**Git**

참고

```
https://git-scm.com/book/ko/v2
https://backlogtool.com/git-guide/kr/
https://rogerdudler.github.io/git-guide/index.ko.html
```

Git, 형상관리

#### 기초 개념

https://git-scm.com/book/ko/v2/%EC%8B%9C%EC%9E%91%ED%95%98%EA%B8%B0-Git-%EA%B8%B0%EC%B4%88

#### SVN vs Git

- Data 관리 방법
  - SVN : **시간순**으로 관리하면서 파일들의 집합 관리
  - Git : **Snapshot의 Stream**으로 관리. 파일이 달라지지 않으면 새로 저장하지 않음. **단지 이전 상태의 파일에 대한 링크만 저장**.
    - Snapshot : 컴퓨터 파일 시스템에서 과거의 한 때 존재하고 유지시킨 컴퓨터 파일과 디렉터리의 모임.

#### 세 가지 영역과 상태



### 세 가지 영역

- Working Directory : Project의 특정 버전을 Checkout한 것.
- Staging Area : Git Directory에 존재. "Index"라고도 불림. 곧 Commit 할 파일에 대한 정보를 저장하는 단순한 파일.
- Git Directory (Repository) : **Git의 핵심**. Git이 프로젝트의 메타데이터와 객체 데이터베이스를 저장하는 곳. 다른 컴퓨터에 있는 저장소를 Clone할 때 생성됨.

### 세 가지 상태

- Comitted : Git Directory에 있는 파일들인 상태
- Staged : 파일을 수정하고 Staging Area에 추가한 상태
  - 예) 10개의 파일을 수정했지만 필요없는 3개 제외하고 7개의 파일만 Commit하고 싶을 때 (≒ Commit할 파일 등록)
- Modified : Checkout 이후 수정했지만, 아직 Staging Area에 추가하지 않은 상태

**명령어 (Local repository)**

#### 저장소 생성

저장소로 지정할 Directory로 이동 후 수행

```
#mkdir DIRECTORY
#cd DIRECTORY
git init
```

#### 변경 내용 반영

"3가지 상태"를 따른다.

1. 파일 수정
2. 저장소에 제출할 파일 추가 (= Index 추가)
3. 저장소에 제출

**저장소에 제출할 파일 추가**

```
git add <FILE 이름>
```

**저장소에 제출**

```
git commit
#이후 commit message 작성
#특별한 경우가 아니면 vim이 열리는 데, i → 편집 → ESC → :wq → ENTER
```

**저장소에 제출할 파일 모두 추가 + 저장소에 제출 + Commit message**

```
git commit -a -m "COMMENT"
#git commit -am "COMMENT"

# -a : all
# -m : message
```

https://git-scm.com/docs/git-commit

#### Branch 생성 및 전환

### 조회

```
git branch
```

### 생성

```
git branch <Branch 이름>
```

- A successful Git branching model
  - https://backlogtool.com/git-guide/kr/stepup/stepup1_5.html

### 전환

```
git checkout <Branch 이름>
```

### 생성 + 전환

```
git checkout -b <Branch 이름>
```

#### Branch 병합

| merge | rebase |
| --- | --- |
| 모든 이력 남음.<br>(정확한 이력 저장 용도) | 이력이 단순해짐.<br>(통합 이력 용도) |

- 적절한 사용 예
  - https://backlogtool.com/git-guide/kr/stepup/stepup1_4.html

### merge

https://git-scm.com/book/ko/v2/Git-%EB%B8%8C%EB%9E%9C%EC%B9%98-%EB%B8%8C%EB%9E%9

C%EC%B9%98%EC%99%80-Merge-%EC%9D%98-%EA%B8%B0%EC%B4%88

※ 현재 Branch로 병합되기 때문에 반드시 현재 Branch가 어디인지 확인 필요!

```
git merge <Branch 이름> [<commit>]

#<commit>
## --ff    (default, fast-forward) : 병합 흔적 남지 않음.
## --no-ff (non-fast-forward)      : 병합 흔적 남음.
```

### 종류

# fast-forward

조상 Brunch를 **“master”**,
병합하려는 Brunch를 **“hotfix”**라고 할 때,

병합하려는 시점에서

1. hotfix가 master보다 더 최신 내용이고,
2. hotfix가 master 내용을 모두 포함하고 있고,
3. master 내용이 갱신되지 않았을 때 (= hotfix와 내용이 같을 때)

병합.
단, 병합 이후 “master 내용 = hotfix 내용”이므로 hotfix 삭제.

- 예) master에서 긴급히 수정해야할 Bug가 있을 경우

```
# 1.현재 Brunch가 master일 때, "hotfix" Burnch 생성 및 전환
git checkout -b hotfix

# 2. 수정
vim FILE

# 3. commit
git commit -a -m "fixed bug"

# 4. master에 hotfix 병합
git checkout master
git merge hotfix

# 5. hotfix의 의미가 없으므로 삭제
git branch -d hotfix
```

**☞ 병합 흔적이 Commit graph에 남지 않는다.**

# 3-way merge

조상 Brunch를 **“master”**, 병합하려는 Brunch를 **“issues”**라고 할 때,

- 병합하려는 시점에서
  master가 수정되어 issues의 조상 Brunch와 내용이 다른 경우

Git은 공통조상을 자동으로 찾아 Merge하고, 현재 Commit(= Branch Pointer)이 Merge한 Commit을 가리키도록 한다.
Merge 수행시 Conflict가 발생하는 데, 충돌 해결을 참조.

**☞ 병합 흔적이 Commit graph에 남는다.**

## 충돌 해결

충돌이 발생하면 **(<Branch 이름>|MERGING)**으로 표시됨.

- 충돌난 파일을 열고, 편집 후 Index 추가 및 Commit.

## 변경/충돌 내역 확인

**※ 만약 git pull 명령을 이용했다면 알기 어려움.**

```
git diff
```

```
git diff <원래 Branch> <비교 대상 Branch>
```

**rebase**

## Branch 이력 확인하면서 병합

```
git rebase <branch 이름>

git rebase <option>
#<option>
## --continue : Conflit 발생시 해결 후 계속 작업 진행.
```

```
## --skip      : 강제 병합.
## --abort     : Conflict 발생시 취소.
```

### Commit 내역 병합

```
git rebase -i <revision>
#<revision>
## 이어서 "~ 혹은 ^"를 붙이면 개수만큼 되돌림.

#지정한 <revision>과 최근 Commit을 병합하여 새로운 Commit 생성.
```

## 관례

- 남기는 Commit message 앞에는 접두어로 **pick** 붙인다.
- 없애는 Commit message 앞에는 접두어로 **fixup** 붙인다.
- **Commit SHA-1 Checksum 값**은 꼭 남겨야 한다.
- 기존의 Commit message를 수정하지 않는다.

#### Tag

#### 조회

```
# 단순 Tag List 조회
git tag -l

# Tag 및 SHA-1 Checksum 값 조회
git show-ref --tags

# 한 Tag의 변경 이력 조회
git show <Tag 이름>
```

#### 추가

### light weight

Version 이름만 붙이기

```
git tag <Tag 이름> <Checksum 4자 이상>
# 특정 <Checksum 4자 이상>을 생략하면 "현재 Branch의 최신 Commit"에 반영
```

### annotated

Version, 작성자, Comment 붙이기

```
git tag -a <Tag 이름> <Checksum 4자 이상>
# 이후 tag message 작성.

# 특정 <Checksum 4자 이상>을 생략하면 "현재 Branch의 최신 Commit"에 반영
```

#### Commit 변경

#### 되돌리기

## 전체 되돌리기

| revert | reset |
|---|---|
| **이전 Commit을 삭제하지 않고,** 되돌리려는 Commit을 새 Commit으로 생성 | **이전 Commit을 삭제하고,** 해당 시점의 Commit으로 변경 |

예) "□ → □ → □ → □ → □" Commit 상태에서 ⓔ에서 ⓒ를 되돌리는 작업을 할 때

| revert | reset |
|---|---|
| □ → □ → □ → □ → □ → □ | □ → □ → □ |

# revert

```
git revert <revision>
#이후 Commit message 추가

# <revision>
## HEAD를 넣으면 가장 최근 Commit 기준
## SHA-1 Checksum 값을 넣으면 해당 시점 기준
### 이어서 "~ 혹은 ^"를 붙이면 개수만큼 되돌림.
### 예) git revert HEAD~~ : 최종 Commit 기준으로 2단계 이전 Commit을 추가
```

# reset

- 특정 시점으로 되돌리기

```
git reset <mode> <revision>

# <mode>
## --hard  : HEAD, Index, Working Directory (완전히 되돌림.)
## --mixed : HEAD, Index                    (default, Index 되돌림. 파일내용 변경X)
## --soft  : HEAD                           (commit만 되돌림. 파일내용 변경 X)

# <revision>
## HEAD를 넣으면 가장 최근 Commit 기준
## SHA-1 Checksum 값을 넣으면 해당 시점 기준
### 이어서 "~ 혹은 ^"를 붙이면 개수만큼 되돌림.
### 예) git reset --hard HEAD~~ : 최종 Commit 기준으로 2단계 이전 Commit을 완전히 되돌림
```

- 원래 상태로 되돌리기 (git reset 명령 실행 이전으로 되돌리기)

```
git reset <mode> ORIG_HEAD
```

## 파일 하나만 되돌리기

```
git checkout <revision> -- <file 이름>

# <revision>
## HEAD를 넣으면 가장 최근 Commit 기준
## SHA-1 Checksum 값을 넣으면 해당 시점 기준
### 이어서 "~ 혹은 ^"를 붙이면 개수만큼 되돌림.
### 예) git checkout HEAD~~ -- helloworld.java : 최종 Commit 기준으로 2단계 이전 Commit으로 완전히 되돌림
## 생략하면 "git add <file 이름>" 명령 실행 시점으로 되돌림.
```

### 완료한 Commit 수정

- 너무 일찍 Commit 했을 때
- 어떤 파일을 빼먹었을 때
- 커밋 메시지를 잘못 적었을 때

최종 Commit을 대체하는 새 Commit 생성.
즉, 마지막 Commit과 현재 Commit 하지 않은 상태를 병합하여 새 Commit을 만든다.

```
git commit --amend
```

#### 불필요한 Directory 및 File 무시

- .gitignore 생성

```
touch .gitignore
#조회는 ls -al
```

- 자동 생성 후 내용 붙여 넣기
  - http://gitignore.io

#### 저장소 상태 확인

```
git status
```

#### Log 조회

```
git log
#--graph 사용하면 ASCII 코드로 흐름 보임.
#-<정수> 입력하면 최근순으로 정수 개수만큼 보임.
```

#### 명령어 (Remote repository)

#### 원격 저장소 → 로컬 저장소 내용 가져오기

```
git clone <원격 서버 주소>
# URL 대신 로컬 저장소도 복제 가능!
```

#### 로컬 저장소를 원격 저장소에 연결

- 로컬 저장소에 이미 해놓은 작업이 있을 때, **빈** 원격 저장소에 옮기고 싶은 경우
  - ☞ **※ 만약 Github을 이용할 경우 README.md 파일 생성 금지!**
    - **같은 이름의 Branch가 있고, 내용이 다른 경우 Push 거부된다!**

```
git remote add <저장소 별칭><원격 서버 주소>
```

```
#<저장소 별칭>은 관례적으로 origin 사용.
```

#### "로컬 저장소 → 원격 저장소" 작업 내역 올리기

```
git push <원격 저장소 별칭> <로컬 브랜치 이름>
#<로컬 브랜치 이름>에 --all 이면 모든 로컬 브랜치 Push
```

#### 로컬 저장소 갱신

| | fetch | pull |
|---|---|---|
| 세세한 변경내역을 알 수 있는가? | O | X<br>(fetch + merge) |

☞ fetch 추천

```
git fetch
```

```
git pull
```

#### IDE에서 Bash 사용방법

#### Eclipse

Gonsole 설치

- http://rherrmann.github.io/gonsole/

1. 위 링크의 “install” 버튼을 Eclipse에 Drag & Drop
2. Help → Ecllipse Marketplace → **Gonesole** 검색 및 설치

#### Visual Studio

Git Tools 설치

- 도구 → 확장 및 업데이트
- 온라인 → **Git Tools** 검색 및 설치

#### Intellij

- File → Settings
- Tools → Terminal

```
# Shell path (64x 기준)
C:\Program Files\Git\bin\sh.exe --login -i
```

From:
https://wiki.ledyx.xyz/ - **Ledyx WIKI**

Permanent link:
**https://wiki.ledyx.xyz/doku.php?id=git&rev=1463155268**

Last update: **2026/10/24 07:03**