1016 Git

— 배운 내용 : Commit, Work Tree, Index, git staus, stage 와 저장소, git log, 태그 관리, 파일 상태 관리, 버전 비교(git diff), commit 되돌리기(reset, revert), 작업 임시 저장(git stash), Branch(생성, 이동, 병합, 충돌, 삭제, 재배치)

- 중요 내용 : Commit, status, stage, git log, 태그 관리, 파일 상태 관리, 버전 비교(git diff), commit 되돌리기(reset, revert), Branch(충돌)

⇒ git add 명령 수행 시 warning 이 발생하는 경우

- 운영 체제마다 줄 바꿈 문자를 인식하는 방법이 달라서 발생하는데 이 경우에는 설정이 필요

```
git config core.autocrlf true
# add 명령을 수행할 때 에러가 발생하지 않는다면 필요 없음
```

**** Git**

1. 개요

⇒ 컴퓨터 파일의 변경 사항을 추적하고 여러 명의 사용자들 간에 해당 파일들의 작업을 조율하기 위한 분산 버전 관리 시스템 혹은 명령어

⇒ 토발즈가 리눅스 커널에 대한 코드를 공유할 목적으로 개발

= 누구나 사용할 수 있는 자유 소프트웨어

1) 장점

⇒ 이력 기록 및 추적이 가능

⇒ 전 세계의 수많은 사용자가 이용 중

⇒ 서버의 역할을 하는 원격 저장소와 각 개발자의 로컬 저장소에 git 은 소스 코드와 변경 이력을 분산 저장하기 때문에 원격 저장소에 문제가 발생해도 로컬 저장소를 이용해서 복원이 가능함

⇒ 변경 이력을 병합하는 것이 가능

2) Git Hub

⇒ Rubt on Rails 로 작성된 분산 버전 관리 툴인 git 의 원격 저장소 호스팅 서비스

⇒ 영리적인 서비스와 오픈 소스를 위한 무상 서비스를 모두 제공함

⇒ git 이 일반적으로 텍스트 명령어 방식인데 git hub 는 GUI 를 제공함

⇒ git hub 는 git 프로젝트 저장소의 역할 이외에도 다양한 기능을 제공함

- git action 과 git hub deployment API 등을 제공함
- project boards 를 제공하므로 협업도 가능

3) 원격 저장소와 로컬 저장소

⇒ 로컬 저장소

- 내 pc 안에 파일이 저장되는 개인 저장소
- 저장소를 새로 직접 만들거나 원격 저장소를 로컬 저장소로 복사해서 생성함

⇒ 원격 저장소

- 파일 원격 저장소 전용 서버에서 관리되며 여러 사람이 함께 공유하기 위한 저장소

4) Commit 용어

⇒ 파일 및 디렉토리의 추가나 변경 사항을 로컬 저장소에 기록하는 것

⇒ commit 을 하게 되면 이전의 commit 상태에서부터 현재 상태까지의 변경 이력이 기록된 Revision(리비전) 이 생성됨

⇒ commit 은 순서대로 저장되므로 최근 commit 부터 거슬러 올라가면 과거의 변경 이력과 관련된 내용을 알 수 있음

⇒ 각 리비전은 영문과 숫자로 구성된 40자리의 고유한 이름이 붙는데 이 이름을 보고 각 리비전을 구분하고 선택하게 됨

⇒ Commit 은 이력을 남기는 중요 작업이므로 commit 을 수행할 때는 메시지를 필수로 입력해야 하는데 권장 사항은 변경 내용을 요약하고 빈 줄을 준 다음 변경한 이유를 작성하는 것

5) Work Tree 와 Index

⇒ Work Tree : 작업 중인 디렉토리

⇒ Index : Commit 을 실행하기 전의 저장소와 work tree 사이에 존재하는 공간

- 저장소 ↔ 인덱스 ↔ 작업 트리
- 작업 트리의 내용을 인덱스에 저장하는 명령이 add 이고 인덱스의 내용은 commit 을 통해 저장소에 저장됨
- 작업 트리 내에서 어떤 파일과 디렉토리만 버전 관리를 할지 선택할 수 있는데 이 선택된 내용이 Index 에 기록이 됨
- index 에 기록되지 않은 파일은 저장소에 기록이 안 됨

⇒ git 의 commit 작업은 작업 트리에 있는 변경 내용을 저장소에 바로 기록하는 것이 아니라 그 사이의 공간인 인덱스에 파일의 상태를 기록(staging)하게 되어 있기 때문에 저장소에 변경 사항을 기록하기 위해서는 기록하고자 하는 모든 변경 사항이 인덱스에 존재해야 함

2. Git 설치 및 확인

1) 설치

⇒ windows : https://git-scm.com/downloads 에서 다운 받아 설치

⇒ Mac : https://git-scm.com/downloads 에서 다운 받아 설치하거나 brew install git 명령어를 사용해서 설치

⇒ linux : https://git-scm.com/downloads 에 설치 방법 메뉴얼을 제공

2) 설치 확인

⇒ git version 혹은 git --version

3) git 의 기본 브랜치 이름을 main 으로 변경

⇒ git 의 기본 브랜치 이름은 matser 로 설정되어 있는데 git hub 의 기본 브랜치 이름은 main 으로 설정되어 있음

- 이름을 변경하지 않으면 git push 작업이 되지 않을 수 있음

```
# 기본 브랜치 이름을 main 으로 설정(변경)
git config --global init.defaultBranch main
```

4) 로컬 저장소에 git hub 사용자 등록

⇒ 사용자 등록하기

```
# 사용자 등록하기
git config --global user.name 유저 이름
git config --global user.email 이메일

# 유저 이름과 이메일 확인
git config --global user.name
git config --global user.email
```

⇒ --global 을 제외한 명령어를 사용하면 현재 프로젝트에 대해서만 적용됨

3. 버전 만들기

1) git init

⇒ 로컬 저장소를 만드는 명령

⇒ 성공하면 메세지가 출력되고 현재 디렉토리에 .git 이라는 디렉토리가 생성됨

- . 이 이름 앞에 붙으면 리눅스나 유닉스에서 숨김 디렉토리로 설정되지만 windows 에서는 아님
- 성공하면 출력되는 메세지는 git initialized ~~

2) git status

⇒ 현재 상태를 확인하는 명령

- 막 시작했다면 commit 된 것도 없고 아무런 변화도 없다고 출력

⇒ 새로운 파일을 생성해서 내용을 추가하고 다시 git status 를 사용해서 변화 확인

- untracked files 라는 메세지와 함께 파일이 출력
- 변화가 발행했는데 아직 추적하고 있지 않은 파일이라는 메세지임 - 아직 인덱스 영역에 포함되지 않은 파일로 여기에서 commit 을 해도 저장소에 반영되지 않음

3) git add

⇒ 스테이지에 올리는 명령

⇒ 이제 파일의 변경 내용을 추적하라는 의미

⇒ 형식

```
git add 파일경로

# 현재 디렉토리의 모든 파일을 추가할 때는 경로 대신 . 사용
# -> git add .
```

⇒ git add 를 통해 파일을 스테이지에 추가하고 git status 를 다시 사용해서 상태 확인

- 파일에 변화가 발생했다고 Changes to be committed 메세지가 출력됨

4) git commit

⇒ 현재까지의 추적 내용을 로컬 저장소에 반영하는 명령

⇒ 기본 형식

```
git commit -m “메세지”
git commit --message "메세지"
```

⇒ 명령을 성공적으로 수행하면 새로운 버전을 만들어냄

⇒ commit 을 성공적으로 수행하면 수행 결과가 메세지로 출력되며 이후 git status 를 수행해보면 변경된 내용이 없다고 메세지가 출력됨

5) git log

⇒ 로컬 저장소의 commit 목록을 출력하는 명령

⇒ commit hash, 만든 사람, 날짜, 메세지가 출력됨

- HEAD 는 현재 사용중인 브랜치를 의미

6) git commit -am “메세지”

⇒ git add . 과 git commit -m 명령을 합쳐서 수행하는 명령

⇒ 한번에 스테이징 영역에 올리고 commit

```
git commit -am "메세지"
```

- 이후 git status 를 통해 확인하면 변경 사항이 없다고 출력

⇒ 현재 상태 확인 : git log

- 두번째로 -am 옵션을 통해 commit 한 부분을 main 브랜치가 가리키고 있는 상태

⇒ git commit -am 명령은 한번이라도 commit 이 된 파일에 대해서만 사용이 가능하고 아직 commit 된 적 없는 파일에 대해서는 수행 불가능

- 새로운 파일을 추가한 다음 디렉토리에 git commit -am 명령을 사용하면 기존의 파일들은 처리가 되지만 새로운 파일에 대해서는 commit 을 수행하지 못함
- git log 를 사용해보면 commit 이 발생했다고 출력
- git status 를 통해 확인해보면 새로운 파일은 아직 추적하지 않는다고 메세지가 출력됨

⇒ 이런 이유들로 인해 commit 을 한 다음에는 log 와 status 를 사용해서 확인하는 과정이 필요함

7) commit 메세지 본문 작성

⇒ 메세지는 제목과 본문으로 구성

⇒ git commit -m 을 통해 작성한 메세지는 본문이 아닌 제목만 작성한 것

⇒ 메세지를 작성하고자 하는 경우는 git commit 명령을 수행하면 됨

⇒ 현재 디렉토리의 모든 파일을 git add . 을 통해 스테이지에 추가하고 git commit 명령을 수행하면 editor 가 수행됨

- 화면에서 첫번째 줄이 제목이 되고 그 아래는 내용이 됨
- 에디터 사용 방법은 리눅스와 동일함 - i 키, esc, :wq 등

8) git log

⇒ commit 을 조회하는 명령어

⇒ 단순한 형태로 조회 - 줄 단위로 조회하기

```
## 1 줄 단위로 출력함
git log --oneline
```

⇒ 자세히 조회하기

```
# 기존의 git log 보다 자세하게 조회
git log --patch

git log -p
```

⇒ 그래프 형태로 조회

```
# 로그 옆에 선이 그어지는데 이 선으로 그래프를 표현
git log --graph
```

- 브랜치가 여러개 만들어지고 여러번 분기를 했을 때는 그래프가 트리 형태로 출력됨

⇒ 모든 브랜치에 대해 commit 명령을 조회

```
git commit --branches
```

9) 태그 관리

⇒ 태그는 commit 에 붙일 수 있는 서브 타이틀

⇒ 태그를 이용해서 버전을 표시하는 경우가 많음

⇒ git tag <태그> 를 수행하면 현재 브랜치에 태그가 추가됨

⇒ 특정 commit 에 추가하고자 하는 경우에는 git tag <태그> <커밋>

⇒ 현재 commit 에 ver:1.0 추가하기

```
# 태그를 부여 - 현재 commit
git tag ver:1.0

# 태그 확인
git log --oneline
```

⇒ 다른 commit 에 태그 추가하기

```
# 다른 commit 에 태그를 추가
# <커밋> 자리에는 commit log 를 하면 출력되는 코드를 사용 - 예를 들면 334358c
git tag ver:0.9 <커밋>
```

⇒ 태그를 조회

```
git tag --list

git tag -l
```

⇒ 태그 삭제

```
# <태그> 위치에는 태그 내용을 그대로 입력
git tag --delete <태그>

git tag -d <태그>

# 예시
git tag -d ver:0.9
```

10) 파일 상태 확인

⇒ git 의 작업 트리

- git 은 관리하는 프로젝트의 작업을 효율적으로 처리하는 '작업 트리' 라는 개념을 이용함
- 작업 트리는 git 이 추적하는 파일과 추적하지 않는 파일을 구분하고 추적하는 파일들의 상태를 구분짓는 영역
- 이 내용은 .git 이라는 숨김 디렉토리에서 관리
- 그래서 .git 파일을 삭제하면 git 과 관련된 모든 내용이 소멸하며 git 과의 연결이 끊어짐

⇒ 작업 트리 영역은 3개 영역으로 구성

- Working Directory : 실제 작업 중인 파일들이 존재하는 영역으로 파일을 생성하거나 기존의 파일을 수정한다면 이는 working directory 에서 작업 중인 것
- Staging Area : 작업 디렉토리의 파일 중 git 이 추적하는 파일을 식별하는 영역으로 .git 디렉토리 내부의 index 파일에서 추적하는 파일을 식별함
- Local Repository : 스테이징에서 추적하는 파일이 commit 으로 등록되는 영역으로 스테이징 영역의 파일들이 하나의 변경 단위인 commit 으로 등록되는 곳
- 작업 디렉토리에서 스테이징 영역으로 등록하는 명령이 git add 이고 스테이징 영역에서 로컬 저장소로 이동하는 명령이 git commit

⇒ 파일의 상태

- untracked 상태 : 현재 작업 디렉토리에는 파일이 존재하지만 git 이 추적하지 않는 상태

- tracked 상태 : git 이 추적 중인 상태로 스테이징 영역과 로컬 저장소에 있는 파일이 여기에 해당

⇒ Unmodified 상태와 Modified 상태

- Unmodified 상태 : 스테이징 영역에 존재하지만 수정하지 않은 상태
- Modified 상태 : 스테이징 영역에 존재하고 수정한 상태

⇒ commit 메세지 수정

- 가장 최근의 commit 메세지 수정

```
git commit --amend -m "메세지"
```

⇒ 유저 이름과 이메일도 수정이 가능

```
# 이메일을 수정할 때는 꺽쇠(< 와 >) 가 필요함
git commit --amend --author "작성자 <이메일>"
```

4. 버전 비교

1) git diff

⇒ 최근 commit 과 작업 디렉토리를 비교하는 명령

⇒ commit 한 이후에 작업 디렉토리에서 어떤 작업이 이루어졌는지 확인하기 위해서 사용

- 파일에 추가한 내용, 지운 내용 등의 변경 사항이 출력됨

2) git diff --staged

⇒ 최근 commit 과 stage(git add) 를 비교하는 명령

- stage 는 add 명령을 사용해야 변경이 발생함
- 결과는 add 를 하기 전의 get diff 와 동일함

3) git diff <커밋> <커밋>

⇒ commit 끼리의 변경 내용을 확인하는 명령

⇒ commit 은 40자리 해시를 전부 이용해도 되고 짧은 해시를 이용해도 됨

⇒ commit 은 순서대로 적용되기 때문에 2개의 순선에 따라 결과가 반대로 출력됨

- commit 의 위치에 따라 변경 내용에 대한 출력도 달라짐

⇒ 해시 코드를 사용하는 대신에 현재 commit 을 가리키는 HEAD 를 이용하는 방법도 가능함

- 바로 이전의 comit 은 HEAD^ 또는 HEAD~1 로 표기하고 그 이전의 commit 은 HEAD^^ 또는 HEAD~2 로 표기하는 것이 가능함

```
# 현재 HEAD 가 가리키는 commit 과 바로 이전에 가리킨 commit 을 비교
git diff HEAD HEAD~1

# 현재 commit 과 2개 앞의 commit 을 비교
git diff HEAD HEAD^^
```

5. Commit 되돌리기

⇒ commit 을 되돌리는 방법은 2가지 - reset 과 revert

1) Reset

⇒ 되돌아갈 버전의 시점으로 완전히 되돌아 가는 방식

⇒ 종류가 3가지 - soft, mixed, hard

- soft reset 은 commit 했다는 사실만 되돌림
- mixed reset 은 commit 과 스테이지를 되돌리는 것
- hard reset 은 디렉토리의 변경 사항까지 되돌리는 것

- 예시

```
A -> B -> C 순서로 commit 을 3번 실행한 경우
A -> 스테이지 -> commit -> B -> 스테이지 -> commit -> C -> 스테이지 -> commit

# soft reset - commit 만 되돌림
# 두번째 B 버전으로 soft reset 을 하는 경우 commit 을 했다는 사실만 되돌리므로
# A -> 스테이지 -> commit -> B -> 스테이지 -> commit -> C -> 스테이지
# reset 명령을 사용하면서 --soft 옵션을 사용
git reset --soft commit코드

# mixed reset - commit, 스테이지를 되돌리지만 작업은 남아 있음
# A -> 스테이지 -> commit -> B -> 스테이지 -> commit -> C 변경
# reset 의 기본이므로 옵션 없이 reset 만 사용
git reset commit코드

# hard reset - B commit 이후의 작업 내용도 되돌림
# A -> 스테이지 -> commit -> B -> 스테이지 -> commit
# reset 명령을 사용하면서 --hard 옵션을 사용
git reset --hart commit코드
```

- commit 을 한 후 reset 을 하고 git log 를 통해 변화를 확인
  - soft reset 의 경우 commit 기록은 되돌아가지만 기존 파일의 변경 사항은 남아있고 스테이지의 변경 사항도 남아 있음 - git diff --staged 를 통해 스테이지 변경 확인
  - mixed reset 의 경우 파일의 변경 내용은 남아있고 git status 를 통해서 스테이지에 추가되었는지를 확인할 수 있음 - 스테이지에 추가되지 않았고 따라서 commit 도 없음
  - hard reset 의 경우 파일의 변경 내용도 남아 있지 않음 - 작업 디렉토리의 변경 내용도 reset

2) Revert

⇒ 버전을 되돌리지만 되돌아간 상태에 대한 새로운 버전을 만드는 방식

- 예시

```
A -> B -> C 순서대로 commit 을 한 상황
- commit 은 총 3번 이루어짐

2번째인 B 로 revert 하게 되면 3번째인 C 가 사라지는게 아니라
2번째 버전의 상태를 가진 4번째 버전이 만들어짐
```

⇒ git revert <취소할 commit>

- 명령을 수행하면 editor 가 실행됨
- 취소할 commit 을 지정하는 revert 는 현재 HEAD 가 가리키는 commit 을 지정할 수 있지만 reset 은 되돌아갈 commit 을 지정하는 방식이므로 현재 가리키는 commit 을 사용할 수 없음

⇒ commit 을 취소하고 git log 를 사용해서 확인

- 3번째 commit 에 대해 revert 를 수행하면 그 이전인 2번째 commit 상태가 되지만 git log 를 통해 확인해보면 commit 은 오히려 1개가 늘어나있음
- 변경 내용이 반영된 새로운 commit 이 생성됨

6. 작업 임시 저장

1) git stash : 변경 사항을 임시로 저장

⇒ stash 명령은 tracked 상태의 파일에 대해서만 사용할 수 있음

⇒ git stash

```
git stash -m “메세지”
```

- git add 와 git commit 이 수행된 파일에 대해 내용을 수정하고 작업 내역을 임시 저장 영역에 저장
- 이후 내용을 수정한 파일을 확인하면 수정한 내용이 사라짐 - 이전 commit 상태로 이동
- 다시 한 번 파일의 내용을 수행하고 git stash 를 통해 작업 내역을 임시 저장
- 이번에도 수정 내용이 사라지고 이전의 commit 상태로 되돌아감

2) 임시 저장한 목록 확인

⇒ git stash list

- git stash 를 통해 임시 저장한 내용을 확인할 수 있음
  - 위의 예시의 경우 2가지 내용이 저장되어 있음
- 임시 저장한 내역은 가장 최근의 내역이 0번이고 이후 번호대로 인덱싱

3) 임시 저장된 작업 적용

⇒ git stash apply <스테이시>

- 뒤의 <스테이시> 부분을 생략하면 가장 최근의 stash 가 적용됨

⇒ 0번 인덱스(stash@{0}) 적용하기

- apply 를 적용하면 원본 파일에 임시 저장한 내용이 반영되어 파일이 수정됨
- commit 이 적용된 상태는 아니고 임시 저장만 반영된 상태

```
# 0번 인덱스의 stash 적용하기
# 0 대신에 다른 숫자를 입력하면 인덱스 변경 가능
git stash apply stash@{0}

# stash@{1} 적용하기
```

```
git stash apply stash@{1}
# 적용 실패
```

- 0번 인덱스이므로 가장 최근에 작업했던 임시 저장이 반영됨
- stash 1번의 경우 stash 0 의 작업 보다 나중의 작업이므로 이미 stash 0 의 작업 안에 stash 1 의 작업 내용이 포함되어 있으므로 1번에 대한 임시 저장 내용을 적용할 수 없음

4) 임시 저장 작업 삭제

⇒ git stash drop <스테이시>

- 임시 저장을 1개씩 삭제

⇒ git stash clear 를 사용하면 모든 작업이 삭제됨

- 한 번에 전부 삭제
- git stash list 를 통해 전부 삭제되었는지 확인 가능

7. Branch

⇒ 버전을 여러 흐름으로 나누어 관리하는 것

1) 브랜치를 만들어서 여러 흐름으로 나누어 관리해야 하는 이유

⇒ 예시 - 어느 정도 만들어진 쇼핑몰 코드를 수정해야 하는 경우

- 1명의 개발자가 장바구니 기능을 추가하고 다른 개발자가 주문 목록 기능을 추가하려 하는 경우
- 이런 경우 서로간의 작업 내용은 각자의 작업과 전혀 관련 없는 부분이 있을 것이고 때로는 동일한 코드를 다르게 수정해서 사용하는 부분도 있을 것임

- 브랜치가 없다면 이 경우 코드를 일일이 대조하고 합칠 코드를 판단하는 작업은 매우 번거로운 작업이며 수작업으로 수행한다면 실수가 발생할 가능성이 높음

⇒ 이런 경우 동일한 코드를 가지고 다른 브랜치를 만들어서 각각 작업을 하도록 한 후 이를 merge 하게 되면 코드를 일일이 살펴봐야 하는 번거로움이 사라짐

- 이러한 경우에도 같은 코드를 다르게 수정하는 부분은 확인을 해야 함
- 브랜치를 나누어서 작업할 때 주의할 점은 동일한 코드를 다르게 수정하는 '충돌' 의 부분
- 이렇게 되면 어떤 작업을 적용할 지 나중에 결정을 해야 함
    - 충돌이 발생하면 적용된 부분을 제외하고 일부 수정 사항은 삭제해야 할 수 있음

2) 브랜치 실습을 위한 디렉토리 생성

⇒ 실습용 디렉토리 생성 및 파일 생성

- C:\Users\USER\Desktop\새 폴더 (2)\branch
- A, B, C 3번 add 하고 commit

⇒ 현재 상태

- commit 3번, 현재 브랜치는 main

3) 현재 브랜치 확인

⇒ git status

⇒ git log --oneline

- 한 줄 씩 로그 확인

⇒ git branch

- 현재 branch 확인 - main

4) 브랜치 생성 명령

⇒ git branch 브랜치이름

- 현재 브랜치를 복제해서 새 브랭치를 생성하게 됨

⇒ dup 라는 브랜치를 생성

- git branch dup

5) 체크 아웃

⇒ checkout 은 해당 브랜치로 작업 환경을 변경하는 것

⇒ git checkout <브랜치>

- dup 브랜치로 이동하려면 git checkout dup
- Switched to branch 'dup’ 가 출력되면 이동 성공

⇒ dup 브랜치에서 새로 파일을 만들고 저장한 후 commit

- d.txt 파일

```
# add
git add d.txt
# commit
git commit -m "fourth commit"

# 확인
> git log --oneline

019d754 (HEAD -> dup) fourth commit
03ccc1a (main) third commit
04ff10a second commit
1dd1427 first commit

# d.txt 파일에 내용을 수정하고 저장, commit
# 이후에 다시 확인
> git log --oneline

d9f1d3e (HEAD -> dup) fifth commit
019d754 fourth commit
03ccc1a (main) third commit
```

```
04ff10a second commit
1dd1427 first commit
```

⇒ 다시 main 브랜치로 이동

- git checkout main
- main 브랜치에서 commit 을 완료한 후 dup 브랜치에서 추가한 d.txt 는 화면(vs code)에서 사라짐
- 자신 브랜치의 마지막으로 돌아가는 것으로 파일이 보이지 않는게 맞음

⇒ 2개의 브랜치를 비교

- git diff <브랜치1> <브랜치2>

```
# 비교
git diff main dup

# 결과 - d.txt 파일에 대한 변화 사항을 보여줌
diff --git a/d.txt b/d.txt
new file mode 100644
index 0000000..1abdff3
--- /dev/null
+++ b/d.txt
@@ -0,0 +1,2 @@
+DDD
+EEE
\ No newline at end of file
```

⇒ 브랜치를 생성하면서 체크 아웃(이동)

- -b 옵션을 이용
- git checkout -b <브랜치이름>

⇒ new 라는 브랜치를 생성하면서 체크 아웃

- git checkout -b new

⇒ git hub 에 push 를 할 때 main 브랜치가 없어서 push 를 할 수 없는 경우 git checkout -b main 을 수행한 후 push 를 하면 됨

6) 브랜치 병합

⇒ git merge <브랜치>

⇒ 현재 브랜치에 브랜치를 병합하는 기능 - 변경 내용을 적용하는 기능

⇒ main 브랜치에 dup 브랜치의 내용을 병합

```
# main 브랜치로 이동
git checkout main

# dup 브랜치의 내용 통합
git merge dup
# 브랜치 통합의 경우 status 나 log 를 사용해서 확인한 수 수행하는게 좋음

# 수행 결과 - d.txt 파일이 추가
Updating 03ccc1a..d9f1d3e
Fast-forward
 d.txt | 2 ++
 1 file changed, 2 insertions(+)
 create mode 100644 d.txt
```

- dup 브랜치에서 생성한 파일이 추가됨
- 이전에 main 브랜치로 이동하면서 보이지 않게 되었던 d.txt 파일을 이제 vscode 에서 다시 확인할 수 있음

7) 충돌

⇒ 브랜치를 병합할 때 충돌이 발생할 수 있음

- 이 경우는 서로 다른 브랜치에서 동일한 파일을 서로 다른 내용을 수정한 상태에서 병항을 하려 하는 상황에서 발생

⇒ main 브랜치에서 a.txt 파일을 수정하고 commit

- 이후 dup 브랜치에서 g.txt 파일을 생성하고 commit
- main 브랜치에서 dup 브랜치를 병합

```
> git add a.txt
> git commit -m "a.txt fix"

> git checkout dup
> git add g.txt
> git commit -m "g.txt add"

> git checkout main
> git merge dup
```

- 별다른 문제 없이 병합이 수해됨
- 서로 다른 브랜치에서 서로 다른 파일을 수정했기 때문
- 즉, 다른 브랜치에서 다른 파일을 가지고 작업을 하는 경우는 문제가 발생하지 않음

⇒ main 과 dup 브랜치에서 서로 동일한 이름의 파일을 가지고 작업

```
git add a.txt
git commit -m "test"

# dup 브랜치에서 a.txt 파일 내용을 다르게 수정하고 commit
git checkout dup
git add a.txt
git commit -m "change"

# main 브랜치로 이동해서 merge 수행
git checkout main
git merge dup

# 수행 결과
Auto-merging a.txt
CONFLICT (content): Merge conflict in a.txt
Automatic merge failed; fix conflicts and then commit the result.
```

- merge 에 conflict 가 발생하고 merge fail

- 파일을 살펴보면 2개 브랜치의 변경 내용이 모두 나타남 - merge editor

```
# 2가지 변경 사항이 존재하는 a.txt 파일
# HEAD 는 현재 브랜치의 변경 내용이고 아래는 다른 브랜치의 내용

<<<<<<< HEAD
test
=======
change!
>>>>>>> dup
```

⇒ 이런 충돌이 발생한 경우 변경 내용 중 하나를 선택한 다음 다시 add 를 수행하고 commit 을 수행해야 함

- 남길(반영할) 변경 사항을 제외한 나머지를 삭제한 후 다시 add, commit
- 이제 2개의 브랜치의 파일 내용이 서로 달라졌으므로 이에 대한 수정이 필요

8) 브랜치 삭제

⇒ git branch -d <브랜치> 또는 git branch --delete <브랜치>

⇒ 현재 브랜치는 삭제할 수 없으므로 다른 브랜치로 이동한 다음 삭제해야 함

⇒ new 브랜치 삭제

```
# 자신의 위치에서 브랜치를 삭제
git branch -d main
# 자신이 있는 브랜치이므로 삭제할 수 없어서 에러 발생

git branch -d new
```

9) 브랜치 재배치

⇒ git rebase <브랜치>

⇒ 브랜치의 재배치는 뻗어나온 기준점을 옮기는 방법

⇒ A 브랜치의 기준점을 B 브랜치의 가장 최근 commit 으로 이동하기

- A 브랜치에서 git rebase B브랜치 를 수행하면 됨
- merge 를 하지 않고도 변경 내용을 가져올 수 있지만 merge 와는 다른 개념

⇒ git hub 에서 다른 유저의 프로젝트에 fork(복사) 를 설정했는데 그 유저의 branch 에 변경이 발생하면 fork 된 프로젝트에는 경고가 발생

    - 이런 경우에는 프로젝트를 업데이트 하라는 메세지가 출력되는데 이 작업이 rebase 와 유사함

⇒ git log --oneline --branches rebase 를 수행해보면 commit 의 수행 순서가 변경되어 있을 수 있음

- 이전에 수행했던 commit 이 더 최근에 수행한 commit 으로 배치가 바뀌게 되는 것
- 브랜치의 기준점이 다른 브랜치의 가장 최근의 commit 으로 변경되었기 때문임

8. 로컬 저장소 관련 명령

⇒ git init : 로컬 저장소 생성

⇒ git status : 작업 디렉토리의 상태 확인

⇒ git add

- git add 파일경로 : 파일 경로에 해당하는 파일을 스테이지에 추가
- git add . : 현재 디렉토리의 모든 파일을 스테이지에 추가

⇒ git commit : 메세지의 제목과 내용을 설정해서 스테이지의 내용을 로컬 저장소에 추가

⇒ git commit -m(혹은 --message) “메세지 제목” : 메세지 제목만 설정해서 스테이지의 내용을 로컬 저장소에 추가

⇒ git log : commit 목록을 출력

- git log --oneline : 한 줄로 간결하게 출력
- git log --patch : 자세하게 출력
- git log --graph : 그래프로 출력
- git log --branch : 모든 브랜치의 commit 목록을 출력

⇒ git tag 태그 : 현재 브랜치에 태그(보통 버전)를 추가

- git tag 태그 커밋 : commit 에 태그를 추가
- git tag --list : 태그 리스트 조회
- git tag -l : 태그 리스트 조회
- git tag -d 태그 : 태그 삭제
- git tag --delete

⇒ git diff

- git diff : 최근 commit 과 현재 작업 디렉토리를 비교
- git diff --staged : 최근 commit 과 스테이지를 비교
- git diff 커밋1 커밋2 : commit 끼리 비교
- git diff 브랜치1 브랜치2 : 브랜치 비교

⇒ git reset

- git reset --soft 되돌아갈커밋 : commit 으로 soft 리셋
- git reset 되돌아갈커밋 : commit 으로 mixed 리셋
- git reset --hard 돌아갈커밋 : commit 으로 hard 리셋

⇒ git revert 취소할커밋 : commit 을 취소한 지점으로 새로운 commit 을 생성

⇒ git branch : 브랜치 확인

- git branch 브랜치이름 : 브랜치 생성
- git checkout 브랜치이름 : 브랜치로 이동

- git branch --delete(혹은 -d) : 브렌치 삭제
- git checkout -b 브랜치이름 : 브랜치를 생성하고 체크 아웃
- git merge 브랜치이름 : 현재 브랜치로 브랜치이름 에 해당하는 브랜치를 병함
- git rebase 브랜치이름 : 현재 브랜치의 기준점을 브랜치이름 에 해당하는 브랜치의 가장 최근 commit 으로 이동