|  |  |
| --- | --- |
| **Program / Project Name:** | Git |
| **Checklist / Template Completed by:** | TBD |
| **Date Completed:** | Click here to enter a date. |

**Tips for Git**

Revision: 0.1

**Detailed Revision History**

|  |  |  |  |
| --- | --- | --- | --- |
| **Rev** | **Date** | **Editor(s)** | **Description of change** |
| 0.1 | 20-MAY-2019 | Sang-Gu Kang | Initial Draft |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

**Table of Contents**

[1 Install 5](#_Toc29491382)

[1.1 Git 5](#_Toc29491383)

[1.2 TortoiseGit 5](#_Toc29491384)

[1.2.1 Download 5](#_Toc29491385)

[1.2.2 Install 5](#_Toc29491386)

[2 사용법 10](#_Toc29491387)

[2.1 Command 10](#_Toc29491388)

[2.2 History 관리(롤백, 리버트, 리셋, 리베이스) 23](#_Toc29491389)

[2.2.1 정확한 작업 취소 방법 선택 23](#_Toc29491390)

[2.2.2 적절한 명령어 선택 플로우 차트 24](#_Toc29491391)

[2.2.3 실험 작업을 위한 브랜치 사용 24](#_Toc29491392)

[2.2.4 단계적 리베이싱 26](#_Toc29491393)

[2.2.5 잃어버린 작업 찾기 개요 28](#_Toc29491394)

[2.2.6 파일 복구하기 30](#_Toc29491395)

[2.2.7 커밋 작업하기 33](#_Toc29491396)

[2.3 상황별 용도 38](#_Toc29491397)

[2.3.1 리모트 저장소 38](#_Toc29491398)

[2.3.2 태그 38](#_Toc29491399)

[2.3.3 브랜치 39](#_Toc29491400)

[2.3.4 되돌리기 Reset과 checkout 39](#_Toc29491401)

[2.3.5 원격 브랜치 삭제 40](#_Toc29491402)

[2.4 브랜치 워크플로우 40](#_Toc29491403)

[2.4.1 Long-Running 브랜치 40](#_Toc29491404)

[2.4.2 토픽 브랜치 40](#_Toc29491405)

[3 Definition and Explanation 41](#_Toc29491406)

[3.1 HEAD 41](#_Toc29491407)

[4 Tools 41](#_Toc29491408)

[4.1 Beyond Compare 41](#_Toc29491409)

[5 Useful Tip 41](#_Toc29491410)

[5.1 Overlay Icon 보이게 하기 41](#_Toc29491411)

# What is Git

## 스테이지와 커밋



# Install

## TortoiseGit

TortoiseGit 설치파일과 언어팩을 다운로드한 후 설치한다.

한글 설정을 놓쳤다면 settings에서 설정해 준다.



사용자 정보를 설정해 준다



# 사용법

## 작업 되돌리기

### 작업트리에서 수정한 파일 되돌리기

git checkout -- <file name>

### staging 되돌리기

git reset HEAD <file name>

HEAD 다음에 파일 이름을 지정하지 않으면 스테이지에 있는 모든 파일을 되돌린다.

### 최신 커밋 되돌리기

git reset HEAD^

HEAD^는 현재 HEAD가 가리키는 브랜치의 최신 커밋을 가리킨다. 최근 3개의 커밋을 취소하려면 $git reset HEAD~3처럼 사용한다.

# 사용법

## Command

git < > [ ] …

#### add

[file name]: 특정한 파일만 준비영역에 추가하기  
[directory name/\*]: 특정한 경로의 모든 파일을 준비영역에 추가하기  
[--all]: 작업폴더에 있는 모든 파일을 준비영역에 추가하기. 현재 git에 기록되지 않은 모든 파일도 준비 영역에 추가된다. 이는 매우 탐욕스런 명령이다. 주의가 필요하다.

[.]: git add .  
작업 트리에서 수정한 파일을 한꺼번에 올릴수 있다. (=git add --all)

파일을 새로 추적할 수 있다(untracted -> tracted). git add 명령은 파일을 새로 추적할 때(새로 만든 파일을 add하여 tracted상태로 만들고자 할 때)도 사용하고 수정한 파일(이미 tracted된 파일이지만 modified된 상태의 파일)을 Staged 상태로 만들 때도 사용한다. Merge할 때 충돌난 상태의 파일을 Resolve 상태로 만들때도 사용한다.

git add README.md process-diagram.png -> 2개의 파일을 추가  
git add branch-naming-rules.png -> 1개의 파일을 추가  
git add –all -> 모든 파일을 추가  
git add ca5g/\* -> ca5g 디렉토리 안의 모든 파일을 추가  
git add \*.svg -> svg 확장자를 가진 모든 파일을 추가

[--update]: git에 기록된 적이 있는 모든 파일 중 마지막 커밋 이후 수정된 파일을 준비 영역에 추가  
git add --update

[--patch]: 한 파일의 일부만을 준비 영역으로 보내 커밋 준비를 한다.  
git add --patch <file name>

[-i]: 대화형 모드

#### branch

브랜치를 생성하거나 삭제하고, 브랜치의 목록을 보여준다.

[-v]: 브랜치 목록과 함께 마지막 커밋 메시지도 함께 보여준다.  
git banch -v

[--merged]: merged된 브랜치 목록을 보여준다.  
git branch –merged

[--no-merged]: merge하지 않은 브랜치 목록을 보여준다.  
git branch –no-merged

[-d]: 브랜치 삭제.  
git branch -d testing

[-D]: merge하지 않은 브랜치를 강제로 삭제한다.  
git branch -D testing

[-vv]: 추적 브랜치가 현재 어떻게 설정되어 있는지 확인하려면

[--all]: 저장소의 모든 브랜치 목록을 볼 때 사용. 로컬 브랜치와 원격 브랜치 목록을 같이 볼 수 있다.  
목록에서 remotes/origin으로 시작하는 목록을 볼 수 있는데, remotes는 ‘local이 아니다’라는 뜻이고, origin은 기본 규약에서 ‘나의 사본은 여기에서 복제된 것’의 의미이다.

[--list]: 모든 로컬 브랜치의 목록을 얻는다.

[--remotes]: 모든 원격 브랜치의 목록을 얻는다.  
실제 원격 브랜치 이름에는 remotes가 붙지 않는다. 그저 해당 브랜치가 어떤 종류인가를 설명해 주는 정보일 뿐이다. 그래서 사용할 수 있는 원격 브랜치의 이름 목록을 얻으려면 –remotes를 사용하면 된다.

Remote branch생성은 [리모트 저장소](#_리모트_저장소) 참조.

#### checkout

다른 브랜치로 이동할 때 사용한다.

[-b]: 브랜치를 만들면서 checkout까지 한번에 한다.  
git checkout -b <new branch name>  
git checkout -b iss53

git checkout -b <new branch name> <branch parent>  
git checkout -b iss53 master

리모트 트래킹 브랜치에서 시작하는 새 브랜치를 만들려면  
git checkout -b serverfix origin/serverfix  
git checkout -b tmAnalysis origin/feature/HA-3653-tm-analyzer

[-d]: 브랜치 삭제.  
git branch -d hotfix

[--track]: 원격 브랜치의 로컬 사본을 생성한다.  
git checkout --track <remote\_name/branch>  
git checkout –track origin/video-lessons

[--filename]: 작업 폴더 안의 파일 수정사항을 폐기한다. 수정사항은 준비 영역에 추가되거나 커밋되지 않은 상태. checkout으로 되돌린 내용은 다시 복구할 수 없다.  
git checkout -- signalTm.cpp

#### cherry-pick

git cherry-pick -x <commit>

#### clean

워킹 디렉토리 청소하기. 작업하고 있던 파일을 stash하지 않고 단순히 그 파일들을 치워버리고 싶을 때.

[-f]: 강제로 그냥 지운다.

[-n]: 가상으로 실행해 보고 어떤 파일들이 지워질지 알려달라는 뜻.  
git clean -d -n

#### clone

[url]



#### commit

[--amend]: 한줄 커밋 후에 추가적인 설명이 필요할 때, 커밋을 개정할 수 있다. commit –m ‘message’으로 커밋한 후에 commit --amend를 입력하면 에디터가 뜨는데 이곳에 내용을 입력하면 된다.  
git add --all  
git commit -m “CH05: Adding technical edits.”  
git commit --amend  
[VIM Editor에 커밋 내용 입력]

[-m]: 메시지와 함께 커밋.

#### config

[--global] 사용자 이름과 이메일을 설정한다.  
git config --global user.name “SangguKang”  
git config --global user.email [kangsanggu74@gmail.com](mailto:kangsanggu74@gmail.com)

git config --global alias.st status  
git config --global alias.br branch

user name과 email을 pull/push할 때마다 입력하지 않도록 변경   
git config credential.helper store --global  
git pull or git push  
name과 email입력(한번은 입력해야 한다) -> 이후부터는 입력이 필요 없다

[--list] 설정한 모든 것을 보여준다.



git config --global core.editor “vim”  
git의 편집기 설정

#### diff

단순히 파일이 변경됐다는 사실이 아니라 어떤 내용이 변경됐는지 살펴보려면 git status 대신 git diff 명령을 사용해야 한다. Patch처럼 어떤 라인을 추가했고 삭제했는지가 궁금할 때 사용한다.

이 명령은 working directory에 있는 것과 Staging Area에 있는 것을 비교한다. 그래서 수정하고 아직 Stage하지 않은 것을 보여준다.  
**git diff**

만약 commit하려고 Staging Area에 넣은 파일의 변경부분을 보고 싶으면 **git diff –staged** 옵션을 사용한다. 이 명령은 저장소에 commit한 것과 Staging Area에 있는 것을 비교한다.

[--check]: 공백문자를 검사해서 공백문자에 대한 오류를 확인할 수 있다.  
git diff --check



Commit 하기 전 마지막으로 실수한 것이 없는지 확인할 수 있는 기회를 제공한다.

#### fetch

리모트 저장소의 데이터를 모두 로컬로 가져오지만, 자동으로 Merge하지는 않는다(git pull 명령 사용은 merge시킬 수 있다).  
원격 브랜치 목록은 자동으로 최신상태로 유지되지 않는다. 목록을 업데이트하려면 fetch 명령어를 사용한다.  
git fetch origin

#### grep

[-n]: 찾을 문자열이 위치한 라인 번호도 같이 출력한다.  
git grep -n html

[--count]: 어떤 파일에서 몇 개나 찾았는지 알고 싶을 때 사용한다.  
git grep –count html

#### init

.git이라는 하위 디렉토리를 만든다. .git 디렉토리에는 저장소에 필요한 뼈대 파일(Skeleton)이 들어 있다.

#### log

저장소의 commit 히스토리를 시간 순으로 보여줌.

[-p]: 각 commit의 diff 결과를 보여준다.

[-2]: 최근 두 개의 결과만 보여준다.

[--stat]: 각 commit의 어떤 파일이 수정됐는지, 얼마나 많은 파일이 변경됐는지, 또 얼마나 많은 라인을 추가하거나 삭제했는지의 통계 정보를 보여준다. 커밋에 관련된 파일까지 함께 살펴볼 때 사용한다.  
git log --stat

[--oneline]: log 메시지를 한 줄로 보기

[--pretty=oneline]: 각 commit을 한 라인으로 보여준다.

[--decorate]: 브랜치가 어떤 커밋을 가리키는지 알 수 있다. git log –oneline --decorate

기타 다른 많은 옵션들은 pro git 등을 참조한다.



[--graph]: graphical하게 보여준다.  
git log --oneline --graph

#### merge

새로운 브랜치(origin/serverfix)의 내용을 merge하려면, git merge origin/serverfix

From\_branch를 to\_branch와 merge한다.  
git merge <from\_branch> <to\_branch>

#### mv

#### push

git push <remote repo name> <branch name>  
git push origin master

원격 저장소에 트랙킹된 저장소에서 git push 명령만 사용하면 알아서 원격 저장소에 push 된다.

git push origin :<branch-name>  
git push origin :feature01  
주의) 원격 저장소의 브랜치를 삭제.

git push origin <branch-name>  
git push origin feature01  
새로운 브랜치 저장소를 원격에 생성.

git push --delete <branch-name>  
git push --delete restoring\_old\_commit  
원격 저장소 삭제

#### pull

서버로부터 데이터를 가져와서 현재 로컬 브랜치와 서버의 추적 브랜치를 merge한다.

git pull <remote> <branch>  
git pull origin feature/HA-3653-tm-analyzer

#### rebase

remote 브랜치에 커밋을 깔끔하게 적용하고 싶을 때 사용한다. 이미 공개 저장소에 push한 커밋을 rebase하지 마라.

git rebase [branch]

git rebase [basebranch] [topicbranch]

#### reflog

저장소 로컬 사본에서 일어난 모든 일의 히스토리를 목록으로 볼 수 있다. 클론 이후부터 시작해서 코드에 영향을 주지 않는 브랜치 체크아웃 같은 행위도 포함된다.  
git reflog

#### remote

이미 설정된 원격 연결이 있는지 알아본다.  
$ git remote --verbose  
origin git@gitlab.com:SangguKang/book\_unix\_linux\_utility.git (fetch)  
origin git@gitlab.com:SangguKang/book\_unix\_linux\_utility.git (push)  
-> origin: 원격 연결의 별명  
-> 첫 번째 줄은 새 작업을 받아오는 장소(fetch)를 나타내고, 두 번째 줄은 새 작업을 업로드할 장소(push)를 나타낸다.

[-v] 단축 이름과 URL을 함께 볼 수 있다.

[show <리모트 저장소 이름>] 예)git remote show origin

[rename]

[remove]

[add]: 현재 작업중인 프로젝트에 팀의 저장소를 추가한다.

[rm]: 원격 저장소 삭제  
git remote rm <remote-name>

#### reset

돌아가려는 커밋으로 저장소는 재설정되고 해당 커밋 이후의 이력은 사라진다. 이미 push한 상태라면 reset을 사용하지 말고 revert만 사용해야 한다.

git reset <옵션> <돌아가고 싶은 커밋>  
옵션: --hard, --soft, --mixed  
git reset –hard a3bbb3c

* staged area(준비영역)에서 파일을 not staged 영역으로 되돌릴 때. 다시 말해 실수로 staged area에 너무 많은 파일을 add했는데 일부 파일을 unstaged area로 내려서 의미있는 일부만 commit할 수 있다. reset으로 unstaged area로 내려간 파일의 수정사항을 되돌리는 것은 아니다.  
  git reset HEAD <file name>: git reset HEAD signal.cpp

#### revert

이전 작업을 삭제. 커밋 히스토리는 그대로 두며, revert한 사실이 명시된 새로운 커밋이 생긴다. 브랜치는 게재됐고 작업 폴더는 정리된 상태이다.

git revert <돌아가고 싶은 커밋>: git revert a3bbb3c

#### rm

staged된 파일을 unstaged하게 할 때 사용한다.

왠만하면 사용하지 말 것!

실수로 지워진 파일을 복구하려면,  
git reset –hard HEAD

  
  


#### show

git show [commit]: 하나의 커밋의 상세 정보를 확인한다. 태그 정보와 커밋 정보를 모두 확인할 수 있다.  
git show <SHA-1>  
git show 1c002d  
git show <branch-name>  
git show master

git show tag: 해당 태그가 적용된 커밋의 상세 정보를 본다.  
git show import

#### show-branch

git show-branch

#### stash

아직 완료하지 않은 일을 커밋하지 않고 나중에 다시 돌아와 작업을 하고 싶을 때 사용한다. Stash는 modified이면서 tracked 상태인 파일과 Staging Area에 있는 파일들을 보관해 두는 장소이다.

#### status

파일의 상태를 확인한다.

[-s] or [--short] 현재 변경한 상태를 짤막하게 보여준다.

#### submodule

두 프로젝트를 서로 별개로 다루면서도 그 중 하나를 다른 하나 안에서 사용하려고 할 때 사용. 각 저장소의 커밋은 독립적으로 관리한다.

[add URL]: git 저장소를 서브모듈로 추가  
git submodule add <http://github.com/chaconinc/DbConnector>

[init]:

[update]:

git submodule status

#### tag

git tag <tag name> <커밋해쉬>  
git tag import ee6426b  
  
위와 같이 새로 추가한 tag는 git tag를 이용해 모든 태그 목록을 확인할 수 있고, 태그된 커밋을 선택해서 git show <tag name>을 이용해 해당 커밋을 검토할 수 있다.

[-a] annotated tag 만들기.

git tag: 모든 태그 목록을 본다.

## History 관리(롤백, 리버트, 리셋, 리베이스)

### 정확한 작업 취소 방법 선택



|  |  |  |
| --- | --- | --- |
| **원하는 것…** | **설명** | **해결책** |
| 작업 폴더 안의 파일의 수정사항을 폐기한다 | 수정사항은 준비영역에 추가되거나 커밋되지 않았다 | git checkout -- filename |
| 작업 폴더의 모든 저장되지 않을 수정사항을 폐기한다. | 파일은 준비 영역에 추가됐지만, 커밋되지는 않았다 | git reset --hard |
| 특정 커밋을 제외한 여러 커밋을 합친다 |  | git reset commit |
| 기록되지 않은 파일을 포함해 모든 저장되지 않은 파일을 삭제한다 | 수정된 파일은 커밋되지 않았다 | git clean -fd |
| 준비 영역의 모든 수정사항과 특정 커밋 전까지 커밋된 작업을 삭제하되, 작업 폴더의 새 파일은 삭제하지 않는다 |  | git reset --hard commit |
| 어떤 작업을 삭제하되, 커밋 히스토리는 그대로 둔다(롤포워드 roll forward) | 브랜치는 게재됐고 작업 폴더는 정리된 상태다 | revert commit |
| 브랜치 히스토리에서 하나의 커밋 삭제하기 | 수정사항은 커밋됐고, 작업 폴더는 정리됐고 브랜치는 게재되지 않았다 | rebase --interactive commit |
| 이전 작업을 계속 진행하되, 다른 커밋과 합친다 | squash 옵션을 선택한다 | rebase --interactive commit |

### 작업을 위한 Git 명령어

|  |  |
| --- | --- |
| **명령어** | **용도** |
| git checkout -b branch | branch라는 이름의 새 브랜치를 생성한다 |
| git add filename(s) | 저장소에 커밋하기 위해 파일을 준비 영역에 추가한다. |
| git commit | 준비 영역에 추가된 수정사항을 저장소에 저장한다 |
| git checkout branch | 현재 작업 브랜치를 특정한 브랜치로 교체한다 |
| git merge branch | 브랜치 branch의 커밋을 현재 브랜치에 통합한다 |
| git branch --delete | 로컬 브랜치를 삭제한다 |
| git branch -D | 커밋이 다른 곳에 통합되지 않은 로컬 브랜치를 삭제한다 |
| git clone URL | 원격 저장소의 로컬 사본을 생성한다 |
| git log | 현재 브랜치의 커밋 히스토리를 읽는다 |
| git reflog | 현재 브랜치의 확장 히스토리를 읽는다 |
| git checkout commit | 특정 커밋을 체크아웃하고 떨어진 HEAD 상태가 된다 |
| git cherry-pick commit | 하나의 브랜치 커밋을 다른 브랜치로 복사한다 |
| git reset --merge ORIG\_HEAD | 최신 병합을 통해 적용된 모든 커밋을 현재 브랜치에서 삭제한다 |
| git checkout -- filename | 수정됐지만 커밋되지 않은 파일을 복구한다 |
| git reset --hard HEAD | 수정된 모든 파일을 이전에 저장된 상태로 복구한다 |
| git reset commit | 특정 커밋 이후에 커밋된 모든 수정사항을 준비영역에서 삭제한다 |
| git rebase --interactive commit | commit 이후 추가된 커밋을 수정하거나 합친다 |
| git rebase - continue | 병합 충돌을 해결하라고 리베이싱 과정을 계속 진행한다 |
| git revert commit | 특정 커밋에 저장된 수정사항을 무력화한다. 이를 실행하면 공유 가능한 역전 커밋의 히스토리가 생성된다 |
| git log --oneline --graph | 현재 브랜치의 히스토리 그래프를 출력한다 |
| git revert --mainline 1 commit | 하나의 병합 커밋을 역전시킨다 |
| git branch --contains commit | 특정 커밋 객체를 포함하는 모든 브랜치 목록을 본다 |
| git revert --no-commit last\_commit\_to\_keep .. newest\_commit\_to\_reject | 여러 커밋을 역전시키고 되돌려지는 커밋 하나당 하나의 객체를 생성하는 대신 하나의 역전 커밋을 생성한다 |
| git filter-branch | 파일을 저장소에서 영원히 삭제한다 |
| git reflog expire | 확장 히스토리를 잊고 오직 저장된 커밋 메시지만 사용한다 |
| git gc --prune=now | 가비지 컬렉터를 실행해 모든 커밋되지 않은 수정사항을 로컬 메모리에서 삭제한다 |

### 적절한 명령어 선택 플로우 차트

#### 파일 수정 사항 제거



### 실험 작업을 위한 브랜치 사용

티켓 브랜치로부터 브랜치를 하나 더 만들어 실험적인 수정사항을 적용하자([예제1-1]). 결과가 만족스럽다면 실험 브랜치를 티켓 브랜치에 병합하면 된다([예제 1-2]). 실험 브랜치는 병합하고 나면 삭제해도 된다([예제 1-3]).

예제1-1 실험 브랜치를 사용해 수정사항 테스트

  
  


예제1-2 실험 브랜치를 메인 브렌치에 병합

  
  
  


예제 1-3 실험 브랜치 삭제

  
  


### 단계적 리베이싱

일반적으로 리베이스는 하나의 브랜치에서 그것의 부모 브랜치에서 만들어진 커밋을 최신 상태로 업데이트하는 과정이라고 한다.

#### 리베이스 시작하기

|  |
| --- |
| git checkout feature  git rebase master |

#### 리베이스 중에 발생한 삭제된 파일에 대한 충돌

|  |
| --- |
| First, rewinding head to replay your work on top of it…  Applying: CH10: Stub file added with notes copied from video recording lessons.  Usng index info to reconstruct a base tree…  A ch10.asciidoc  Falling back to patching base and 3-way merge…  CONFLICT (modify/delete): ch10.asciidoc deleted in HEAD and modified in CH10:  Stub file added with notes copied from video recording lessons.. Version CH10:  Stub file added with notes copied from video recording lessons. Of ch10.asciidoc  left in tree.  Failed to merge in the changes.  Patch failed at 0001 CH10: Stub file added with notes copied from video recording  lessons.  The copy of the patch that failed is found in:  /Users/emmajane/Git/1234000002182/.git/rebase-apply/patch  When you have resolved this problem, run “git rebase --continue”.  If you prefer to skip this patch, run “git rebase --skip” instead.  To check out the original branch and stop rebasing, run “git rebase --abort”. |

When you have resolved this problem, run “git rebase --continue”.

앞의 문구는 무엇을 해야 할지를 설명해 준다.  
1. 병합 충돌을 해결한다.  
2. 병합 충돌이 해결된 것 같으면 다음과 같은 명령어를 실행한다.  
git rebase --continue

기본 파일 비교 도구를 사용해 충돌이 일어난 파일을 열어서 1단계를 해결한다.  
git mergetool ch10.asciidoc

더 이상 병합 충돌 표시가 없으면 병합 도구를 끝내고 Git이 안내해주는 다음 단계로 넘어간다.  
git rebase --continue

|  |
| --- |
| ch10.asciidoc: needs merge  You must edit all merge confilicts and then mark them as resolved using git add |

Git에 문제가 무엇인지 명령어 status를 사용해 물어본다.  
git status

|  |
| --- |
| rebase in progress; onto 6ef4edb  You are currently rebasing branch ‘ch10’ on ‘6ef4edb’.  (fix conflicts and then run “git rebase --continue”)  (use “git rebase --skip” to skip this patch)  (use “git rebase --abort” to check out the original branch)  Unmerged paths:  (use “git reset HEAD <file>… “ to unstage)  (use “git add/rm <file>…” as appropriate to mark resolution)  deleted by us: ch10.asciidoc  no changes added to commit (use “git add” and/or “git commit -a”) |

수정사항을 적용하지 않으려면  
git reset HEAD ch10.asciidoc

충돌을 해결했다면  
git add book.asciidoc  
git status

|  |
| --- |
| rebase in progress; onto 6ef4edb  You are currently rebasing branch ‘ch10’ on ‘6ef4edb’.  (all conflicts fixed: run “git rebase --continue”)  Changes to be commited:  (use “git reset HEAD <file>… “ to unstage)  modified: book.asciidoc |

git rebase --continue

### 잃어버린 작업 찾기 개요

git에서 커밋된 작업을 완전히 삭제하기란 매우 어렵지만, 작업을 잃어버리는 일은 쉬운 편이다. 어떤 작업을 잃어버렸을 때, 가장 처음으로 할 일은 그 작업이 저장됐던 커밋을 찾는 것이다.

이때 유용한 명령이 git log와 git reflog이다(2.1 Command 참조).

명령어 log와 reflog 모두 저장소에 저장된 특정 상태의 커밋 ID를 보여준다. 이 커밋 ID를 찾기만 하면 상태를 확인하고 특정한 시점의 코드 베이스 상태를 임시로 복구할 수 있다.

현재 상태: br-master, b4faee3



저장소에서 특정 커밋 확인하기: git checkout 74dad73

  






특정 커밋을 체크아웃하면 해당 브랜치의 연결된 히스토리로부터 떨어지게 된다. 현재 상태를 저장하려면 새로운 브랜치를 체크아웃하고 현재 상태를 새 브랜치에 저장하면 된다. git checkout -b restoring\_old\_commit



수정할 것이 있으면 수정을 한다.



추가나 수정이 끝나면 새 브랜치를 기존 브랜치에 통합한다.



병합이 완료되면 임시 브랜치를 삭제해 로컬 저장소를 정리한다.



### 파일 복구하기

지워서는 안되는 파일이나 수정해서는 안되는 파일을 수정한 경우.

* 커밋 전이고, 준비 영역 전이라면



* 이미 준비영역에 추가했다면  
  -. 삭제 -> git add -> git status  
    
  -. 복구하고 싶은 파일을 준비영역에서 삭제  
    
  -. File이 작업 영역에 있으므로 checkout  
    
  -. 작업 폴더의 수정사항을 모두 취소하고, 이전 커밋에 저장된 파일로 되돌리길 원할 때 한번에 처리할 수 있다.  
    
  

### 커밋 작업하기

커밋은 특정 시점의 모든 파일의 상태를 포함하는 저장소의 스냅샷이라고 할 수 있다. reset을 사용해서 특정 커밋을 완전히 삭제할 수 있고, revert를 사용해서 특정 커밋의 효과를 역전시킬 수도 있고, rebase를 사용해서 커밋의 순서를 바꿀 수도 있다.

이번에는 다른 사람과 아직 공유하지 않은 커밋을 가지고 작업한다고 가정한다(예) 브랜치를 push하지 않은 상태).

#### 커밋 수정하기

커밋에 작은 수정 하나가 빠진 경우, 해당 커밋을 수정해서 수정 사항(추가 파일, 파일 수정, 메시지 업데이트)을 업데이트할 수 있다. 자주 간략한 한 줄 커밋 메시지를 잘 짜인 작업 요약문으로 변경하거나 활요할 수 있다.

**공유된 히스토리(작업을 이미 push 했다면)는 절대 변경하지 말라**.

git add --all  
git commit --amend

메시지만 업데이트할 경우, add 명령어는 생략하고 commit만 할 수 있다.  
git commit --amend

파일을 수정하거나 할 경우에는 위의 첫 예와 같이 add 명령어와 commit 명령어를 함께 사용한다.

이때, 새 수정사항이 기존 커밋에 추가되고 변경된 커밋 객체에는 새로운 ID가 할당된다.

#### Reset으로 커밋 합치기

reset 명령어를 사용해서 리베이스의 커밋 합치기(**squash**) 효과를 낼 수 있다. 앞서 말했지만 공유되지 않은 브랜치의 히스토리를 변경하는 데만 reset을 사용하는 것이 가장 좋다(즉, 로컬 브랜치나 서버에 push하지 않은 브랜치를 말하는 것이다).

구슬이 꿰여져 있는 줄을 생각해 보자. 그 줄에 구슬이 20개 달려있는데, 4번째 구슬을 쥐고 앞의 3개의 구슬을 빼내서 하나의 큰 구슬로 교체한다고 가정해 보자.

명령어 reset을 사용해 구슬의 끝 부분(가장 최신 커밋이 끝에 위치)을 기존의 맨앞 구슬에서 4번째 구슬로 지정할 것이다. 그리고는 실 끝에 있는 3개의 구슬을 떼어낸다. 빼낸 3개의 구슬을 합쳐 하나의 커밋으로 만들고 만들어진 커밋을 다시 줄에 추가하는 것이다.

-. log 명령어를 사용해 커밋 히스토리를 확인한다.  


-. 만일 줄의 끝에 놓고 싶은 구슬이1643448(이것이 끝에서 4번째 구슬)이라면, 남길 구슬을 손가락으로 잡고 나머지는 줄에서 빼낼 것이다.  


-. 이제 3개의 떨어진 구슬이 있다. 이 구슬은 저장소에 기록되지 않은 수정사항으로 나타날 것이다. 파일 콘텐츠는 수정되지 않을 것이다.  
  


-. 명령어 diff를 사용해 새 커밋에 어떤 변경사항이 포함되는지 확인할 수 있다.  


-. 앞의 떨어져나간 3개의 구슬(커밋)에 포함된 모든 수정사항을 하나의 커밋으로(하나의 큰 구슬) 합치려면 명령어 add를 사용해 해당 수정사항을 준비영역으로 추가해야 한다.  
  


-. 이제 관련 파일은 준비영역에 추가되었고 명령어 diff를 사용해도 저장소에 커밋할 내용이 보이지 않을 것이다. 준비 영역의 수정사항을 확인하려면 --staged 옵션을 사용한다.  


-. 커밋 과정을 완료한다. 이제 3개의 커밋은 하나의 커밋으로 합쳐질 것이다.  


#### 인터랙티브 리베이싱으로 커밋 수정하기

리베이싱은 논란이 많은 주제 중 하나이다. 리베이싱은 주로 히스토리가 기록된 방식을 바꾸는데 사용된다. 보통 작업 폴더의 파일 내용은 변하지 않는다. 잘못 사용하면 공유 브랜치에 혼란을 초래할 수 있다.

리베이싱을 사용하기 적절한 상황:

1. 브랜치를 업데이트할 때  
2. 커밋을 좀 더 읽기 쉬운 히스토리로 만들 때(이 절에서는 이 부분에 대해 언급)

인터렉티브 리베이싱에서는 여러 개의 작은(자잘한) 커밋을 하나의 온전한 아이디어로 합치는 기본 원칙을 볼 수 있다.

-. 시작점으로 삼을 커밋을 선택: 최근 커밋 3개(1643448, 376efa3, 1c6ab90)를 하나의 커밋으로 만들겠다고 결정했다. 커밋 74dad73을 시작점으로 선택하자.  


Abort!!!

#### 브랜치 병합 취소하기



## 1인 이상의 팀

## 상황별 용도

### 리모트 저장소

* 리모트 저장소 확인하기  
  git remote -v
* 리모트 저장소 추가하기  
  ???
* 리모트 저장소를 Pull 하거나 Fetch하기  
  git fetch [remote-name]
* 리모트 저장소에 push하기  
  git push origin master
* 리모트 저장소 살펴보기  
  git remote show [remote 저장소 이름] 예)git remote show origin
* 리모트 저장소 이름을 바꾸거나 리모트 저장소를 삭제하기  
  git remote rename pb paul  
  git remote remove paul
* 리모트 저장소에 새로운 브랜치 생성하기  
  git checkout -b tm-analyzer  
  git push origin tm-analyzer

### 태그

태그는 특정한 커밋을 정확히 찾아내기 위해 사용된다. 일종의 책갈피라 생각할 수 있다. 공유된 브랜치에서는 보통 배포 목적으로 사용되고 사용 방식도 전체 팀이 정한 규약에 따라야 한다.

* 새로운 태그 추가하기  
  git tag <tag name> 커밋 해쉬  
  git tag import fa04c30
* 태그 조회하기  
  git tag
* 태그 붙이기  
  ?
* 나중에 태그하기  
  ?
* 태그 공유하기  
  ?
* 태그를 checkout하기  
  ?

### 브랜치

* 새 브랜치 생성하기  
  git branch <branch-name>  
  git branch testing
* 새 브랜치 생성하기 - 기존 브랜치 내용에서  
  git checkout -b <branch-name>  
  git checkout -b feature01
* 리모트 브랜치 생성하기  
  git push origin <branch-name>  
  git push origin feature01
* 리모트 브랜치 삭제하기  
  git push origin :<branch-name>  
  
* 브랜치 이동하기  
  git checkout <branch-name>  
  git checkout testing

### 되돌리기 Reset과 checkout

* 파일 상태를 Unstaged로 변경  
  git reset HEAD <file name>
* Modified 파일 되돌리기(수정된 파일 되돌리기)  
  git checkout -- <file name>  
  매우 위험한 명령. 원래 파일로 덮어쓰기 때문에 수정한 내용은 전부 사라짐. 변경한 내용을 쉽게 버릴 수는 없고 하지만 당장은 되돌려야만 하는 상황이라면 Stash와 Branch를 사용한다.

### 원격 브랜치 삭제

더 이상 필요 없는 원격 브랜치 삭제하기

git push origin :<branch-name>  
git push origin :feature01

## 브랜치 워크플로우

### Long-Running 브랜치

개발자가 많이 선호하는 워크플로우. 배포했거나 배포할 코드만 master 브랜치에 merge해서 안정버전의 코드만 master 브랜치에 둔다. 개발을 진행하고 안정화하는 브랜치는 develop이나 next라는 이름으로 추가로 만들어 사용한다. 테스트를 거쳐서 안정적이라고 판단되면 master 브랜치에 merge한다.



### 토픽 브랜치

어떤 한 가지 주제나 작업을 위해 만든 짧은 호흡의 브랜치이다. Git에서는 브랜치를 하나 만드는데 드는 비용이 크지 않기 때문에 매우 일상적으로 브랜치를 만들고 merge하고 삭제한다.

# Definition and Explanation

## HEAD

Git의 축약어. 현재 브랜치의 가장 최신 커밋을 가리킨다.

# Tools

## Beyond Compare

3way merge를 지원한다.

좌측에는 BASE, 중간에는 LOCAL, 오른쪽에는 REMOTE가 있다.

**LOCAL**은 **현재 내가 작업중인 저장소**로 내가 수정한 부분이 있고, **REMOTE**는 merge를 하려고 하는 다른 사람이 수정한 부분이 있다. **BASE**는 LOCAL과 REMOTE의 조상이다. 즉, BASE로부터 파생된 것이라는 뜻이다. 그래서 merge를 할 때 BASE로 돌아갈 수도 있고, LOCAL을 참조할 수도, REMOTE를 참조할 수도 있다.

아래 창의 화살표를 이용해 원하는 부분을 변경하거나, 직접 입력하여 수정한다. 화살표 이용을 권장한다. 그런 다음 느낌표를 눌러 merge했음을 알린다.

# Useful Tip

## Overlay Icon 보이게 하기

레지스트리의 우선순위를 변경해주면 된다. 15개 까지만 표시해 준다고 한다.

-. regedit.exe 실행



-. 아래 위치로 이동 HKEY\_LOCAL\_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\ShellIconOverlayIdentifiers



-. Tortoise 항목들의 이름을 바꾸어 상위로 이동시킨다.(앞에 공백을 추가하면 됨)

-. 재부팅 하면 Overlay Icon이 보인다.

-. 만약 보이지 않는다면, 백업 하기 위해 내보내기를 한 후, reg 파일이 생성되면 더블 클릭하고 재부팅한다.

