Using git

git 자주 사용하는 명령어 모음	
깃 초기화	git init 경로명
깃 상태 확인	git status
깃 저장소 복제	git clone 원격저장소URL 새폴더이름
파일 등록과 커밋	git add 파일이름 → git commit → 에디터에서 커밋 메시지 작성
	git commit -a → 에디터에서 커밋 메시지 작성
	git commit -am "커밋 메시지"
로그 확인	git log
커밋 비교	git diff
원격 저장소 별칭 확인	git remote
원격 저장소 별칭과 URL 확인	git remote -v
원격 저장소와 연결	git remote add 원격저장소별칭 원격저장소URL
원격 서버 삭제	git remote rm 원격저장소별칭
커밋 가져오기	git pull 또는 git fetch
커밋 전송하기	git push 원격저장소별칭 브랜치이름
현재 브랜치 확인	git branch
브랜치 생성	git branch 브랜치이름
브랜치 이동	git checkout 브랜치이름
스태시 저장	git stash
스태시 읽기	git stash pop
브랜치 병합	git merge 브랜치이름
리베이스 병합	git rebase 브랜치이름
리셋	git reset 옵션 커밋ID
리버트 취소 커밋	git revert 커밋위치
태그 관리	git tag
태그 전송	git push 원격저장소별칭 태그이름
서브모듈로 연결	git submodule add 원격저장소URL 폴더이름

why git?

- 분산버전관리 (Distributed Version Control System) → 이것이 주는 장점? → 기대효과
- 사용자의 요구사항
  - 추가할 기능 많음; 각 기능별로 작업 이력을 git commit에 저장하므로 작업 이력을 관리 할 수 있음
  - 항상 모든 이슈사항은 응급!; 버그나 수정사항들이 발생하여 프로그램을 수정해야 할 경우, 요청자는 항상 빠르게 대처 가능.
- 개발자의 요구사항
  - 개발과 실제 운용 소스가 다름; 현재 작업 중인 내용이 있다면 현재 작업 내용을 백업하고, 운영(서비스)되는 배포판의 소스를 가져와서 수정한 후 배포하고 수정된 내용을 작업 중이던 소스에도 적용해야 하는 경우에 git은 유연하게 대처 가능.
  - 개발 사항에 대한 이력이 필요; 각 작업별로 언제, 어떤 내용을, 누가 했는지 이력을 관리하면 프로젝트 관리에 이점.

pros. and cons.

pros.

- 강력한 브랜치 기능; 브랜치를 쉽게 만들고 병합 가능, 일시적인 작업에 대한 이력 관리 쉬움
- 속도와 성능; 처리 속도가 빠르고 기능이 많음
- 분산작업; 장소에 구애 받지 않고 협업이 가능(메인 서버에 연결없이 작업 가능), 저장소의 완전한 복사본으로 작업
- 데이터 보장성; 모든 파일을 암호화하여 저장
- 무료; 비용 안 듬.

cons.

- 배우기 어려움.
- 어떻게 사용하는 가가 중요하므로 잘 정의된 개발 프로세스가 필요.
- git hub와 같은 서비스를 이용할 경우 비용 발생. (bitbucket은 5명까지 무료 사용 가능)

setting up a git

- 설치 osx, windows, linux
- global settings

git 시작하기

- 생성 init, clone
- git이 파일(디렉토리)를 관리하는 방법
  - working directory → stage area → git repository
  - working directory; 사용자가 작업하는 디렉토리
  - stage area; git repository에 저장될 파일 목록
  - git repository; git이 파일을 저장하는 저장소
- git repository의 파일 관리 add, remove, commit
- 상태 보기 status
- 뒤로 되돌아 가기 reset
- 버전 되돌아보기 log
- 태그 붙이기 tag
- 이건 저장하지 않을 거야 .gitignore
  - repository에 넣으면 좋은 것들; 스펙문서, 설계서, 매뉴얼, 도움말, 소스코드, 이미지, 외부라이브러리, 빌드스크립트
  - 넣지 말아야 할 것들; 바이너리, 설치본, 영업자료, 마케팅자료, 판매자료 등.
- stash
- HEAD; 최종 커밋 작업의 위치 포인터, HEAD와 스테이지 영역의 내용을 합쳐 다음 커밋을 생성
- ^,~; HEAD^^, HEAD~~, HEAD^3, HEAD~3
- AHEAD; 서버로 전송되지 않은 로컬 커밋
- BHEAD; 로컬로 내려 받지 않은 커밋

- pull → 작업 → commit → pull → push
- 대부분의 git 명령어는 working tree clean 상태에서 작업
- reset; 저장소를 외부에 공개했거나 공유하고 있다면 주의해서 리셋 사용
- revert; 기존 커밋을 남겨 두고 취소에 대한 새로운 커밋 생성.
- reset vs. revert; 커밋 정보의 삭제 여부. 리셋: 커밋 삭제, 리버트: 취소 커밋 생성
- versioning; major.minor.patch, SemVer(Semantic Versioning)
    - RC(Release Candidate), GA(General Availability), M(Milestone)
- tag; 특정 커밋에 태그 부착
    - Annotated; 태그 이름 + 정보 포함
    - Lightweight; 태그 이름만 포함
    - ./git/ref/tags 폴더

```
$ git config --global core.editor "에디터경로"
$ git config --global credential.helper cache # 인증정보 캐시

$ git init
$ git clone 원격저장소URL 새폴더이름

$ git add 파일이름

$ git rm --cached 파일이름 # 스테이지에서 파일이름 삭제, 파일 등록 후 커밋하지 않음
$ git reset HEAD 파일이름 # 한 번이라도 커밋을 했을 경우

$ git status

$ git mv 파일이름 새파일이름 # 파일 이름 변경, mv -> git rm -> git add 와 같음

$ git commit
$ git commit -a # 커밋 하기 전 자동으로 모든 파일을 등록 후 커밋

$ git log
$ git log --pretty=short
$ git show 커밋ID # 특정 커밋의 상세 정보
$ git diff # 워킹 디렉토리와 스테이지 영역 간 변경 사항
$ git diff HEAD # 최신 커밋과 워킹 디렉토리 간 변경 사항

$ git checkout -- 수정파일이름 # 수정된 파일을 커밋 상태로 되돌림
$ git add 수정파일이름

$ git commit -m "커밋메시지"
$ git commit -am "커밋메시지"
$ git commit --allow-empty-message -m "" # 빈 커밋 작성
$ git commit --amend # 방금 전에 작성한 커밋 메시지 수정
$ git commit -v # 커밋 메시지에 diff 내용 추가

$ git stash # 현재 작업들을 임시 스택에 저장, 워킹디렉토리+스테이지영역 파일까지 모두 저장
$ git stash save
$ git stash save "WIP: message"
```

```
# 작업 중인 내용을 강제 커밋(비추천)
$ git commit -am "temp" # 임시 커밋
$ ... # 다른 브랜치 작업 후 현재 브랜치로
$ git reset -soft HEAD^ # 리셋 복워
###

$ git stash list # 스태시 리스트
$ git stash show # 스태시와 워킹디릭토리 차이
$ git stash show -p stash{0} # 상세

$ git stash pop # 스태시 스택에서 pop, 자동 삭제, 충동시에는 삭제 안함
$ git stash apply --index # 스테이지 등록된 상태까지 복구

$ git stash branch 브랜치이름 # 스태시를 브랜치에 적용

$ git stash apply # 스태시 스택에서 읽어오지만 삭제하지 않음
$ git stash apply stash@{0} # 0번 스태시로 복원

$ git stash drop  # pop 충돌시, apply 했을 시 스태시 삭제

$ git clean # 워킹디렉토리 안에 추적되지 않는 파일을 모두 삭제, -f 강제, -n 가상으로 처리하고 사용자 확인, -d untracked 상태의 파일만 삭제, -x .gitignore 설정된 파일까지 삭제

$ git log --oneline # 로그 한줄씩

$ git reset --hard HEAD^^^  # HEAD로부터 세번째로
$ git rest 옵션 커밋ID # 옵션: soft; 스테이지 영역을 포함한 상태로 복원, mixed; 기본값, hard: 실제 파일이 삭제된 이전 상태로 복원

$ git reset --soft HEAD~ # 이전 커밋으로 soft 옵션을 사용한 리셋. 파일을 수정하고 add 명령어로 스테이지 영역에 올려 커밋을 실행하기 직전의 단계로 되돌림
$ git commit --amend # 위와 유사함

$ git reset --mixed 커밋ID
$ git reset 커밋ID # 위와 같은 명령
$ git reset --mixed HEAD~ # 리셋한 후 스티이지 상태까지 복원하지 않음, 커밋하려면 add 명령어를 먼저 실행

$ git reset --hard HEAD~ # 사용한 커밋 이후의 모든 내용 삭제, 워킹 디렉토리 내용도 함께 삭제

# 스테이지 리셋
$ git add 파일이름
$ git reset 파일이름
$ git reset --mixed HEAD 파일이름 # 파일 이름을 지정하여 리셋하면 해당 파일은 unstage 상태
$ git reset 커밋ID 파일이름

# 작업취소
$ git reset --hard HEAD
```

```
# 병합 취소
$ git reset --merge HEAD~


$ git revert HEAD # 현재 커밋을 리버트, 새로운 리버트 커밋 생성

$ git revert 커밋ID
$ git revert 커밋ID .. 커밋ID # 범위 연산자 ..를 사용, 여러 커밋 리버트

$ git revert --mainline 숫자 병함커밋ID # 리버트로 병합 취소

$ git tag

$ git tag -l # 또는 --list

$ git tag -a 버전  # annotated 태그 생성, 에디터로 작성

$ git log --decorate

$ git tag -a 버전 -m "메시지" # 에디터 안뜸

$ git tag -d 태그이름

$ git show 태그이름 # 태그의 상제 정보 확인

$ git tag 태그이름  # lightweight, 커밋의 체크섬만 가지고 있다

$ git tag -a 태그버전 커밋ID # 특정 커밋ID에 태그 버전 생성

$ git checkout 태그버전 # 태그버전으로 체크아웃, 이 경우 추가 커밋 작성 불가
$ git checkout -b 브랜치이름 태그이름 # 태그 기준으로 브랜치 생성

$ git push 태그이름 # 원격저장소에 태그이름 push
$ git push 원격저장소이름 --tags # 모든 태그를 한꺼번에 원격저장소에 push
$ git push --delete 원격저장소이름 태그이름 # 원격저장소의 태그 삭제
$ git push 원격저장소이름 로컬태그이름:원격저장소태그이름 # 원격저장소에 로컬과 다른 이름으로 태그 전송
```

git branch

- 기준이 되는 메인 프로젝트와 개발 프로젝트 혹은 디버그 프로젝트, 핫픽스 프로젝트 master, branch
    - 가지 치기, 가지 잘라내기
- 지금 작업하고 있는 위치 head
    - 개발 브랜치에서 핫픽스 브랜치로 전환 checkout
- 메인+브랜치⇒새로운 메인, 브랜치1+브랜치2⇒새 버전의 브랜치1 merge

    - 병합(merge)하기 전에 diff
    - rebase
- Fast-Forward 병합; 순차적 커밋에 맞추어 병합 처리
- 3-Way 병합; 공통 조상 커밋 + 공통조상커밋을 포함하는 브랜치 + 새로운 브랜치 ⇒ 하나로 병합, 병합을 성공적으로 완료 후 새로운 커밋을 추가로 생성 (=병합 커밋)
- Rebase; 두 브랜치를 서로 비교하지 않고 순차적으로 커밋 병합. 병합 커밋 없음, 브랜치의 마지막을 가리키는 커밋 위치 다름. 원격 저장소에 푸시하면 리베이스 사용하지 않는 것이 원칙.

```
$ git branch # 브랜치 목록
$ git branch 브랜치이름 커밋ID # 커밋ID를 기점에서 브랜치이름으로 브랜치 생성, 커밋ID 없으면 HEAD 포인터 기준

$ git rev-parse 브랜치이름 # 커밋 해시값(SHA1) 확인
$ git branch -v  # verbose, 브랜치 세부 사항

$ git checkout 브랜치이름 # 브랜치이름으로 이동, working tree clean 이어야 함, HEAD 포인터도 이동
$ git checkout -- 파일이름 # 파일로 체크아웃
$ git checkout - # 이전 브랜치로 이동
$ git checkout -b 브랜치이름 # 브랜치이름 생성하고 그 브랜치로 이동

$ git checkout 커밋해시키 # 해시키로 체크아웃
$ git checkout HEAD~1 # 한 단계 전
$ git checkout - # 현재로 이동

$ git log --graph --all # 텍스트 그래프 출력
$ git show-branch --more=10 # 출력 커밋 개수 제한

$ git merge 브랜치이름 # 현재 브랜치 + 브랜치 병합
$ git merge 브랜치이름 --edit  # --edit 옵션 없으면 커밋 메시지 자동생성, 옵션 붙이면 메시지 에디트

$ git reset --hard HEAD^ # 방금 병합한 내용 취소

$ git branch -d 브랜치이름 # 병합을 완료한 브랜치만 삭제 가능

$ git merge --abort # 병합시 충돌일 경우, 병합 취소
$ git ls-files -u # 충돌한 파일들의 집합 확인
$ git commit -m "resolve complicit" # 충돌 해결 후 커밋

$ git branch --merged # 병합한 브랜치 표시
$ git branch --no-merged # 병합하지 않은 브랜치 표시

$ git rebase 원본브랜치  #  merge(현재기준에서 파생브랜치랑 병합)와 반대로 파생에서 원본으로 병합, 브랜치의 HEAD 포인터가 다름
$ git checkout 원본브랜치
$ git merge 대상브랜치  # rebase 후에 merge 다시 함.

$ git rebase --continue  # rebase 병합시 충돌 났으면 그것을 해결하고 충돌된 부분들을 한
```

```
단계씩 진행, 리베이스 안되면 git add 명령어

$ git rebase --abort # 리베이스 취소

$ git rebase -i HEAD~3 # 커밋 묶기, 이 예에서는 커밋 3개가 커밋 하나로 변경
```

원격 repository

- Protocols; Local, HTTP, SSH, Git
- 서버 repository에서 혹은 서버 repository으로 pull, push
- 업스트림 트래킹; 리모트 브랜치와 로컬 브랜치를 연결하는 중간 다리 역할
- ./git/config

```
$ git remote add 원격저장소별칭 폴더경로 # 로컬 저장소를 서버로 이용
$ git remote add 원격저장소별칭 원격저장소URL # 원격 저장소 연결

$ git remote # 원격 저장소의 이름 출력
$ git remote -v  # verbose, URL도 확인

$ git remote rename 변경전이름 변경후이름
$ git remote show 원격저장소이름 # 상세 정보

$ git remote rm 원격저장소이름 # 원격 저장소 삭제

$ git push 원격저장소별칭 브랜치이름 # 원격저장소/브랜치에 현재 브랜치 전송
$ git push -u 원격저장소별칭 브랜치이름
$ git push 원격저장소별칭 로컬브랜치이름:새로운원격브랜치



$ git pull
$ git fetch

$ git merge 원격저장소별칭/브랜치이름

$ git branch -r # 리모트 브랜치 목록
$ git branch -a # 모든 브랜치 정보 확인
$ git branch -vv # 트래킹 브랜치 목록

$ git checkout --track 원격저장소별칭/원격브랜치 # 업스트림 브랜치 생성

$ git checkout -b 새이름 원격저장소별칭/브랜치이름  #
$ git branch -u 원격저장소별칭/브랜치이름 # 업스트림 연결, -u(==--set-upstream-to),
기존 브랜치를 특정 원격 브랜치로 추적

$ git push --set-upstream 원격저장소별칭 원격브랜치 # 업스트림 설정

$ git merge 원격저장소별칭/브랜치이름 # 리모트 브랜치와 병합
```

```
$ git checkout -b 임시브랜치이름 원격저장소별칭/원격브랜치 # 원격 브랜치의 포인터를 사용하여 임시 브랜치 생성하거나 직접 체크 아웃

$ git branch -d 브랜치이름 # 브랜치 삭제, -d 옵션은 스테이지 상태가 깨끗할 때만 삭제 허용
$ git branch -D 브랜치이름 # 브랜치 강제 삭제

$ git push 원격저장소별칭 --delete 리모트브랜치이름 # 리모트 브랜치 삭제
```

git flow stories

- 도구의 사용법 보다 사용 방법이 더 중요하다.
- 메인 프로젝트는 놔두고 개발 프로젝트로 작업하기
  - master에서 develop 브랜치 만들어 작업하고 develop에서 release 브랜치 생성, release
- 한참 개발 중인데 급한 버그 수정 요청이 들어왔다
  - master에서 hotfix 브랜치를 만들어 작업하고 master와 현재 개발중인 브랜치로 merge
- 하나의 프로젝트 여러 명의 개발자
  - develop 브랜치에서 각 개발자별로 브랜치 생성후 작업 (보통 기능별로 개발자에게 분배) 혹은 develop에서 기능별 브랜치 생성
- git-flow, github-flow, gitlab-flow 가 조금씩 다르다.

submodule

- 저장소를 모듈화, 저장소 분리
- 저장소 하나가 다른 깃 저장소를 포함하는 형태
- 메인 저장소 ↔ 부 저장소
- 2개 이상인 저장소를 부모/자식 관계로 연결
- 서버 저장소를 서브폴더 형태로 취급
- .gitmodules
- 메인에서 서브저장소 추가 → .gitmodules 추가 → 커밋
- 저장소마다 별도로 커밋 수행
- 메인 저장소는 서브모듈의 변경 내용을 모니터링, 서브모듈이 변경되면 메인 커밋

```
$ git submodule -help

$ git submodule add 원격저장소URL 폴더이름 # 메인저장소에서 작업, 서브저장소 등록
$ git add .gitmodules  # .gitmodules 등록
$ git commit -m "add submodule" # 커밋

$ git submodule init # 서브모듈 초기화, 하위 저장소의 내용을 가져오기 위해. 메인저장소에서 실행
$ git submodule update # 서브모듈 업데이터

# submodule 사용시, 아래 두 줄은 세트라고 생각하면 됨.
$ git pull origin master
$ git submodule update
```

fork, pull request

- PR(Pull Request) @github, MR(Merge Request) @gitlab
- fork → clone forked repository → make branch and move to branch on forked repository → coding jobs on branch → push → pull request

Etc

- refs, reflog
- file annotation; blame,
- replace; 기존 커밋을 다른 커밋인 것처럼 변경
- garbage collect; 연결고리가 없는 고립된 객체들, 주로 리셋/리베이스 등을 자주할 때 발생
- prune
- rerere(reuse recorded resolution); 어떤 문제로 충돌이 발생할 때 이를 기록

```
$ git rev-parse 브랜치이름 # 브랜치이름의 해시값 확인
$ git show 해시값

$ ls .git/refs -all # 저장소 refs 파일 목록

$ git reflog  # reflog, 시스템에서 정의한 며칠 간의 기록만 보관

$ git blame 파일이름 # 누가 코드의 어느 라인을 수정했는지 파악할 때 유용
$ git blame -L 시작줄,마지막줄 파일이름. -e: 사용자 이름 대신 이메일 출력, -w: 공백 문자 무시, -M: 같은 파일 내에서 복사나 이동 감지, -C: 다른 파일에서 이동이나 복사된 것을 감지

$ git replace 커밋ID1 커밋ID2 # 커밋ID1 -> 커밋ID2 연결

$ git gc --auto

$ git prune --dry-run --verbose # 객체 삭제, --dry-run: 실행하지 않고 작업할 내역만 출력, --verbose: 작업한 결과 출력
$ git reflog expire --expire=now  --expire-unreachable-now --all  # reflog 삭제
$ git prune --expire now -v # 객체 삭제 실행

$ git remote prune # 원격 저장소의 브랜치를 병합한 후 삭제, 삭제된 원격 저장소 브랜치는 더이상 참조할 수 없다
$ git fetch --prune # 오래된 브랜치 정리

$ git config rerere.enabled true  # rerere 기능 사용, --global 옵션 가능
$ git rerere status # 상태
```

references of git

- git 간편 안내서

- Pro git 온라인 북 - 한글판
- git-flow
- git 사용자 설명서

Git 사용법 정리

```
*basic concept
workspace -> staging area(index) -> local repository -> remote repository
stash

*file status lifecycle
1. untracked; git에 의해 추적되지 않는 파일. add the file 후 tracked로 변경됨
2. tracked; git에 의해 추적되는 파일. remove the file 하면 untracked로 변경됨
  2.1 unmodified; 수정되지 않은 파일. edit the file 후 modified로 변경됨
  2.2 modified; 수정된 파일. stage the file
  2.3 staged; staging area에 있는 파일. commit 후에는 unmodified로 변경됨


*기본 동작
만들기
(저장소 받아오기)
파일 저장: 생성, 삭제, 수정
로컬 저장소 저장: 생성, 삭제, 수정
원격 저장소 저장: 생성, 삭제, 수정
브랜치: 생성, 삭제, 수정, 이동, 병합

*install git
-Unix/Linux
-Mac
-Windows


*config git
.gitignore
Glob pattern


*starting w/ git
git init; create local repository
git remote add <repository>; connect local repository with remote
repository
git pull; get data from remote repository


git clone <repository>
ex) git clone /local/repository/path
ex) git clone user@hostname:/remote/repository/path
```

```
*basic commands
git add; workspace -> staging area(index)
git commit -a; workspace -> local repository
git push; local repository -> remote repository


git fetch; remote repository -> local repository
git merge; merge

git pull; fetch & merge

git status; modified, staged or unstaged
git diff; show differences between staged and unstaged
git diff --staged or git diff --cached; show differences between staged and
committed

*snapshot

*branch
-create branch
git branch <name>
git checkout <name>

git branch -b <name>; branch & checkout

-delete branch
git branch -d <name>


-merge branch
git checkout master
git merge branch


-list branch
git branch
git branch --no-merged
git branch -merged

*rebase

*tag
-create tag
git tag <name>
git tag -a <name>


-delete tag
git tag -d <name>
```

```
-list tag
git tag


*rollback
git checkout --; rollback unstaged file

git reset HEAD; unstage staged file

git fetch origin; rollback commited file
git rest --hard oring/master

*stashing


 * gitignore 재 적용
git rm -r --cached .
git add .
git commit -m "git ignore applied and fixed untracked files"
git push

git pull
git add .
git commit -m "git ignore applied"
git push
references
```

pro git 한글판 http://git-scm.com/book/ko/ git 간편안내서 http://rogerdudler.github.io/git-guide/index.ko.html

From:
https://theta5912.net/ - **reth**

Permanent link:
https://theta5912.net/doku.php?id=public:computer:git&rev=1629706155

Last update: **2021/08/23 17:09**