

Markdown&Git 사용법 정리

Name: SYJung

※ Markdown 사용법 총정리

▶ 제목(Header)

▼ <h1>부터 <h6>까지 제목 표현 가능

제목 1

제목 2

제목 3

제목 4

제목 5

제목 6

▼ 제목1(h1)과 제목2(h2)는 다음과 같이도 표현 가능

제목 1

=====

제목 2

▶ 강조(Emphasis)

이텔릭체 (기울어진 글씨)

두껍게

이텔릭체+두껍게

~~취소선~~

▶ 목록(List)

1. 순서가 필요한 목록

2. 순서가 필요한 목록

- 순서가 필요하지 않은 목록(서브)

- 순서가 필요하지 않은 목록 (서서브)

3. 순서가 필요한 목록

1. 순서가 필요한 목록(서브)

4. 순서가 필요한 목록

- 순서가 필요하지 않은 목록에 사용 가능한 기호

- 대쉬(hyphen)

* 별표(asterisks)

+ 더하기(plus sign)

▶ 링크(Links)

[Google](<https://www.google.co.kr/>)

[Naver](<https://www.naver.com/> "링크 설명 작성")

라이선스 확인 [LICENSE](./LICENSE "라이선스")

유튜브로 가고 싶으면 [이것][Youtube]을 클릭하세요.

스팀으로 가고 싶으면 [요거][1]를 클릭해주세요.

문서 안에서 [참조링크]를 그대로 사용할 수 있다.

[Youtube]: <https://www.youtube.com/> “유튜브.”

[1]: <https://www.daum.net/> "다음“

[참조링크]: <https://docs.python.org/ko/3/library/index.html>

다음과 같이 문서 내 일반 URL이나 꺾쇠 괄호(< >)안의 URL은 자동으로 링크를 사용한다.

구글 홈페이지: <https://google.com>

네이버 홈페이지: <<https://naver.com>>

▶ 이미지(Images) - 링크와 비슷하지만 앞에 !가 붙는다

![대체 텍스트](이미지 주소 복사하고 붙여넣기)

![이미지 추가](이미지 주소 "링크 설명 텍스트도 넣을 수 있어요!")

▶ 코드(Code) 강조 - `을 입력하세요

이렇게 `강조`가 가능합니다.

▼ 블록(block)코드 강조 - `를 3번 이상 입력하고 코드 종류도 적습니다.

```
```python
```

```
a = 10
```

```
b = 20
```

```
print(a + b)
```

```
```
```

▼ 인라인 강조 (코드 강조와는 관계 x)

줄 4번 뛰어서 인라인으로 작성할 수 있습니다.

코드를 작성하려면 `를 3번 입력

▶ 표(Table)

```
첫번째 열	두번째 열	세번째 열	네번째 열
기본	좌측 정렬	가운데 정렬	우측 정렬
아무거나	아무거나	아무거나	아무거나
```

▶ 인용문(BlockQuote)

> 남의 말이나 글에서 직접 또는 간접으로 따온 문장.
> *(네이버 국어 사전)*

> 인용문을 작성하세요!
>> 중첩된 인용문을 만들 수 있다.
>>> 중중첩된 인용문 1
>>> 중중첩된 인용문 2

▶ 주석처리

```
<!-- 주석으로 처리가 가능합니다
안녕하세요 여러분 이것은 주석입니다.
-->
```

▶ 줄바꿈(Line Breaks)

```
동해물과 백두산이 마르고 닳도록
하느님이 보우하사 우리나라 만세 <!-- 띄어쓰기 2번-->
무궁화 삼천리 화려 강산
대한 사람 대한으로 길이 보전하세
```

▶ 목록[List]에다가 링크를 달수도 있다.

1. [자기소개](#Introduction)
 - [링크](#Link)
2. [코드](#Code)

※ Version Control System (VCS)

- 버전을 관리해주는 시스템

문제점 1: 파일명을 이용한 버전 관리

ex)

보고서.hwp,

보고서_버전2.hwp,

보고서_최종.hwp

보고서_최최종.hwp,

보고서_진짜_최종.hwp

등등 이런 식으로 버전 관리를 한다면 단점이 있다.

단점: 최종 버전이 무엇인지 모름 + 무엇이 수정되었는지 모름

→ 굳이 이렇게까지 관리하는 이유는 무엇일까?

- 기록하여 자신이 언제든 복구하고 싶어서이다.

해결책 1: Local Version Control Systems

- Local Computer에 DB가 존재함. DB에서 version 1 + version2 + version3까지 작업 완료함. 예전 version1의 파일을 가져오고 싶어서 Checkout 했다고 가정. 현재의 디렉토리에 파일 여러 개가 있는 상태에서 version1의 파일을 바꿔주고 version2도 바꿔주는 식으로 가능함.

장점 1: 파일명이 같다 + 자신이 어떤 버전이 제일 마지막 버전인지 확인 가능

(Database에서 checkout하면 그때그때마다 덮어쓰기 해줌)

문제점 2: 공동작업 할 때 불편함.

ex)

보고서.hwp

보고서_철수_그림수정.hwp

보고서_철수_그림수정_영희_표수정.hwp

보고서_제출본.hwp

보고서_제출본_철수_그림_다시수정.hwp

보고서_제출본_최종일까?.hwp

등등 이런 식으로 공동 작업하면 많은 문제 발생

해결책 2: Central Version Control Systems

- 회사에 Server를 하나 넣은 상태이고 컴 A와 컴 B가 있다고 가정. 컴 A가 버전1, 2, 3 중 버전3으로 checkout하고 수정함. 그러는 동안 컴 B가 버전 2를 checkout해서 수정함. 이런 방식으로 가능해짐.

문제점 2: 컴 A가 버전 3을 Checkout하는 동안 버전 3 문서의 소유권 전부 가져감. 그래서 컴 B는 할게 없어짐.

1. 한 사람이 Checkout하면 다른 사람이 쓸 수가 없음.
2. 만약 합친다면 컴파일 안되거나 오류 발생 가능성 있음.

해결책 3: Distributed Version Control Systems

- 서버 DB가 있음. 이 DB를 각 사용자 컴퓨터에 DB 복사 가능. 자기 컴퓨터에서 VCS있어서 각각 수정 가능. 그리고 서버 업로드하고 만약 업로드 겹치면 서로 만든 문서 적당히 합치고 컴파일 되도록 만듦. 이런 느낌임.
- Distributed Version Control Systems를 제일 처음 만든 것은 Git이다.

장점 3: 모두가 행복하게 공동작업(협업) 가능 + 모든 Operation이 Local에게 작동한다.

핵심:

1. 마치 Version Control Systems를 Git으로 생각하면 됨.
2. 마치 데이터베이스(DB)를 Git Repository라고 생각하면 됨.
3. 기존의 Central Version Control Systems는 모든 작업은 전부 온라인에서 이루어졌음. 하지만 요즘은 모든 사람의 컴퓨터에 분산되어 저장되어있는 Distributed Version Control Systems를 쓴다. 그 중 대표적인 것이 바로 Git이다.

◆ 여기 3가지 작업이 Git에서 편하게 해준다.

▼ Add Changes Sequentially

1. 무언가 변화를 계속 순차적으로 이어붙인다.
2. 파일이 다양해도 상관없다.
(파일 지우기/추가, 폴더 지우기/추가, 파일의 내용을 삭제/추가 가능)
3. 원하는 버전으로 시간여행 가능

▼ Save Different Versions

1. 분기가 가능하다. (Branch)

▼ Merge Different Versions

1. 분기하고 합치기도 가능하다. (Merge)

- 예전 구현 방식 -

▶ Delta-based Version Control (CVS, Subversion and etc로 동작)

- 이전 버전에서 변경된 것만 저장. (변화량만 저장)

장점: 변경된 것만 저장하므로 저장용량(space)이 줄어든다.

단점: history가 길수록 롤백하는데 시간이 오래걸림.

- Git의 동작 방식-

▶ Snapshot-based Version Control (Snapshot-based로 동작)

- 파일 A를 수정하다 가정하면 이 수정한 파일을 그대로 가지고 있다.

장점: 시간여행이 빠르다.

단점: 각각의 버전이 각 파일을 가지고 있으니 버전의 크기가 크다. 하지만 텍스트는 용량이 작기 때문에 굳이 예전 방식인 Delta-based Version Control을 사용할 필요는 없다. 그래서 Git은 이런 식으로 동작한다.

▶ Git은 리누스 토발즈(Linus Torvalds)가 만든 Version Control System이다.

1. Git Bash하면 터미널 창이 열리고 이 터미널에서 CLI(커맨드 라인 인터페이스)로 명령어를 입력해서 Git 사용한다.

2. GUI 이용해서도 쓸 수 있다. CLI가 너무 어려워서 GUI(그래픽 유저 인터페이스)를 제공해주는 tool을 이용해서 사용한다.

▶ GitHub

- Git을 사용할 때 Repository가 있고 그 저장소를 서버에 저장한다고 설명했다. 즉, 서버라고 생각하면 된다. 서버에 Git Repository를 만들고 그것을 각각 사용자의 PC에 Clone을 하여 Local 작업을 한다.

정리: GitHub은 Git을 사용하는 저장소가 모여있는 허브라고 생각하면 된다.

※ Git 명령어

▶ Configuration

`git config --global --list`

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

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

▶ Create a Local Repository

`git init`

▶ add

`git add 파일명`

`git add -man` (add의 manual 보여줌)

`git add -i` (interactive picking 기능이다.)

`git add -u` (업데이트한 tracked 파일을 add)

`git add -all` (모든 tracked와 untracked 파일을 add)

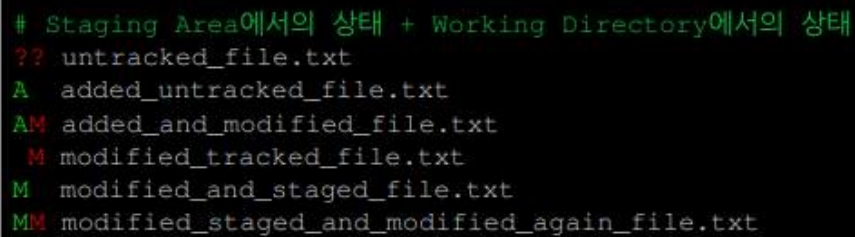
`git add .` (news, modifications without deletions 전부 add)

▶ status

`git status`

`git --short`

`git status -s`



```
# Staging Area에서의 상태 + Working Directory에서의 상태
?? untracked_file.txt
A  added_untracked_file.txt
AM added_and_modified_file.txt
M  modified_tracked_file.txt
M  modified_and_staged_file.txt
MM modified_staged_and_modified_again_file.txt
```

- ??: untracked
- A: added
- M: modified
- D: deleted

▶ commit

git commit

git commit -m "커밋명"

Working Directory: 작업공간

Staging Area: 준비공간

Local Repository: 저장소

Working Directory (add)→ Staging Area (commit)→ Local Repository

▶ log

git log

git log -n 2 (최근 log 2개 보여줌)

git log --oneline

git log --all --decorate --oneline --graph (--decorate는 빼도 됨)

(난 보통 git log --oneline --all --graph로 씬ㅎㅎ)

▶ checkout

git checkout 해시

git checkout 브랜치

git checkout - (이전 commit으로 체크아웃)

▶ diff

git diff 해시1 해시2 (수정 또는 변경한 것을 비교해서 보여준다.)

▶ .gitignore (필터링한다고 보면 됨.)

ex) vim .gitignore 해서 파일 만들어서 설정하면 됨

---vim .gitignore하고 작성하라.---

*.log

!test.log (모든 .log는 무시하되 test.log는 무시하지마라)

folder (folder라는 파일이나 디렉토리든 무시해라)

delete/ (delete라는 디렉토리나 그 하위 녀석들은 무시해라)

저기 작성된 것은 다 무시함 (stage에 안 올라가고 무시함. 물론 !test.log는 예외임.)

*.log는 .log로 되는 파일들은 다 무시하라는 의미이지만

!을 앞에 붙이면 "단, ~~은 허용" 약간 이런 의미가 됨

▶ **commit -amend** (이전 commit에 새로운 commit을 덮어쓰는 개념)(해시 바뀜 o)

git commit --amend -m "변경할 메시지"

(메시지 마음에 안들 경우 + 메시지 수정한 상태에서 파일 내용도 수정하고 싶을 경우)

git commit --amend --no-edit

→ commit 메시지는 건드리지 않고 수정한 것을 덮어쓰겠다.

▶ **rebase -i** (만약 중간에 있는 commit의 메시지를 수정 + commit을 압축하고 싶을 때)

git rebase -i 해시 (입력한 해시와 이전 해시는 포함하지 않고 이후 commit을 설정한다.)

git rebase -i -root (처음 commit부터 끝까지 설정 - 전부 가능)

reword: commit 메시지 수정

squash: commit을 합체

▶ **reset**

(현재 commit들이 마음에 안든다 → 이전의 commit로 돌리고 싶을 때 reset을 사용)

▼ reset은 3가지 옵션이 있다.

--hard: repository, staging area, working directory까지 전부 바뀜 (전부 없애줌)

--mixed: working directory만 놔두고 나머지는 바꿈

<working directory(변경된 것)은 살아있는데 staging area는 없어진다는 얘기>

--soft: working directory(변경된 것), staging area 이렇게 둘은 살려줌.

ex) 해시1~해시5까지 있다고 가정

git reset --hard 해시3 (해시1, 2, 3은 남고 Head와 Master은 해시3으로 이동)

- 물론 해시 4, 5는 완전히 삭제된 것이 아님. (단지 master도 이동해서 어려울 뿐)

♠ **HEAD~n (본래 해시를 쓰지만 상대적으로 쓰고 싶을 경우)**

git reset --soft HEAD~1 (현재에서 하나 전 해시)

git rebase -i HEAD~2 (현재에서 2개 전 해시가 해당)

▼ **checkout과 reset의 공통점과 차이점**

공통점: Head가 움직임

차이점: checkout은 Head는 움직이지만 Master은 그대로 남아 있음. 그러나 reset은 Head와 Master 모두 이동함.

(commit이 삭제되는 것이 아님. 그런데 master가 이동해서 다시 돌아가기가 힘들)

결론: reset을 사용할 때는 조심히 사용하라.

♠ checkout - 파일명

git checkout -- 파일명

<staging area에 올라오기 전의 수정된 파일(modified 파일)을 없앴. (잘 안 쓴다.)>

▶ branch

git branch (현재 우리의 branch를 보여줌)

git branch 브랜치명 (branch 생성)

git branch -d feature (feature라는 branch를 지움)

git checkout -b feature (feature라는 branch를 만들고 이동)

▶ merge

ex) 만약 feature를 master branch로 merge하고 싶으면 master branch로 이동해서 merge한다.

git merge feature

▼ Fast-forward (앞으로 빨리 감기)

- conflict이 안날 경우 commit을 안 만들고 그대로 사용하면서 포인터만 이동한다.

이것을 Fast-forward라고 함

▼ merge하다가 conflict이 난 경우

- **git merge tool** 입력하여 meld 실행하여 해결한다.

- vim 또는 visual studio에서 고쳐도 상관없다.

▼ Conflict을 해결할 경우

git merge --continue 입력하고 진행하면 된다.

▶ push (로컬에서 작업한 것을 서버로 정보 줄 때)

git push origin master

git push

만약 로컬에서 처음 작업하고 push 할 경우

git push -u origin master (꼭 -u옵션을 붙여야 함.)

만약 master branch 말고 feature branch를 push 하는 법

git push -u origin feature

▶ clone (서버에서 로컬로 복사할 때)

git clone https://github.com/yeobdoll/Rebase_Test.git (예시임)

▶ **fetch** (Github로부터 가져와)

- 항상 master가 최신버전인지 체크해야함.

git fetch

▶ **pull** (서버에서 작업한 것을 로컬로 줄 때)

♠ pull = fetch + merge (Github로부터 가져와서 합치라는 의미)

git pull (git fetch안하고 pull해도 됨)

Tip. 다른 branch로 checkout할 때 타이핑하다 tab 누르면 자동완성됨

▶ **rebase**

ex) 만약 feature branch를 rebase하고 싶으면 feature branch에서 진행하면 됨.

git rebase master (master로 rebase 해줘!)

Tip.

git rebase --continue (계속)

git rebase --abort (취소)

--continue와 --abort 옵션은 merge, rebase 등 상황에 맞게 활용.

▶ **tag**

- 포인터는 3가지 종류가 있다. (branch, Head, tag)

git tag (현재 저장된 tag들을 보여준다.)

git tag 태그명

git tag 태그명 해시

git tag -a 태그명 -m "태크 설명" (annotated tag 하는 법-부수적인 정보 입력)

git tag -d 태그명 (태그 지우기)

tag에서 branch를 따서 또 작업 가능.

git checkout -b branch명 태그명

▼ **tag push 하는 법**

git push origin -tags (tag들 전부가 push 됨)

git push origin 태그명 (명시적으로 push)

▶ stash

만약 내가 수정하고 있는 상태인데 누군가가 다른 곳을 보고 싶다 하는 상황일 경우 모든 작업이 stash라는 공간으로 잠깐 이동시키면 된다.

```
git stash      (모든 작업이 stashed area로 잠깐 이동)
```

```
git stash pop  (작업 다시 불러와 적용하기)
```

- stash는 pop 구조이다.

▶ blame

- 어떤 파일을 누가 언제 어디를 고쳤는지 보고 싶을 때

```
git blame 파일명
```

▶ alias (잘 안씀)

- 긴 명령어를 짧게 고칠 수 있게 해줌

```
git config --global alias.co checkout (git co → checkout 기능을 한다.)
```

```
git config --global alias.st 'status -s'
```

```
git config --global alias.tree 'log --all --decorate --oneline --graph'
```

```
git config --global --unset alias.st (alias 지울 때)
```

▶ reflog

- 레퍼런스의 로그 (우리의 모든 commit을 뜨게 해준다.)

```
git reflog
```

▶ editor

```
git config --global core.editor "실행파일의 위치"
```

▶ mergetool

설정 방법

```
sudo apt install meld (또는 사이트에서 다운으로 알고 있음)
```

```
git config --global merge.tool meld
```

실행

```
git mergetool
```

♠ merge 안한 상태인 branch지을때는 -D 옵션 사용하면 된다.

git branch -D feature

※ 협업

- 항상 협업할 때는 master branch는 건들지 않는다.
- 협업할 때는 로컬에서 master로 merge하면 안된다.

-----J User -----

1. 나의 branch를 push 했다고 가정. (local → server)
2. GitHub 사이트에 협업하는 Repository를 확인해보면 "Compare & pull request"라는 버튼이 생긴다. 혹은 Pull requests로 가면 new pull request를 눌러도 됨.
(master branch에 merge하고 싶으면 이걸 누르면 됨. Github에서 다른 사람과 보면서 merge하는 것이다.)
3. 버튼을 눌렀으면 pull request를 만드는 창이 나온다. 아래 보면 Able to merge가 뜨는지 확인해라. (merge가 conflict 안 나고 잘 되면 됨)
4. 내가 한 일을 쓴다. 그리고 아래에 세부 변경사항을 쓰면 됨.
ex) 팀 페이지 제목 변경
5. 이제 변경했으니깐 다른 사람에게 리뷰 받고 싶으면 오른쪽에 Reviewers 클릭해서 내 팀원 선택하면 됨.(팀 페이지가서 자기 팀 멤버 확인 + Assignees은 나 자신 선택하면 됨.)
(담당자라는 의미)
6. 이제 Create pull request를 클릭해주면 됨.
이러면 이제 어떤 뜻이냐면 "내가 이렇게 변경을 해서 master branch에 merge하고 싶어요. pull request를 통해 리뷰 부탁드립니다. 라는 의미를 가짐.
7. 다른 리뷰어들에게 메일이 간다.

-----리뷰어입장 -----

1. 메일을 받으면 Pull requests로 감. 여기서 Commits 또는 Files changed를 살펴보면 됨.
2. Files changed에서 잘못된 것이 있으면 잘못된 곳에 마우스 커서 갖다 놓으면 +버튼이 생긴다. 이거 누른다.
3. 그러면 comment를 남기는 창이 뜬다. 여기다가 공손하게 다시 수정해달라고 쓰면 된다.
- 여기서는 Start a review보다는 Add single comment를 다는 것이 더 편하다.
4. 그다음 오른쪽 위 Review changes 버튼을 누른다.

ex) Write칸에 “팀 번호가 틀렸어요.”

이런 식으로 쓰고 Request changes 체크하고 Submit review 클릭하면 됨.

5. 그럼 Pull requests 창에 추가됨. 그러면 거기에 아마 상대방이 답변을 해줄거다.

-----리뷰를 통해 J User가 다시 수정함 -----

1. Files changed에 가보면 수정을 다시 해준 것을 확인한다. 그리고 오른쪽 approved these changes 버튼 누른다.

ex) 좋아요 등을 써주고 올리면 됨.

2. 그리고 리뷰가 완전히 완료되면 Merge pull request를 클릭하면 된다. 그러면 Pull request successfully merged and closed가 나오면서 merge가 완료되면서 pull requests 가 없어진다.

3. Github에 보면 master로 merge가 완료된 것을 볼 수 있다.