
...
...
footer (optional)
```

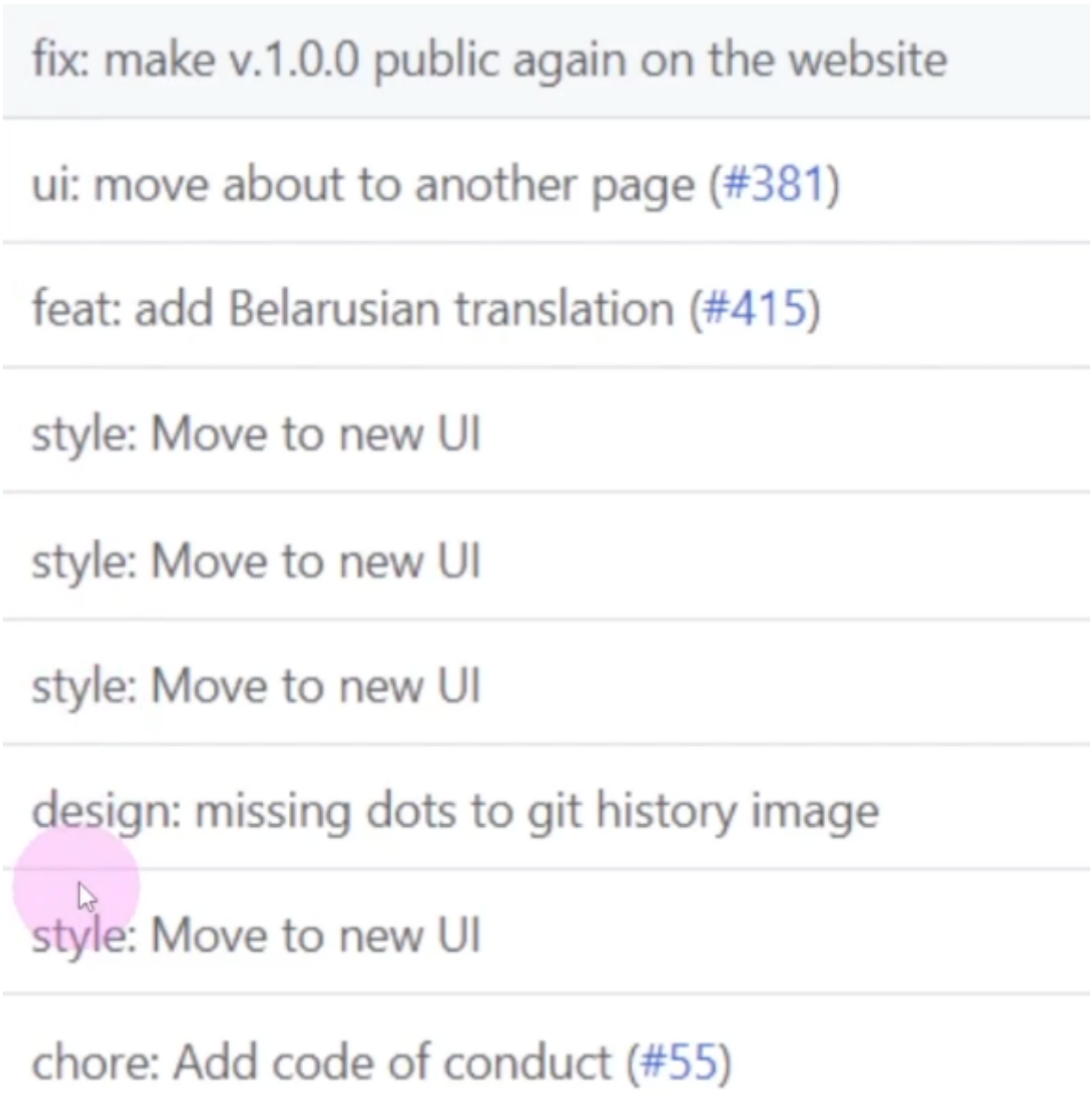
예시

```
feat: 압축파일 미리보기 기능 추가
사용자의 편의를 위해 압축을 풀기 전에
다음과 같이 압축파일 미리보기를 할 수 있도록 함
- 마우스 오른쪽 클릭
- 윈도우 탐색기 또는 맥 파인더의 미리보기 창
- Closes #125
```

타입	설명
feat	새로운 기능 추가
fix	버그 수정
docs	문서 수정
style	공백, 세미콜론 등 스타일 수정
refactor	코드 리팩토링
perf	성능 개선
test	테스트 추가
chore	빌드 과정 또는 보조 기능(문서 생성기능 등) 수정

코드 리팩토링: 사용하지 않는 코드 또는 중복된 코드를 지우고 코드의 로직을 깨끗하게 디자인 하는 것
chore : 자잘한 수정이나 빌드 업데이트, 파일 혹은 폴더명 수정 등

커밋 리스트 예



S7-Lesson 2. 세심하게 스테이징하고 커밋하기

1. 내용 확인하며 hunk별로 스테이징하기

Tigers 변경

- manager: `Thanos`
- coach: `Ronan`
- 새 members: `Gamora`, `Nebula`

Leopards 변경

- manager: `Peter`
- coach: `Rocket`
- 새 members: `Drax`, `Groot`

아래 명령어로 hunk별 스테이징 진행
`git add -p`

- 옵션 설명을 보려면 `?` 입력 후 엔터
- `y` 또는 `n`로 각 헝크 선택
- 일부만 스테이징하고 진행해보기
- `git stats`와 소스트리로 확인

2. 변경사항을 확인하고 커밋하기

`git commit -v`

- `j`, `k`로 스크롤하며 내용 확인
- `git diff --staged`와 비교
- 커밋 후 남은 헝크를 다른 버전으로 커밋해보기