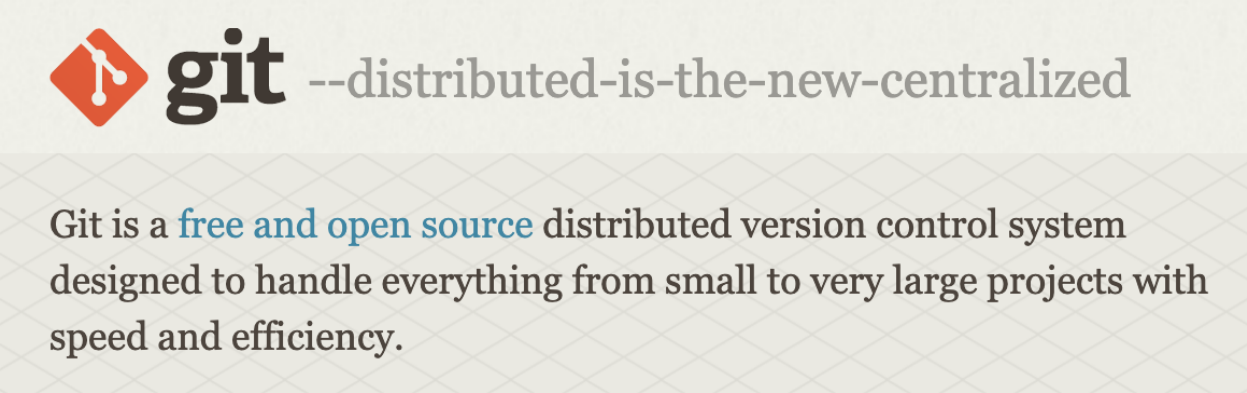
Git 이론 - 2021.10.09(토)

Git은 무엇인가?

https://git-scm.com/

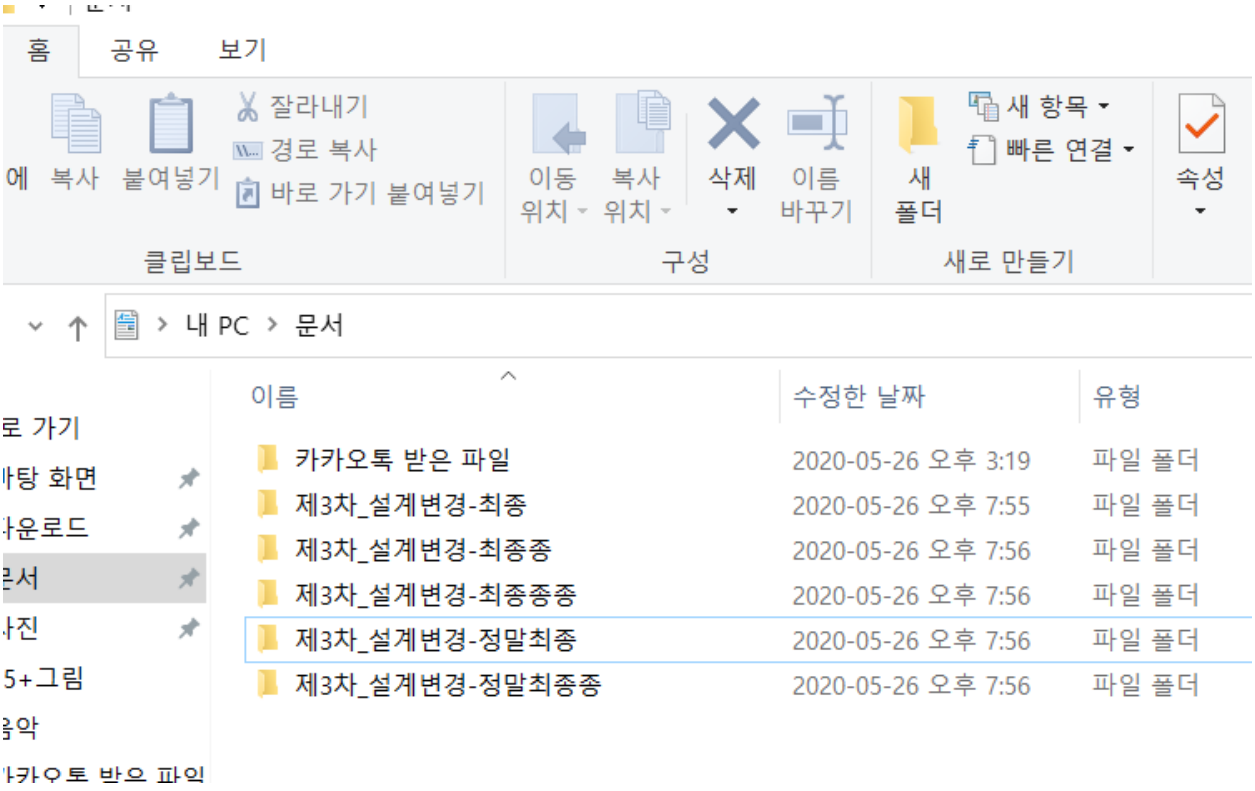


- Linux 커널의 코드 관리를 위해 개발한 프로젝트

버전관리란?

- 파일의 변화를 시간에 따라 기록한 뒤, 나중에 특정 버전으로 다시 되돌릴 수 있는 시스템
- 코드의 버전을 왜 관리할까?
  - 어떤 사유로 어떤 코드가 수정되었는지 안다면? 코드에 대한 관리/이해가 쉬워진다

버전관리의 역사



- 다들 이런 경험 있으시죠?

1. Local VCS (Local Version Control System)

- 로컬에서 파일의 변경된 부분(delta)를 저장

2. Central VCS (Central Version Control System)

- 중앙 서버에서 파일을 관리하고 클라이언트는 서버에서 파일을 받아서 사용
- 중앙 서버에 의존한다는 단점이 있다. SPOF(Single Point of Failure). 중앙서버의 자료를 주기적으로 백업해주지 않으면 복구가 어려움
- CVS, SVN(subversion)등 몇년 전까지만해도 많이 사용

3. Distributed VCS (Distributed Version Control System)

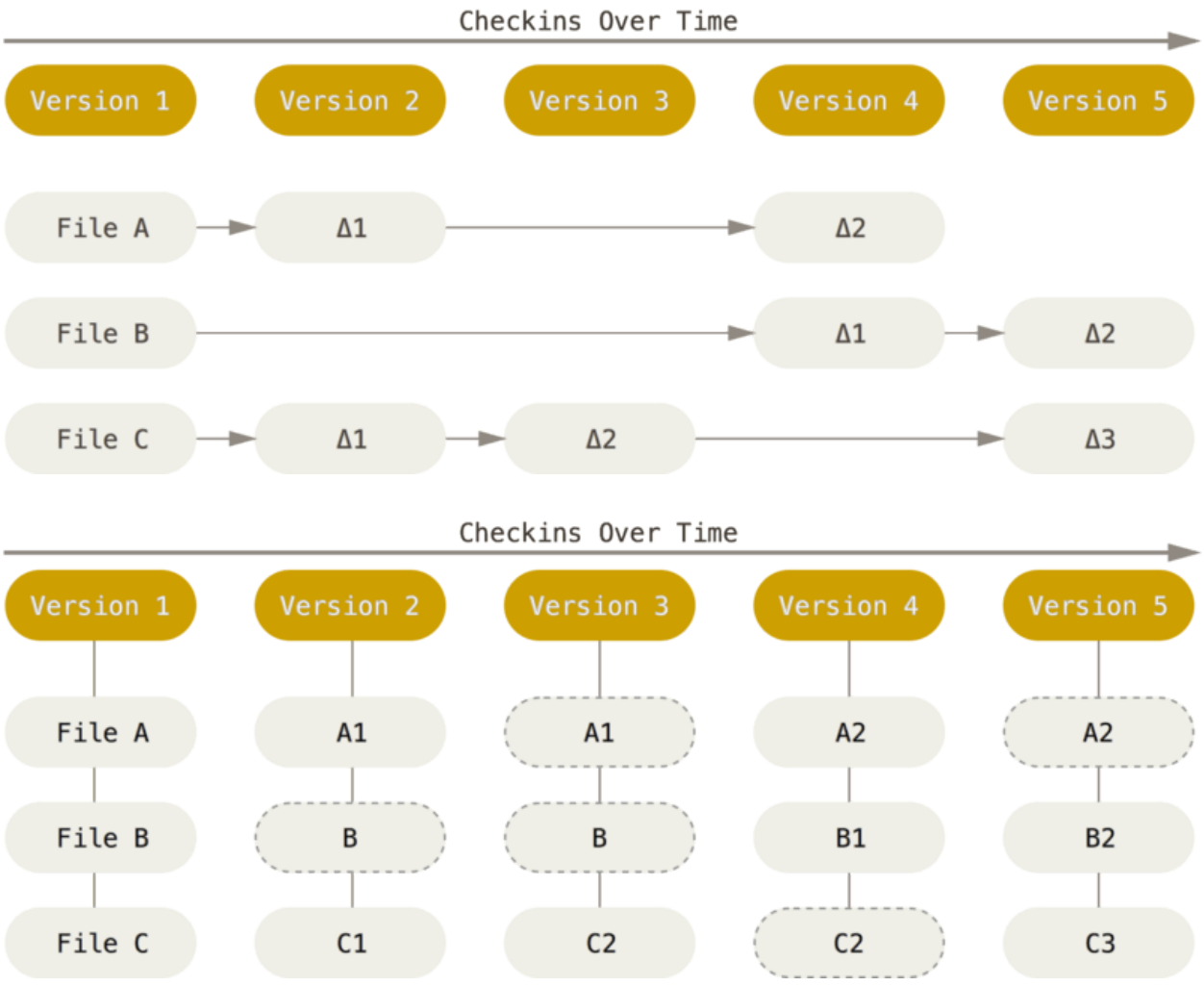
- Git이 바로 이 방식!
- 클라이언트가 서버에서 마지막 스냅샷을 받아오는게 아니고, 저장소의 모든것(히스토리 포함)을 복제

- 원격 중앙 서버가 있는게 아니고 로컬도 중앙이 될 수 있다. 어딘가에 clone(복제)해둔 코드가 있다면 코드의 복구는 식은죽 먹기!
- 로컬에서 반영한 내용이 잘못되었다도 하더라도 복구가 쉽고, 다른 사람의 코드와 충돌이 날 가능성이 적음
- CVCS보다 복잡하지만 적응하고나면 이만한 파일 관리 방법이 없음
- 빠르고 가볍고 단순하고 여러 작업자가 작업해도 문제 없다!!

어떤 방식으로든 버전관리를 하면 혼자 작업할 때도 좋습니다 👍

Git의 기본 개념과 용어

파일의 차이(delta)를 관리하는게 아니고 스냅샷(snapshot)을 관리



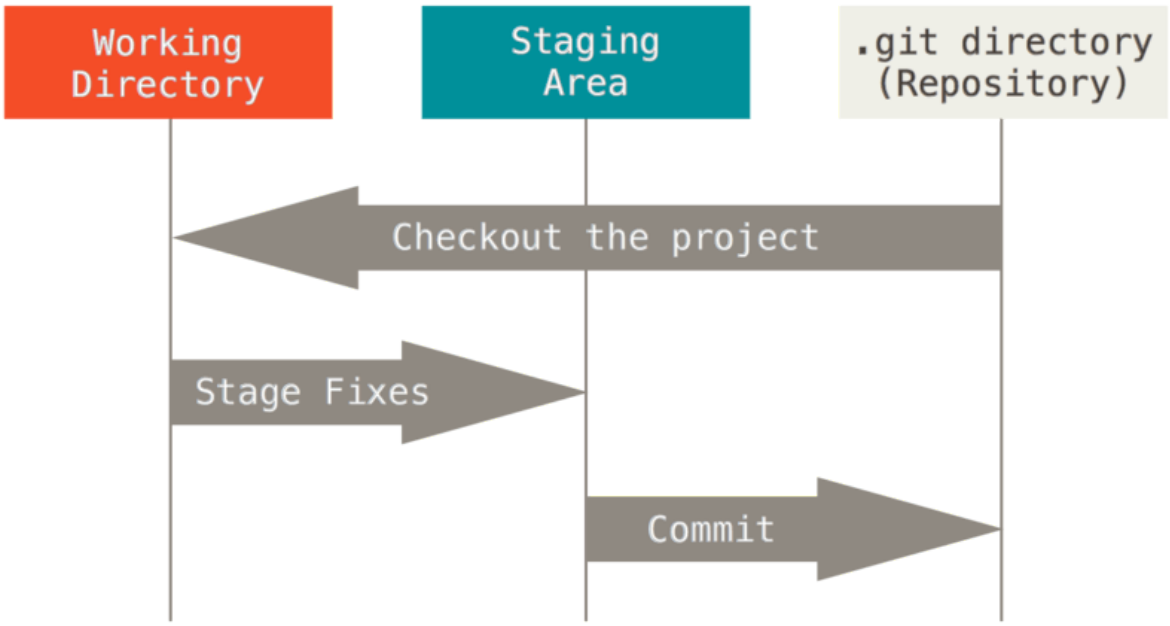
- 이전 버전과의 차이(델타)만 저장하는 것이 아니라, 모든 파일의 현재 상태를 스냅샷으로 관리한다.
- 참고 - git이 저장하는 방식

Hash(체크섬)으로 데이터를 관리

- Git은 데이터를 저장하기 전에 항상 체크섬을 구하고 그 체크섬으로 데이터를 관리
- SHA-1 해시로 체크섬을 만들고, 이 해시는 git안에서 식별자로 사용된다.
- 예) `24b9da6552252987aa493b52f8696cd6d3b00373`

파일의 상태

- Modified : 파일이 수정되었으나, 로컬 데이터베이스에 커밋되지 않음
- Staged : 수정한 파일을 커밋할 것이라고 표시해 둠
- Committed : 로컬 데이터베이스에 커밋 된 상태(스냅샷으로 저장된 상태)



프로젝트 생성하기

git 설치하기

- 설치 가이드
- https://git-scm.com/downloads
- CLI(Command Line Interface), GUI (Graphic User Interface) 모두 있지만, CLI를 권장!! (모든 기능을 가지고 있음 & 익숙해지면 편리하고 빠름)
- Linux 환경의 명령어(shell script/bash script)에 익숙해지는 것을 권장드립니다 😸

기본 설정 & 프로젝트 생성

git 설치 후 필수 설정

```
git --version

# set name and email
git config --global user.name "name"
git config --global user.email "email"# show all configs
git config --list
```

프로젝트 생성

- 프로젝트의 단위는 저장소(Repository)
- 프로젝트 생성 방법
  - github.com 에서 저장소를 만들고 초기화시킨 뒤 로컬로 clone하기
  - 로컬 디렉토리에 `git init`을 한 뒤 github.com에 푸시하기

기본 명령어 익히기

- 모든 명령어는 `-help` 를 붙이면 매뉴얼을 볼 수 있다!

init, clone

- 저장소 생성할 때 쓰는 명령어
- init : 직접 생성

```
echo "# hello-git" >> README.md
git init
git add README.md
git commit -m "first commit"
git branch -M main
git remote add origin https://github.com/creamcream1217/hello-git.git
git push -u origin main
```

- clone : remote 저장소에서 복제하기
  - https://github.com/new

`git clone <repository url>`

status

`git status`

diff

`git diff # Committed vs Modified`

`git diff --staged # Committed vs Staged`

`git diff <commit 1> <commit 2>`

`git diff HEAD HEAD^`

`git diff <branch1> <branch2>`

add

`git add <filename>`
`git add .`

`.gitignore`

reset

```
git reset HEAD <filename> # is the opposite of git add <filename>
# staged 상태인 파일을 untracked로 변경
```

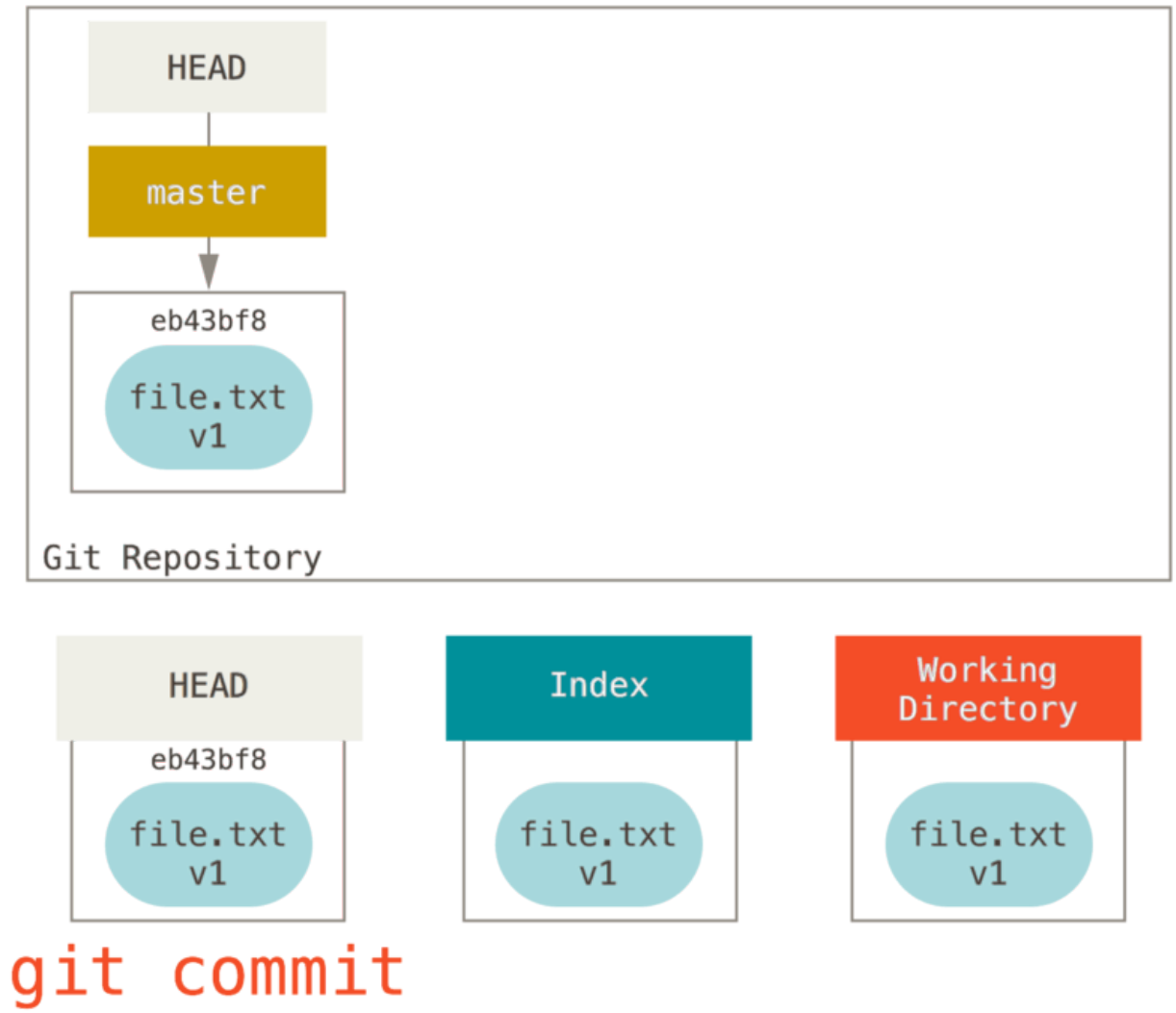
restore

```
git restore --staged <filename> # staged 상태인 파일을 untracked로 변경
```

```
git restore <filename> # working tree에서의 변경사항을 버림
git checkout -- <filename>
```

commit

`git commit`



commit --amend

```
git commit -m 'initial commit'
git add forgotten_file
git commit --amend
```

commit --fixup

```
git commit -m 'initial commit'
git add forgotten_file
git commit --fixup HEAD
```

log

`git log`

`-graph`

- 브랜치와 머지 히스토리 정보까지 아스키 그래프로 보여준다.

`-patch`

- 각 커밋의 diff 결과를 보여준다

`-pretty`

- 히스토리 내용을 보여줄 때 기본 형식 이외에 여러 가지 중에 하나를 선택할 수 있다
- `git log --pretty=format:"%h %s" --graph`

`-oneline`

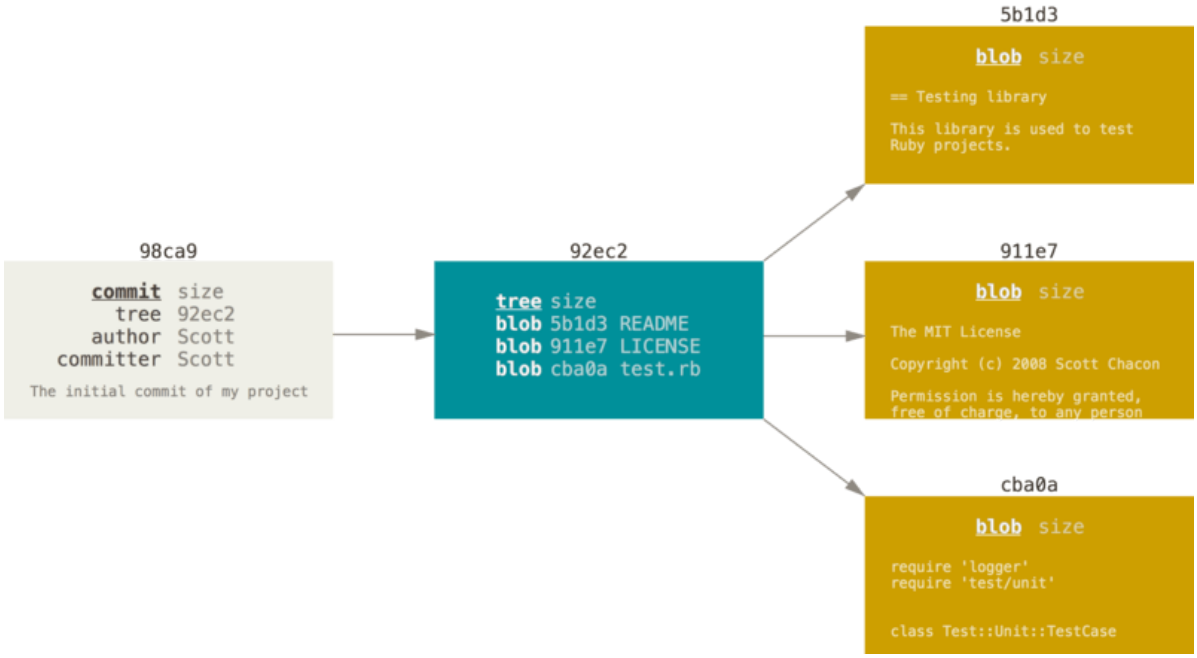
`git log --oneline --decorate --graph --all`

rm, mv

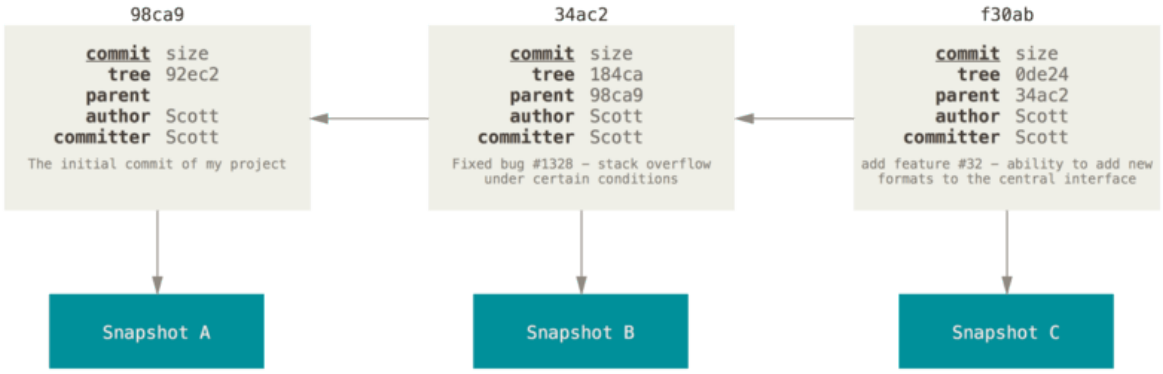
협업을 위한 명령어 익히기

커밋

- 커밋을 하면 커밋 객체가 생성된다. 내부적으로는 아래와 같은 모습



- 커밋을 할 때 마다, 이전 커밋에 대한 포인터도 저장한다.



HEAD

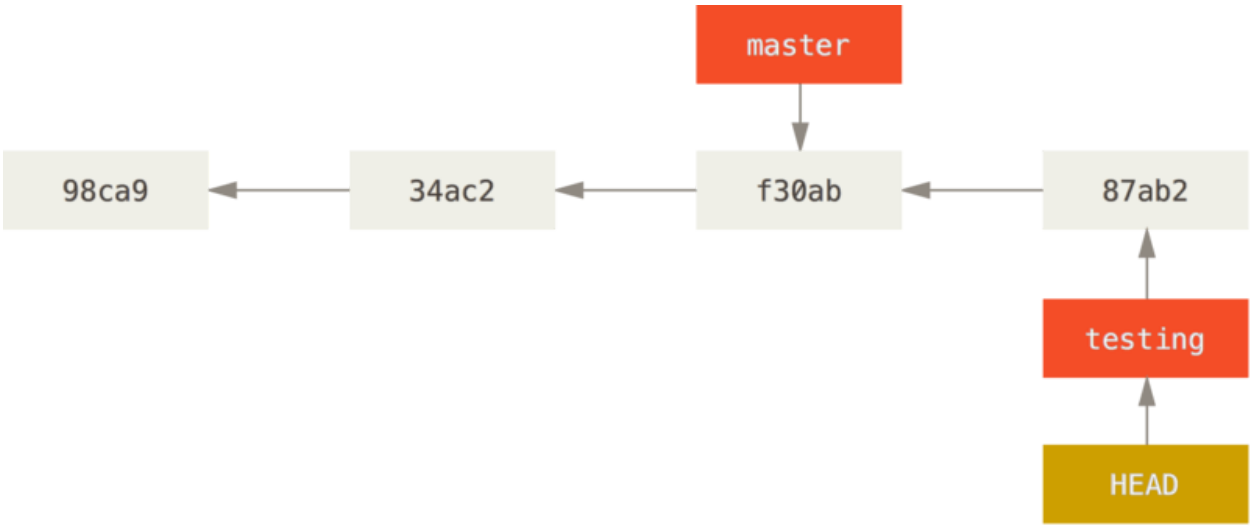
- HEAD는 현재 체크아웃된 커밋을 가리킨다 = 현재 작업중인 커밋
- HEAD는 항상 작업트리의 가장 최근 커밋을 가리킨다.
- 작업트리에 변화를 주는 git 명령어들은 대부분 HEAD를 변경하는 것!

branch, checkout

- 기존의 코드와 관계없는 개별적인 작업장
  - 코드를 여러가지의 브랜치로 나누어서 관리하면 개발/협업하기 수월

- 사실은 커밋 사이를 이동할 수 있는 **포인터**에 가까움
- `branch` 명령으로 새로운 브랜치를 생성 & `checkout`으로 해당 브랜치로 **HEAD**를 이동

```
git branch new-branch
git checkout new-branch
touch a.txt
git commit -am 'Add a file'
```

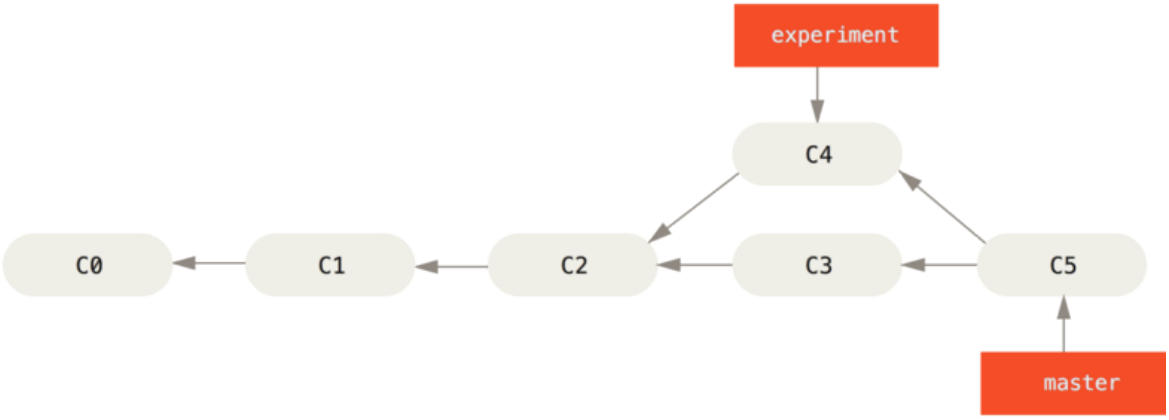


checkout

- checkout(HEAD 지정)은 브랜치 뿐만 아니라 커밋에도 가능하다

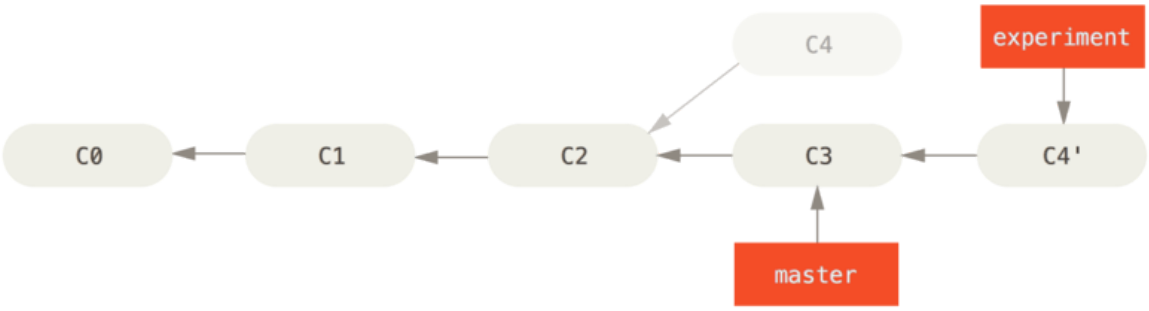
merge

- 브랜치끼리 합치는 것
- 추가한 브랜치에서 작업이 완료되면 다시 메인 브랜치로 합칠 때 사용한다.
- merge commit이 생성됨



rebase

- merge와 비슷한 기능. 하지만 커밋 그래프의 모양이 다르다
- 커밋 히스토리가 깔끔해지길 원한다면 이 방법을 사용하자.



- ⚠️ rebase를 하면 기존의 커밋을 사용하는게 아니라 새로운 커밋이 생성됨!! 즉, 공유하는 코드는 함부로 rebase를 하면 안된다.

충돌 해결하기

- 같은 내용을 여러 브랜치에서 변경한 뒤, 머지하는 경우에 발생

remote, fetch, pull, push

- git 저장소는 원격서버(remote)와 로컬(local)에 둘 다 존재
- 원격 저장소의 정보는 git remote 명령으로 등록하고 확인 할 수 있다.

`push`

- 로컬 브랜치의 내용을 원격 서버로 전송

`git push origin feature-add-menu`

`fetch`

- 서버에 존재하지만 로컬에 없는 데이터를 받아와 저장함.

`git fetch origin`

- 워킹 디렉토리의 파일 내용은 변경되지 않고, git merge를 해야 합쳐진다.

`pull`

- `fetch` + `merge`

현업에서 브랜치를 사용하는 방법, 코드 배포

- 우형에서 사용하는 git-flow

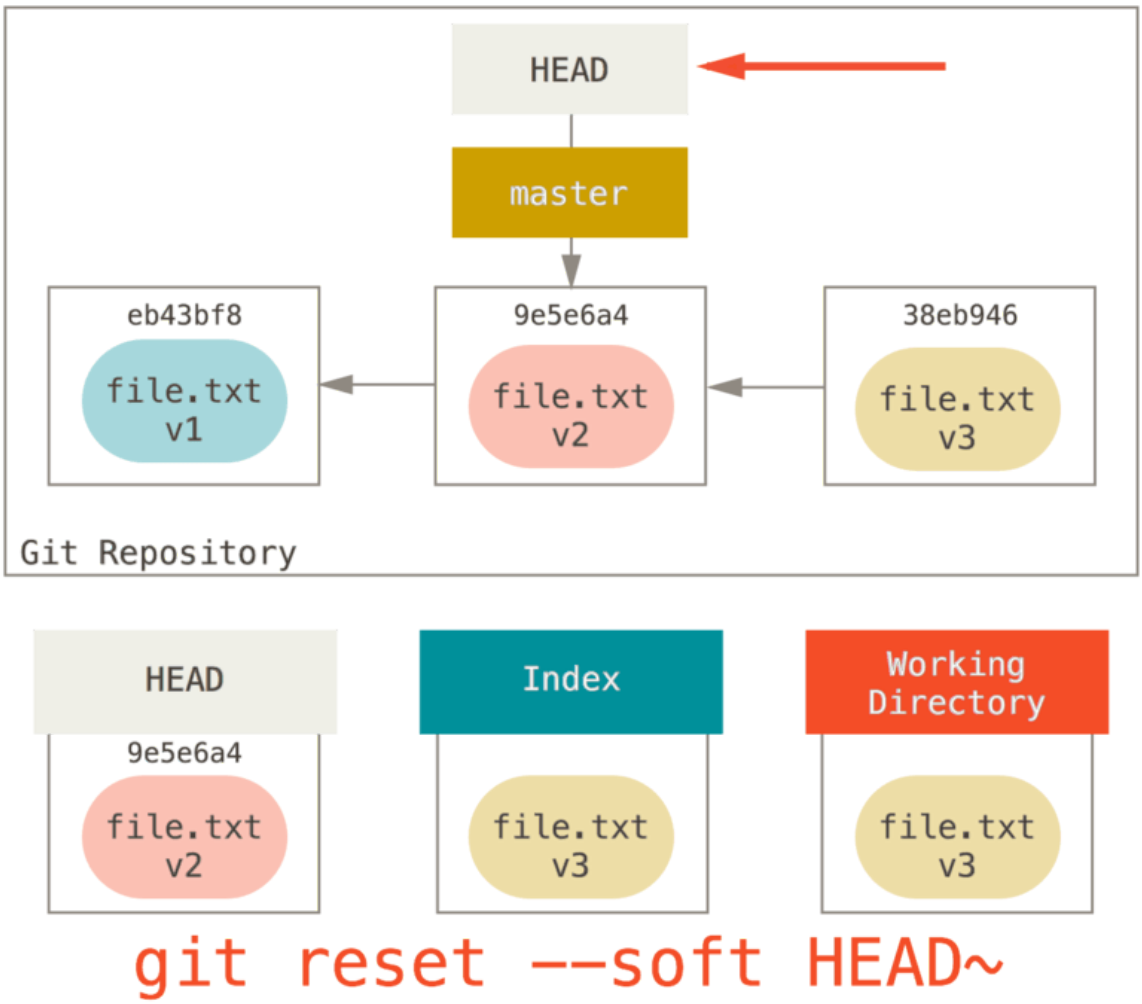
커밋 수정하기

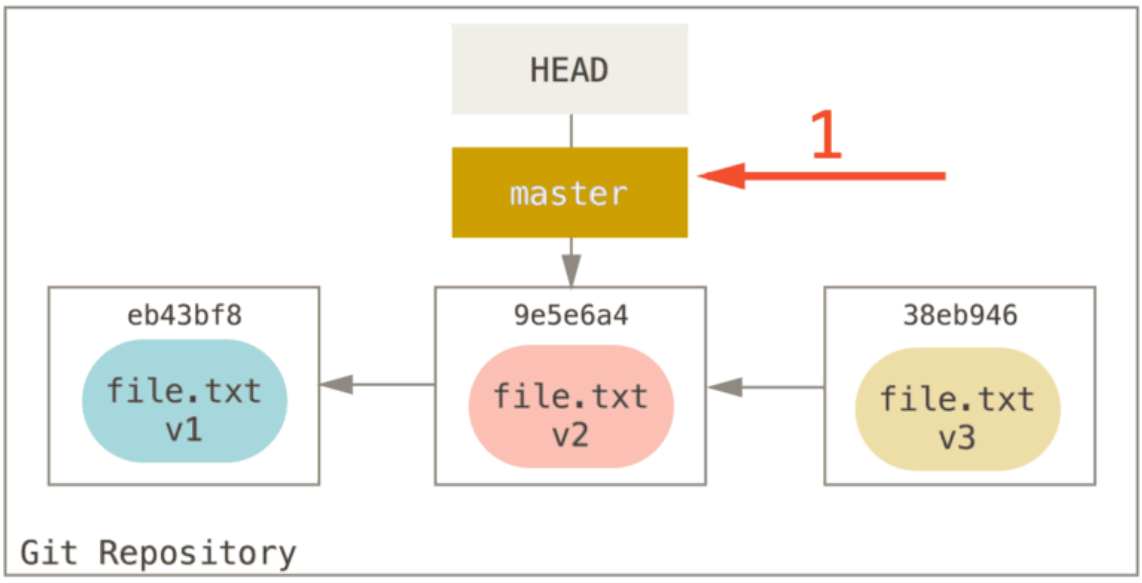
커밋과 관련된 팁

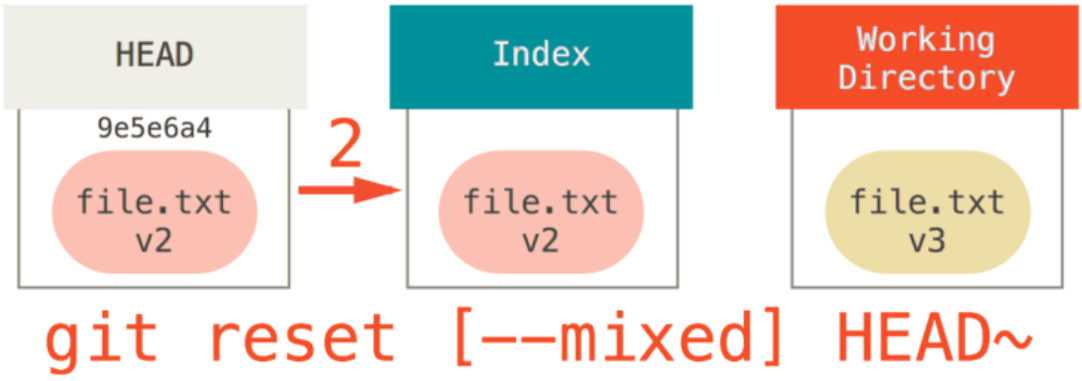
- SHA-1 해시값으로 되어있는 커밋, 전체가 아닌 일부만 사용해도 된다 `commit 8984de38057a0cd157f978ff16d8a56e53fc9d53` = `8984d`
- 한번에 한 커밋 위로 움직이는 `^` : `git checkout main^`
- 한번에 여러 커밋 위로 움직이는 `~<num>` : `git checkout main~1`
- `git branch -f main HEAD~3`

reset

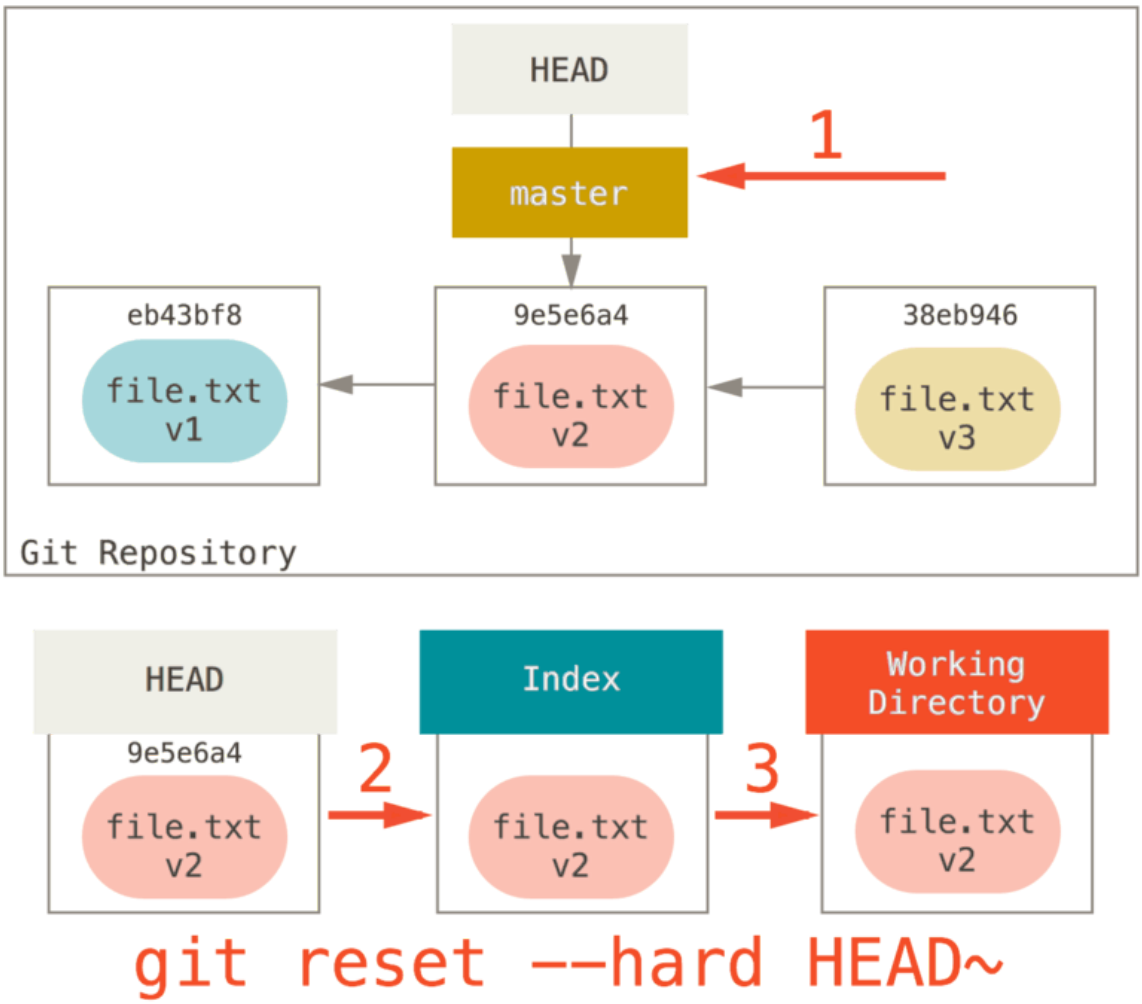
- 브랜치가 예전 커밋을 가리키도록 옮김. 커밋을 안했던 것 처럼 바꾸기 때문에 이전 히스토리가 바뀐다. RESET 개념 이해하기


HEAD
master
eb43bf8
file.txt
v1
9e5e6a4
file.txt
v2
38eb946
file.txt
v3
Git Repository
HEAD
9e5e6a4
file.txt
v2
Index
file.txt
v3
Working
Directory
file.txt
v3
git reset --soft HEAD~






```
git reset [--mixed] HEAD~
```



revert

- reset 처럼 히스토리를 바꾸는 것이 아니고, revert 시키고 싶은 대상 커밋의 반대되는 내용이 새로운 커밋으로 생성된다.

cherry-pick

- 커밋 정리가 점점 복잡해 질 때 아주 유용한 명령어. merge하지 않을 브랜치에서 원하는 커밋만 쏙쏙 뽑아온다
- HEAD 아래에 다른 커밋들에 대한 복사본을 만듦
- `git cherry-pick <commit1> <commit2> ....`

rebase -i (interactive rebase)

- HEAD로 부터 몇개의 커밋 범위를 정하여 rebase 할 수 있다. cherry-pick은 원하는 커밋의 커밋아이디를 알아야 가능하지만 rebase -i 는 편집기에서 커밋을 선택할 수 있기 때문에 범위만 알면 수정이 가능.

```
# Commands:
# p, pick <commit> = use commit
# r, reword <commit> = use commit, but edit the commit message
# e, edit <commit> = use commit, but stop for amending
# s, squash <commit> = use commit, but meld into previous commit
# f, fixup <commit> = like "squash", but discard this commit's log message
# x, exec <command> = run command (the rest of the line) using shell
# b, break = stop here (continue rebase later with 'git rebase --continue')
# d, drop <commit> = remove commit
# l, label <label> = label current HEAD with a name
# t, reset <label> = reset HEAD to a label
# m, merge [-C <commit> | -c <commit>] <label> [# <oneline>]
```

stash

- 워킹 디렉토리의 내용을 어딘가에 저장해두는 것

rebase와 관련된 주의사항!!

- 남들과 공유하는 커밋은 가급적 rebase하지 말자.
- git pull 할 때, `-rebase` 옵션을 사용해 rebase 를 하고 머지하자.

기타

그 외 명령어와 사용 팁

- alias, tag 등

현업에서 github를 사용하는 방식

- **Issue**, **Pull Request**

협업과 커밋 메시지 컨벤션

- https://overcome-the-limits.tistory.com/entry/협업-협업을-위한-기본적인-git-커밋컨벤션-설정하기
- 커밋 템플릿 만들기
  - `.gitcommit-template.txt`
  - `git config --global commit.template <경로>`

```
# <타입> : <제목> 형식으로 작성하며 제목은 최대 50글자 정도로만 입력
# 제목을 아랫줄에 작성, 제목 끝에 마침표 금지, 무엇을 했는지 명확하게 작성

################
# 본문(추가 설명)을 아랫줄에 작성

################
# 꼬릿말(footer)을 아랫줄에 작성 (관련된 이슈 번호 등 추가)

################
# feature : 새로운 기능 추가
# fix : 버그 수정
# docs : 문서 수정
# test : 테스트 코드 추가
# refactor : 코드 리팩토링
# style : 코드 의미에 영향을 주지 않는 변경사항
# chore : 빌드 부분 혹은 패키지 매니저 수정사항
################`
```

오픈소스 프로젝트로 git사용법 훔쳐보기

- https://github.com

github 활용하기

- https://pages.github.com/