

Kang's Git Cheat Sheet

2021.08.31

- `git init`
 - `git config --global user.name your_username`
 - `git config --global user.email your_email@example.com`
 - `git config --global core.editor`
 - `git config --global init.defaultBranch main` # Branch명 변경 (main)
 - `git branch -M main` (master 확인후 조치)
 - `git config --global core.autocrlf true`
(개행문자 처리 for windows, for Unix or Mac OS : `git config --global core.autocrlf input`)
-
- `git status` # 현재 상태 확인.
 - `git add p1.txt` # staging area 에 등록
 - `git add -p <file>`, `git add --patch`
Add some changes in <file> to the next commit, Patch단위로 검토할 수 있음 참고: <https://blog.outsider.ne.kr/1247>, S(split)선택후 , 원하는 hunk만 선택(y)하고 완료 후에 commit 진행. 다시 `add -p`를 통하여 똑 같은 과정을 진행하여 수정사항을 분리하여 commit 완료
 - `git status` # 녹색으로 된 new file 내용 확인
 - `git commit` # 저장소에 commit 객체 생성
→ commit 객체를 나타내는 Message(기록내용)을 삽입하라고 메모창이 뜸 (메모창(notepad)은 `git config core.editor` 로 기본 editor로 설정해 놓은 상태임)
 - `git commit -m 'second commit'`
 - `git commit -am 'work allocation commit'`
→ '-am' 은 all(changed file)과 message을 같이 하겠다는 의미
 - `git commit --amend -m "message"` # Change the last commit
 - `git commit -m 'message title' -m 'description'` # 첫번째 -m은 title 두번째 -m은 설명 기입
-
- `git log`
 - `git log --oneline`
 - `git log -1`
 - `git log -p <file>` # Show changes over time for a specific file
 - `git log --branches --oneline` # 전체 브랜치(main과 feature-1) 확인
→ --branches 옵션이 없으면 해당 branch log의 직계존속(history)만 보여줌
 - `git log --branches --oneline --graph`
로그에 모든 브랜치를 표시하고, 그래프로 표현하고, 브랜치 명을 표시하고, 한 줄로 표시할 때
 - `git log main..feature-1` # main에는 없고 feature-1에만 있는 내용

<ul style="list-style-type: none"> ○ <code>git log feature-1..main</code> # feature-1에는 없고 main에만 있는 내용 ○ <code>git diff main..feature-1</code> # 브랜치 간 내용 비교
<ul style="list-style-type: none"> ○ <code>git branch</code> # branch와 현재 위치 확인 ○ <code>git branch -v</code> ○ <code>git branch -av</code> # List all existing branches ○ <code>git branch -M main</code> # 브랜치가 master일 경우 수정 ○ <code>git branch feature-1</code> # “feature-1 “ 브랜치 생성 ○ <code>git branch -v</code> # 2개의 브랜치 중에서 main에 HEAD가 있음 (별표, 녹색) ○ <code>git checkout feature-1</code> # “feature-1 “ 브랜치로 전환, switch로 변경중 ○ <code>git switch feature-1</code> ○ <code>git checkout -</code> # 이전 branch로 switch ○ <code>git checkout main</code> # main으로 브랜치를 전환 ○ <code>git checkout -b feature-2</code> # feature-2 브랜치를 만들고 이동함. <ul style="list-style-type: none"> ※ <code>git checkout -b br2 = git branch br2</code> 를 실행하고 <code>git checkout br2</code> 실행한 결과 ○ <code>git switch -c feature-2</code> # 위 명령과 동일 ○ <code>git branch -d feature-2</code> (<code>-D</code> : merge가 되지 않았어도 삭제) # 브랜치 삭제 : 다른 브랜치로 이동하여 브랜치 삭제 (on branch br2에서 br2 branch 삭제 불가) ○ <code>git push -d origin develop</code> # 원격 브랜치 branch 삭제 ○ <code>git branch -u <origin/develop></code> # 현재 브랜치를 origin의 develop 브랜치에 track 시키고자 할 때 ○ <code>git checkout --track <remote/branch></code> # Create a new tracking branch based on a remote branch, Remote 브랜치를 pull하면서 track을 동시에 수행
<ul style="list-style-type: none"> ○ <code>git merge feature-1</code> # 현재 branch로 feature-1의 내용을 가져옴 ○ <code>git merge --abort</code> <ul style="list-style-type: none"> ※ merge 중에 conflict가 나서 merge 이전으로 돌리고 싶을 경우 merge 이전 상태로 돌아 감.
<ul style="list-style-type: none"> ○ <code>git log -p</code> # commit간의 차이가 위 아래로 전부 표시되어 보임 <ul style="list-style-type: none"> ※ 파일로 결과를 저장할 때 : <code>git log -p >> 파일명</code> ○ <code>git diff <commit ID1>..<commitID2></code> ← 주의 # ‘..’ 앞뒤로 space가 없어야 함, commit 간의 차이점 확인. * commit ID = commit 해시 ○ <code>git diff</code> # <code>git add</code> 전후 파일 비교 : add한 후에는 의미 없음 즉, add 하기 전의 수정파일과 add 혹은 마지막 commit한 파일 간의 비교 ○ <code>git show {commit ID}</code>
<ul style="list-style-type: none"> ○ <code>git reset --hard <commit ID></code> # 예전commit로 돌아가기 (기존 삭제) ○ <code>git reset --soft <commit ID></code> # commit 되돌리기 (기존 복원) ○ <code>git reflog show -3</code> # 돌아갈 commit 확인하기

<ul style="list-style-type: none"> ○ git checkout <commit ID1> <ul style="list-style-type: none"> ※ detached HEAD 발생. 살려서 계속 작업을 해야 한다면 신규 브랜치 생성 → git checkout -b '신규브랜치명' <commit ID1>
<ul style="list-style-type: none"> ○ git tag tag명 (예: git tag v1.0) # Lightweight 태그 : tag명만 부여 ○ git tag -a tag명 -m "message " (예: git tag -a v1.0 -m "version 1.0) # Annotated 태그 : 태그명명자, 이메일, 태그생성 날짜, 태그 메시지까지 저장 ○ git tag # 현재 tag 상태 확인 ○ git show tag명 # 명명자, 이메일, 날짜 등 표시 ○ git tag -a v3.0 -m "Version 3.0 " # 현재 위치에서 tag 설정 (annotated) ○ git checkout 57560ec ○ git tag v2.0 # commit 위치로 이동후 설정 (lightweight) ○ git tag -a v1.0 fd13417 -m "first version 1.0 " # commitID로 설정 ○ git push origin tag명 (예: git push origin v1.0) #한 개의 tag push ○ git push origin --tags # 전체 tag 일괄적으로 push하기 ○ git tag -d v1.0 # 로컬 tag 삭제 : git tag -d tag명 ○ git push origin : v1.0 # 원격 tag 삭제 : git push origin : tag명
<ul style="list-style-type: none"> ○ git stash # 워킹 디렉토리에서 commit하지 않은 수정한 파일만 저장 ○ git stash apply # 브랜치로 돌아온 후에 하던 작업을 불러 올 ○ git stash list # 저장된 stash 목록 보기 ○ git stash drop # stash 목록 지우기 ○ git stash pop # stash apply 와 stash drop을 동시에 수행하는 명령어
<ul style="list-style-type: none"> ○ git checkout main ○ git merge my-branch # merge ○ git merge --squash my-branch # Squash & Merge ○ git rebase master # Rebase & Merge ○ git rebase --abort # Abort a rebase ○ git rebase --continue # Continue a rebase after resolving conflicts
<ul style="list-style-type: none"> ○ git remote add origin https://.... (github 에서 복사한 저장소 주소) - 최초 연결시 한 번만 수행 ○ git remote -v # List all currently configured remotes ○ git remote show <remote> # Show information about a remote ○ git push -u origin main (혹은 git push --set-upstream origin main) - 이후에는 git push만 수행 - 뜻 : 로컬에 있는 main을 원격저장소인 origin에 있는 main으로 보내고 일치시킨다. ○ git push -u origin main : master → (로컬의 main을 origin의 master와 일치) ○ git push -u origin feature-1 # 이후는 push만 진행 (feature-1 브랜치에서) ○ git push -all { -u origin }

○ git push { origin main }
○ git pull origin main # remote(origin) main branch 를 내려 받기 ○ git pull {origin main } ※ push, pull은 branch(commit object) 단위로 움직임
○ git clone “github에서 복사한 저장소 주소”
○ git checkout main ○ git remote add upstream “fork site 주소 “ ○ git pull upstream main # upstream main의 최신 내용을 나의 로컬 main으로 복사
○ git blame <file> # Who changed what and when in <file>
○ git rm -r --cached . # Stage되어 있는 파일을 unstage시킬 때 ○ git restore <파일명> # Not Stage되어 있는 파일을 내용 변경 전으로 돌릴 때, Git status에서 적색으로 있다가 명령 후에는 working tree clean으로 표시됨