



Git은 분산 버전 관리 시스템으로, 기본적으로 로컬에서 작업할 수 있습니다. 즉, Git 저장소는 사용자의 로컬 컴퓨터에 저장되며, 모든 버전 기록과 파일 변경 사항은 로컬에서 관리됩니다.

하지만 Git은 원격 저장소와도 연동할 수 있습니다. 원격 저장소는 GitHub, GitLab, Bitbucket 등과 같은 서비스에서 호스팅되며, 로컬 저장소와 원격 저장소 간에 코드를 푸시하거나 풀하는 등의 작업을 통해 협업할 수 있습니다. 그래서 Git은 로컬과 원격 모두에서 작업할 수 있는 시스템이라고 할 수 있습니다.



Git -> 분산 버전 관리시스템, 기본적으로 로컬에서 작업할 수 있음

Git 저장소는 로컬 컴퓨터에 저장, 모든 버전 기록과 파일 변경 사항은 로컬에서 관리됨

- Git은 원격 저장소와도 연동가능 (GitHub, GitLab, Bitbucket) 등과 같은 서비스에서 호스팅되며, 로컬 저장소와 원격 저장소 간에 코드를 통해 (push, pull) 협업가능

다시말해, Git은 로컬과 원격 모두에서 작업할 수 있는 시스템이라고 할 수 있음

command창이나 git bash에서 git.init같은 명령어는 local에서 활용되는 명령어일거아냐?

맞습니다. `git init` 명령어는 로컬에서 Git 저장소를 초기화하는 데 사용됩니다. 이 명령어를 실행하면 현재 디렉토리에 `.git` 이라는 숨김 폴더가 생성되며, 이 폴더에는 Git의 버전 관리 정보와 메타데이터가 저장됩니다.

즉, `git init` 은 로컬 파일 시스템에서 Git 버전 관리의 첫 단계를 설정하는 명령어로, 원격 저장소와는 직접적으로 관련이 없습니다. 이후에는 `git add`, `git commit`, `git status` 등과 같은 명령어들을 사용하여 로컬 저장소에서 파일을 관리하고, 원격 저장소와의 상호작용은 `git remote` 와 `git push`, `git pull` 명령어를 통해 이루어집니다.

로컬 저장소에서 파일을 관리하는 코드 : `git init` `git add`, `git commit`, `git status`

원격 저장소와 상호작용하는 코드 : `git remote`, `git push`, `git pull`

- git commit (local pc내의 특정 폴더를 저장소(repository)로 만들려면 repo를 가르키는 master가 필요하고, 이것은 최초 1회 commit을 통해 생성가능함
- 

- git의 사용목적은 다수의 사용자와 함께 프로젝트를 진행하는 것이기 때문에, 전체 프로젝트를 저장소로 옮기는 것이 아닌, '프로젝트 정보와 변경 사항이 적용되는 bare repository'가 원격저장소로 사용됨

## 1. 로컬에서 변경 사항 확인

Git에서 변경 사항을 확인하는 방법은 몇 가지가 있습니다.

- `git status`:

- 현재 작업 디렉토리의 상태(추적되지 않은 파일, 스테이징된 파일 등)를 보여줍니다.
- 예시:

```
bash 코드 복사  
  
$ git status
```

- `git diff`:

- 아직 커밋하지 않은 변경 사항을 비교하여 보여줍니다.
- 예시:

```
bash 코드 복사  
  
$ git diff
```

- `git log`:

- 프로젝트의 모든 커밋 내역을 시간 순서대로 볼 수 있습니다. 각 커밋 메시지와 커밋된 파일에 대한 정보가 표시됩니다.
- 예시:

```
bash 코드 복사  
  
$ git log
```

옵션으로 `--oneline` 을 사용하면 간단하게 커밋 내역을 한 줄씩 요약해서 볼 수 있습니다.

```
bash 코드 복사  
  
$ git log --oneline
```

## 2. GitHub에 올리기 (Push)

GitHub에 변경 사항을 반영하기 위해서는 `git push` 명령어를 사용합니다. 먼저 원격 저장소 (GitHub)와 로컬 저장소가 연결되어 있어야 하며, 그렇지 않다면 아래 명령어로 원격 저장소를 설정할 수 있습니다:

```
bash
```

[코드 복사](#)

```
$ git remote add origin https://github.com/yourusername/your-repository.git
```

그리고 로컬 커밋을 GitHub로 업로드하기 위해서는 다음 명령어를 사용합니다:

```
bash
```

[코드 복사](#)

```
$ git push origin master
```

여기서 `origin`은 원격 저장소의 별칭이며, `master`는 로컬의 브랜치 이름입니다. 원격 저장소가 `main` 브랜치를 사용하는 경우에는 `master` 대신 `main`을 사용해야 할 수 있습니다.

## 3. GitHub에서 변경 사항 확인 (Follow-up)

GitHub에서는 푸시된 변경 사항을 다음과 같이 확인하고 관리할 수 있습니다:

- 커밋 내역 (Commit History):
  - GitHub에서 특정 저장소에 들어가면 "Commits" 탭에서 커밋 내역을 확인할 수 있습니다. 각 커밋 메시지와 함께 커밋된 파일의 변경 사항을 볼 수 있습니다.
- 파일 비교 (Diff):
  - GitHub에서는 커밋 간의 차이점도 쉽게 확인할 수 있습니다. 변경된 파일들이 어떻게 수정되었는지 커밋 간 비교가 가능합니다.
- GitHub에서 Pull Request:
  - GitHub는 팀 프로젝트에서 사용될 때, 서로 다른 브랜치에서 작업한 후 이를 합치기 위한 **Pull Request** 기능을 제공합니다. 이를 통해 다른 팀원들이 내 변경 사항을 리뷰할 수 있습니다.

예시)

## 2.1 git 저장소(repository) 만들기

그런데, 새로 만들 폴더를 저장소(repository)로 만드려면, repo를 가르키는 master가 필요하고, 이것은 **최초 1회 commit**을 통해 생성할 수 있습니다.  
그렇기 때문에 아래 사진처럼 ReadMe.txt 파일을 add하고 commit을 함으로써 저장소가 생성이 된 것입니다!

```
DG@DESKTOP-5LF5EV5 MINGW64 ~/git/Blog_git/project_origin (master)
$ git add .
warning: LF will be replaced by CRLF in ReadMe.txt.
The file will have its original line endings in your working directory.

DG@DESKTOP-5LF5EV5 MINGW64 ~/git/Blog_git/project_origin (master)
$ git status
On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

        new file:   ReadMe.txt

DG@DESKTOP-5LF5EV5 MINGW64 ~/git/Blog_git/project_origin (master)
$ git commit -m "Create test repository"
[master (root-commit) 73a331d] Create test repository
 1 file changed, 1 insertion(+)
 create mode 100644 ReadMe.txt

DG@DESKTOP-5LF5EV5 MINGW64 ~/git/Blog_git/project_origin (master)
$ git log
commit 73a331d8bc61aa54519d08cf7d960cb217229c26
Author: DESKTOP-5LF5EV5\DG <tmzkd195@naver.com>
Date:   Tue May 21 14:19:19 2019 +0900

    Create test repository
```

사진 3. git repository 생성

위 `git commit -m "Create test repository"`는 local에서 git을 관리하는 코드에 속하나?



네, `git commit -m "Create test repository"`는 로컬에서 Git을 관리하는 명령어입니다. 이 명령어는 현재 로컬 저장소에 변경 사항을 기록(커밋)하는 작업을 수행하며, 로컬 저장소에서 버전 관리를 할 때 사용하는 기본적인 Git 명령어입니다.

이 커밋은 아직 원격 저장소에 반영된 것이 아니라, 로컬에서만 저장된 상태입니다. 원격 저장소에 이 커밋을 반영하려면 `git push` 명령어를 사용해야 합니다.

🔊 📄 🌟 🔄 ⚙️

"Create test repository"는 뭐야?



"Create test repository"는 `git commit -m` 명령어에서 사용된 커밋 메시지입니다. 이 메시지는 해당 커밋에 대한 설명을 짧게 작성한 것으로, 이번 커밋의 내용이나 목적을 알 수 있게 해줍니다.

특히 "Create test repository"라는 메시지는 이 커밋이 테스트용 저장소를 만들었다는 의미로 사용된 것입니다. 즉, 이 커밋은 처음으로 프로젝트를 초기화하거나 설정 파일을 추가한 작업을 기록하는 커밋인 것으로 보입니다. 커밋 메시지는 자유롭게 작성할 수 있으며, 작업의 목적을 명확히 하기 위해 작성됩니다.



오늘은 "Create test repository" 라는 커밋 메시지를 사용하고, 내일은 "Second test repo" 라는 커밋 메시지를 사용하는 경우, 두 개의 커밋이 서로 다른 시점에 기록된 변경 사항을 나타내게 됩니다.

커밋 메시지는 각 커밋이 어떤 변경 작업을 수행했는지 설명하는 데 사용되므로, 다음과 같은 방식으로 해석할 수 있습니다:

1. 오늘의 커밋 ("Create test repository"):

- 테스트용 저장소를 처음으로 생성하고, 초기 설정이나 파일을 추가한 경우를 설명하는 커밋입니다.

2. 내일의 커밋 ("Second test repo"):

- 이후에 추가적인 변경 작업을 수행한 커밋입니다. 예를 들어, 새로운 파일을 추가하거나 기존 파일을 수정한 경우일 수 있습니다.

각 커밋은 프로젝트의 변경 내역을 기록하기 때문에, 커밋 메시지를 통해 어떤 작업이 이루어졌는지 쉽게 파악할 수 있습니다. 커밋 메시지는 짧지만 명확하게 작성하는 것이 좋습니다. "Second test repo" 같은 메시지는 그 커밋이 정확히 무엇을 했는지 알기 어려울 수 있으므로, 더 구체적으로 작성하는 것이 권장됩니다. 예를 들어 "Add README file" 이나 "Fix bug in script" 처럼 작업 내용을 명확하게 나타내는 것이 좋습니다.

commit시, "create test repo"와 "second test repo", "third test repo" 이런식으로 version1, 2, 3 update한 내용들을 git과 github 에서 모두 확인할 수 있다.

first commit, second commit, third commit, .. 이런식으로 진행할 시,  
git init을 계속해서 해줘야하나?



`git init` 명령어는 새로운 Git 저장소를 만들 때 한 번만 실행하면 됩니다. 즉, 이미 초기화된 저장소에서 첫 번째 커밋을 만든 후에는 `git init` 을 다시 할 필요가 없습니다.

일단 저장소를 한 번 초기화하고 나면, 그 이후에는 계속해서 파일을 수정하고, `git add` → `git commit` 순서로 커밋을 추가할 수 있습니다. `git init` 을 다시 할 필요 없이 첫 번째 커밋 이후로 추가적인 커밋을 계속 진행할 수 있습니다.

예를 들어, 아래와 같은 단계로 진행하면 됩니다:

### 1. 첫 번째 커밋 (First commit)

```
bash 코드 복사  
  
$ git init          # 저장소 초기화 (한 번만)  
$ git add .         # 파일을 스테이징  
$ git commit -m "First commit" # 첫 번째 커밋
```

### 2. 두 번째 커밋 (Second commit)

```
bash 코드 복사  
  
$ git add .         # 수정된 파일을 다시 스테이징  
$ git commit -m "Second commit" # 두 번째 커밋
```

### 3. 세 번째 커밋 (Third commit)

```
bash 코드 복사  
  
$ git add .         # 새로운 변경 사항을 스테이징  
$ git commit -m "Third commit" # 세 번째 커밋
```

이처럼 `git init` 은 저장소를 처음 만들 때 한 번만 사용하고, 그 이후에는 `git add`, `git commit` 을 반복해서 커밋을 추가하는 방식으로 작업을 계속 진행하면 됩니다.

first commit, second commit, third commit, .. 이런식으로 진행할 시, git init을 계속해서 해줘야하나?



`git init` 명령어는 새로운 Git 저장소를 만들 때 한 번만 실행하면 됩니다. 즉, 이미 초기화된 저장소에서 첫 번째 커밋을 만든 후에는 `git init` 을 다시 할 필요가 없습니다.

일단 저장소를 한 번 초기화하고 나면, 그 이후에는 계속해서 파일을 수정하고, `git add` → `git commit` 순서로 커밋을 추가할 수 있습니다. `git init` 을 다시 할 필요 없이 첫 번째 커밋 이후로 추가적인 커밋을 계속 진행할 수 있습니다.

예를 들어, 아래와 같은 단계로 진행하면 됩니다:

### 1. 첫 번째 커밋 (First commit)

```
bash 코드 복사  
  
$ git init          # 저장소 초기화 (한 번만)  
$ git add .         # 파일을 스테이징  
$ git commit -m "First commit" # 첫 번째 커밋
```

### 2. 두 번째 커밋 (Second commit)

```
bash 코드 복사  
  
$ git add .         # 수정된 파일을 다시 스테이징  
$ git commit -m "Second commit" # 두 번째 커밋
```

### 3. 세 번째 커밋 (Third commit)

```
bash 코드 복사  
  
$ git add .         # 새로운 변경 사항을 스테이징  
$ git commit -m "Third commit" # 세 번째 커밋
```

이처럼 `git init` 은 저장소를 처음 만들 때 한 번만 사용하고, 그 이후에는 `git add`, `git commit` 을 반복해서 커밋을 추가하는 방식으로 작업을 계속 진행하면 됩니다.



first commit, second commit, third commit, .. 이런식으로 진행할 시, git init을 계속해서 해줘야하나?



`git init` 명령어는 새로운 Git 저장소를 만들 때 한 번만 실행하면 됩니다. 즉, 이미 초기화된 저장소에서 첫 번째 커밋을 만든 후에는 `git init` 을 다시 할 필요가 없습니다.

일단 저장소를 한 번 초기화하고 나면, 그 이후에는 계속해서 파일을 수정하고, `git add` → `git commit` 순서로 커밋을 추가할 수 있습니다. `git init` 을 다시 할 필요 없이 첫 번째 커밋 이후로 추가적인 커밋을 계속 진행할 수 있습니다.

예를 들어, 아래와 같은 단계로 진행하면 됩니다:

### 1. 첫 번째 커밋 (First commit)

```
bash
$ git init          # 저장소 초기화 (한 번만)
$ git add .         # 파일을 스테이징
$ git commit -m "First commit" # 첫 번째 커밋
```

### 2. 두 번째 커밋 (Second commit)

```
bash
$ git add .         # 수정된 파일을 다시 스테이징
$ git commit -m "Second commit" # 두 번째 커밋
```

### 3. 세 번째 커밋 (Third commit)

```
bash
$ git add .         # 새로운 변경 사항을 스테이징
$ git commit -m "Third commit" # 세 번째 커밋
```

이처럼 `git init` 은 저장소를 처음 만들 때 한 번만 사용하고, 그 이후에는 `git add`, `git commit` 을 반복해서 커밋을 추가하는 방식으로 작업을 계속 진행하면 됩니다.

`git init`으로 repository를 일단 만들면, `git init`을 하지않고, `git add` . `git comit -m`으로 계속 진행해 주면 된다. `git init`을 계속 진행할 시, 오히려 기존의 저장소에 데이터가 덮혀버리는 문제가 생길 수 있으니 주의해야한다.

## commit 여러 버전들 확인하는법

```
Windows PowerShell
On branch main
nothing to commit, working tree clean
PS G:\내 드라이브\옵시디언\Obsidian_Github> git add .
warning: in the working copy of 'test 2.md', LF will be replaced by CRLF the next time Git touches it
PS G:\내 드라이브\옵시디언\Obsidian_Github> git commit -m "Thrid commit"
[main 4eee49b] Thrid commit
1 file changed, 1 insertion(+)
create mode 100644 test 2.md
PS G:\내 드라이브\옵시디언\Obsidian_Github> git diff
PS G:\내 드라이브\옵시디언\Obsidian_Github> git log
commit 4eee49b1812bac5fcbbb0fae6b525498a966db6f (HEAD -> main)
Author: HAWNONA <skgk4620@gmail.com>
Date: Thu Sep 12 00:14:12 2024 +0900

    Thrid commit

commit afd90e20523b0e07bf57657e373bcb43ccaaef9
Author: HAWNONA <skgk4620@gmail.com>
Date: Thu Sep 12 00:11:46 2024 +0900

    First repo

PS G:\내 드라이브\옵시디언\Obsidian_Github> git --oneline
unknown option: --oneline
usage: git [-v | --version] [-h | --help] [-C <path>] [-c <name>=<value>]
    [--exec-path[=<path>]] [--html-path] [--man-path] [--info-path]
    [-p | --paginate | -P | --no-pager] [--no-replace-objects] [--no-lazy-fetch]
    [--no-optional-checkouts] [--no-advice] [--bare] [--git-dir=<path>]
    [--work-tree=<path>] [--namespace=<name>] [--config-env=<name>=<envvar>]
    <command> [<args>]
PS G:\내 드라이브\옵시디언\Obsidian_Github> |
```



만약 4번째 commit까지 했을 때, 두번째 commit 버전에서의 파일을 보고싶으면 어떻게 하면되?



Git에서 이전 커밋 버전의 파일을 확인하거나 특정 커밋으로 돌아가고 싶다면, 다음과 같은 방법을 사용할 수 있습니다.

## 1. `git checkout` 으로 특정 커밋 상태로 이동하기

특정 커밋의 파일 상태를 확인하려면, 커밋 해시를 사용해 그 시점의 파일 상태로 이동할 수 있습니다. 예를 들어, 두 번째 커밋 시점으로 돌아가려면:

### 1. 커밋 해시 확인하기

먼저 `git log` 명령어로 커밋 내역을 확인합니다.

bash

코드 복사

```
git log --oneline
```

예시 출력:

sql

코드 복사

```
f8e8c9c (HEAD -> main) 4th commit
b7d5e2a 3rd commit
a1b2c3d 2nd commit
9f8e7c1 1st commit
```

여기서, 두 번째 커밋은 `a1b2c3d` 라는 해시를 갖고 있다고 가정합니다.

## 2. 특정 커밋으로 이동하기

`git checkout` 을 사용하여 두 번째 커밋으로 이동합니다:

```
bash
```

[코드 복사](#)

```
git checkout a1b2c3d
```

이 명령어는 저장소의 상태를 두 번째 커밋 시점으로 변경하여 그 당시의 파일을 볼 수 있게 해줍니다. 그러나 이 상태에서는 **detached HEAD** 상태가 됩니다. 즉, 브랜치에 속하지 않은 상태에서 파일을 확인하는 것입니다.

## 3. 다시 원래 브랜치로 돌아오기

다시 원래 브랜치 상태로 돌아가려면:

```
bash
```

[코드 복사](#)

```
git checkout main
```

이 명령어를 실행하면 현재 브랜치(예: `main`)의 최신 커밋으로 돌아가게 됩니다.

## 예시

```
PS G:\내 드라이브\옵시디언\Obsidian_Github> git log
commit 4eee49b1812bac5fcbbb0fae6b525498a966db6f (HEAD -> main)
Author: HAWNONA <skgk4620@gmail.com>
Date: Thu Sep 12 00:14:12 2024 +0900

    Thrid commit

commit afd90e20523b0e07bf57657e373bcb43ccaaeaf9
Author: HAWNONA <skgk4620@gmail.com>
Date: Thu Sep 12 00:11:46 2024 +0900
```

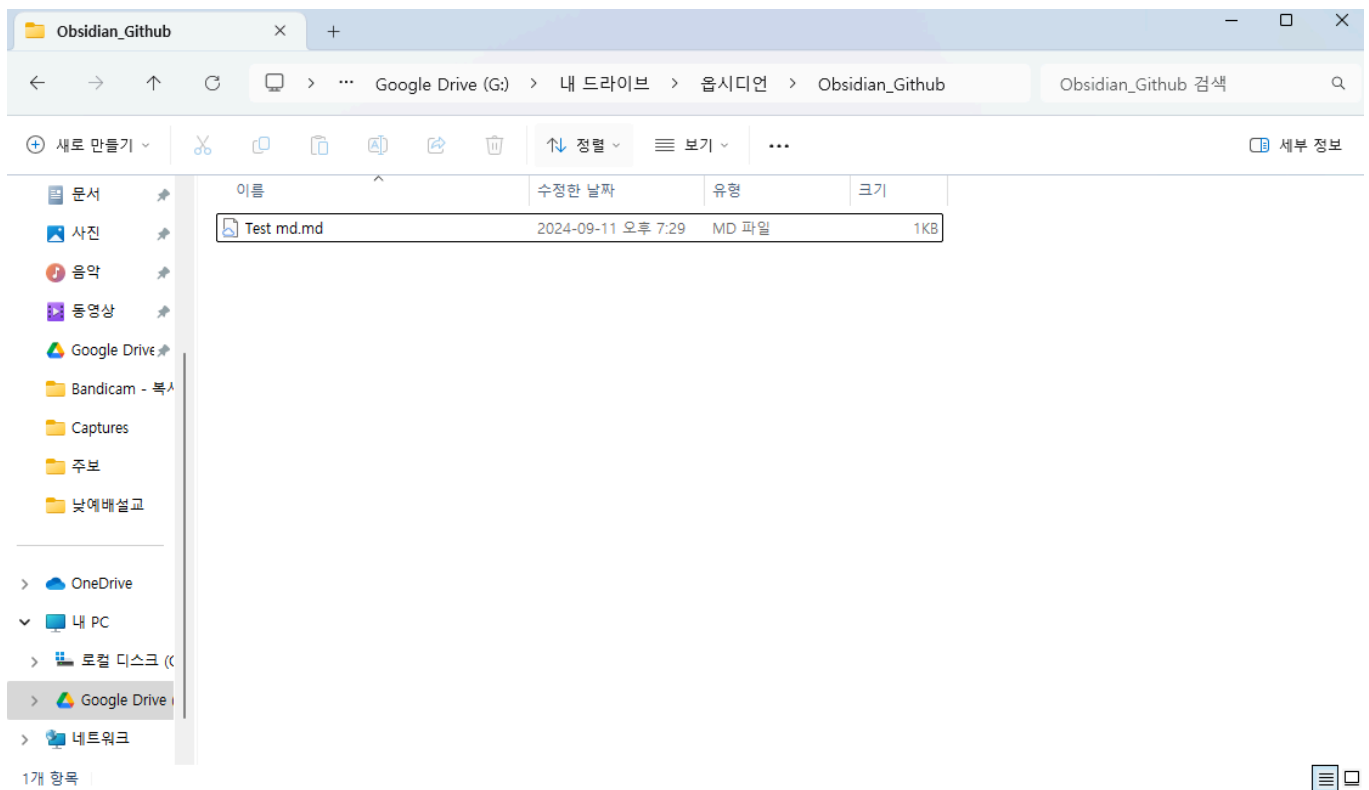
```
PS G:\내 드라이브\옵시디언\Obsidian_Github> git checkout afd90e20523b0e07bf57657e373bcb43ccaaeaf9
Note: switching to 'afd90e20523b0e07bf57657e373bcb43ccaaeaf9'.
```

You are in 'detached HEAD' state. You can look around, make experimental changes and commit them, and you can discard any commits you make in this state without impacting any branches by switching back to a branch.

If you want to create a new branch to retain commits you create, you may do so (now or later) by using `-c` with the switch command. Example:

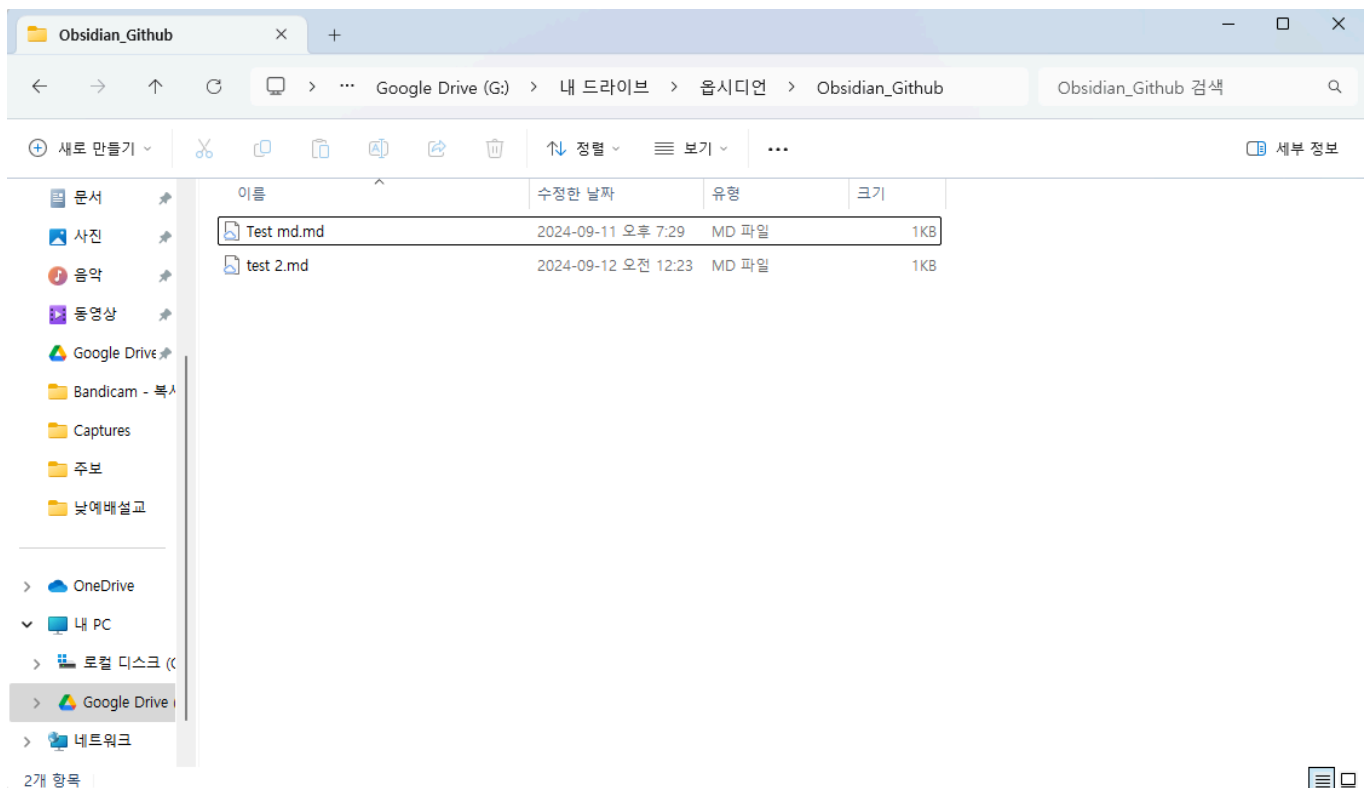
```
git switch -c <new-branch-name>
```

Or undo this operation with:



이전 버전으로 돌아옴

```
PS G:\내 드라이브\옵시디언\Obsidian_Github> git checkout main
Previous HEAD position was afd90e2 First repo
Switched to branch 'main'
```



최신버전으로 다시 돌아옴

## 2. `git show` 로 특정 파일만 확인하기

만약 두 번째 커밋의 전체 파일이 아닌 특정 파일만 보고 싶다면, `git show` 명령어를 사용할 수 있습니다. 예를 들어, 두 번째 커밋(`a1b2c3d`)에서 `example.txt` 파일의 상태를 보고 싶다면:

```
bash
```

[코드 복사](#)

```
git show a1b2c3d:example.txt
```

이 명령어는 특정 파일의 내용을 바로 보여줍니다.

---

## 3. `git diff` 로 커밋 간 차이 확인하기

두 번째 커밋과 다른 커밋 간의 차이점을 보고 싶다면, `git diff` 명령어를 사용할 수 있습니다. 예를 들어, 두 번째 커밋과 네 번째 커밋 간의 차이를 확인하려면:

```
bash
```

[코드 복사](#)

```
git diff a1b2c3d f8e8c9c
```

이 명령어는 두 커밋 사이에서 어떤 파일이 변경되었는지를 보여줍니다.

---

## 4. `git restore` 로 특정 파일만 복원하기

두 번째 커밋에서 특정 파일을 현재 작업 디렉토리로 복원하려면 `git restore` 를 사용할 수 있습니다. 예를 들어, 두 번째 커밋(`a1b2c3d`)에서 `example.txt` 파일만 복원하려면:

```
bash
```

[코드 복사](#)

```
git restore --source=a1b2c3d example.txt
```

이 명령어는 두 번째 커밋에서 `example.txt` 를 현재 디렉토리로 복원합니다.

### 요약

- 특정 커밋 상태로 이동하고 싶다면 `git checkout` [커밋 해시] .
- 특정 파일의 내용을 보고 싶다면 `git show` [커밋 해시]:[파일명] .
- 두 커밋 간 차이를 보고 싶다면 `git diff` [커밋 해시1] [커밋 해시2] .
- 특정 커밋의 파일을 현재 상태로 복원하려면 `git restore --source=[커밋 해시]` [파일명] .

이 방법들을 통해 원하는 커밋 시점의 파일을 쉽게 확인할 수 있습니다.

- 

## **bare repository 만들기**

```
$ git clone --bare {프로젝트 이름} _ {프로젝트 이름}.git
```

## **Branch와 Clone의 차이**

서로 다른 **브랜치(branch)**를 **clone**과 동일하게 판단하면 안 됩니다. 두 개념은 Git에서 다른 목적과 역할을 가지고 있습니다.

## **브랜치(branch)**와 **클론(clone)**의 차이점

### 1. 브랜치(branch):

- 브랜치는 하나의 Git 저장소 내에서 서로 다른 작업 흐름을 나누기 위해 사용됩니다.
- 동일한 Git 저장소 내에서, 여러 브랜치를 만들어 서로 독립적으로 작업할 수 있습니다.
- 예를 들어, `main` 브랜치에서는 안정된 버전의 코드가 관리되고, `feature/login` 브랜치에서는 새로운 기능이 개발될 수 있습니다.
- 브랜치 간에는 코드의 공유가 가능하며, 각 브랜치는 같은 저장소에서 다른 **역사(history)**를 가질 뿐입니다.

### 2. 클론(clone):

- **클론(clone)**은 전체 Git 저장소를 로컬로 복제하는 작업입니다.
- 원격 저장소(GitHub, GitLab 등)에 있는 저장소의 전체 히스토리, 브랜치, 커밋 등을 복사하여 로컬에 동일한 복사본을 생성합니다.
- 클론을 통해 만들어진 로컬 저장소는 원격 저장소와 동일한 상태로 시작하며, 이후 로컬에서 작업한 후 **푸시(push)**하여 변경 사항을 원격 저장소에 적용할 수 있습니다.
- 클론은 저장소 자체를 복사하는 것이지, 저장소 내의 특정 브랜치와 직접적으로 연결되는 개념은 아닙니다.



## 비교

개념	브랜치 (Branch)	클론 (Clone)
정의	동일한 저장소 내에서 독립적인 작업 흐름을 나누는 방법	원격 저장소를 복제하여 로컬 저장소로 가져오는 작업
역할	같은 프로젝트에서 서로 다른 버전/작업을 수행	전체 Git 저장소의 복사본을 로컬에 생성
작업 범위	하나의 저장소 안에서 발생하는 작업 흐름	원격 저장소의 전체 복사본을 가져옴
목적	기능 개발, 버그 수정, 실험 등을 독립적으로 진행하기 위해	원격 저장소의 모든 데이터를 로컬로 가져와 작업하기 위해
데이터 공유	같은 저장소 내에서 병합(merge)을 통해 변경 사항 공유 가능	클론 후 <code>pull</code> 과 <code>push</code> 로 원격 저장소와 데이터 공유 가능

## 예시

- **브랜치 작업:** 동일한 프로젝트 안에서 브랜치마다 서로 다른 작업을 진행할 수 있습니다. 예를 들어, `main` 브랜치에는 안정된 코드가 있고, `feature/login` 브랜치에서는 로그인 기능을 개발하고 있다고 할 수 있습니다.
- **클론 작업:** GitHub에 있는 프로젝트를 로컬로 복제하고, 로컬에서 작업을 진행한 후 다시 원격 저장소에 푸시(push)하여 다른 팀원들이 접근할 수 있게 합니다.

## 요약

- 브랜치는 하나의 저장소 내에서 작업 흐름을 분리하기 위한 방법으로, 서로 다른 브랜치에서 독립적으로 개발을 진행할 수 있습니다.
- 클론은 전체 저장소를 복제하는 작업으로, 원격 저장소의 모든 커밋, 브랜치, 히스토리를 로컬로 가져옵니다.

따라서, 브랜치는 하나의 저장소 내에서 작업 흐름을 분리하는 것이고, 클론은 저장소 자체를 복사하는 작업입니다.

## branch 종류



main branch는 뭐야? 또 다른 branch가 있나?



## main 브랜치란?

**main** 브랜치는 Git에서 기본 브랜치(default branch)를 의미합니다. 대부분의 Git 프로젝트는 **main** 브랜치에서 시작되며, 그 브랜치가 기본 작업 공간이 됩니다. 예전에는 **master** 브랜치가 기본이었지만, 최근에는 **main** 브랜치를 더 많이 사용합니다.

- **기본 브랜치**: 프로젝트를 생성하거나 초기화(`git init`)할 때 자동으로 생성되는 브랜치입니다. 대부분의 경우, 프로젝트의 최신 버전이 이 **main** 브랜치에 유지됩니다.
- **공식 릴리스 또는 주요 버전**: **main** 브랜치는 주로 주요 릴리스나 안정적인 버전을 관리하기 위한 브랜치로 사용됩니다.

## 브랜치(branch)란?

Git에서 브랜치는 코드의 다른 버전을 작업할 수 있게 하는 기능입니다. 프로젝트의 \*\*한 줄기(버전)\*\*를 만들고 나중에 그 변경 사항을 \*\*병합(merge)\*\*할 수 있습니다.

- 브랜치는 마치 시간 여행처럼 특정 지점에서 분기하여 독립적으로 작업할 수 있게 해줍니다.
- 각 브랜치는 서로 독립적으로 작업할 수 있으므로, 새로운 기능을 개발하거나 버그를 수정할 때도 다른 브랜치에 영향을 주지 않습니다.

## 다른 브랜치들

Git에서는 여러 브랜치를 쉽게 만들 수 있습니다. 일반적으로 사용되는 브랜치의 예는 다음과 같습니다.

1. **main** 브랜치:

- 프로젝트의 기본 브랜치로, 안정된 릴리스 버전이 포함됩니다.
- 모든 최종 수정 사항과 완성된 기능은 **main**에 병합됩니다.

2. 기능 브랜치 (Feature Branch):

- 새로운 기능을 개발할 때 사용하는 브랜치입니다.
- **feature/** 접두어를 사용하는 것이 일반적입니다.
- 예시: **feature/new-login-system**

```
bash
```

[코드 복사](#)

```
git checkout -b feature/new-login-system
```

3. 버그 수정 브랜치 (Bugfix Branch):

- 버그를 수정할 때 사용하는 브랜치입니다.
- 예시: **bugfix/fix-login-error**

```
bash
```

[코드 복사](#)

```
git checkout -b bugfix/fix-login-error
```

4. 개발 브랜치 (Development Branch):

- 새로운 기능들을 개발하고 테스트할 때 사용하는 브랜치로, **develop** 또는 **dev** 브랜치라고도 불립니다.
- **main** 브랜치에 병합하기 전에 이곳에서 작업을 합니다.

5. 릴리스 브랜치 (Release Branch):

- 배포 전에 마지막으로 테스트와 점검을 진행하는 브랜치입니다.
- 예시: **release/v1.0**

6. 핫픽스 브랜치 (Hotfix Branch):

- **main** 브랜치에 이미 배포된 코드에서 긴급한 버그 수정이 필요할 때 사용하는 브랜치입니다.
- 예시: **hotfix/security-patch**

## 브랜치 생성과 관리

Git에서 새로운 브랜치를 생성하고 전환하는 것은 매우 간단합니다.

### 1. 새로운 브랜치 생성:

bash

📄 코드 복사

```
git checkout -b feature/new-feature
```

위 명령은 `feature/new-feature` 라는 새 브랜치를 만들고 그 브랜치로 전환합니다.

### 2. 브랜치 목록 보기:

bash

📄 코드 복사

```
git branch
```

현재 저장소에 있는 브랜치들의 목록을 볼 수 있습니다.

### 3. 브랜치 전환:

bash

📄 코드 복사

```
git checkout main
```

위 명령은 다시 `main` 브랜치로 전환합니다.

### 4. 브랜치 병합 (Merge): 작업한 브랜치의 변경 사항을 다시 `main` 브랜치로 병합할 수 있습니다.

bash

📄 코드 복사

```
git checkout main  
git merge feature/new-feature
```

5. **브랜치 삭제**: 브랜치 작업이 끝났으면 필요 없는 브랜치는 삭제할 수 있습니다.

```
bash
```

[코드 복사](#)

```
git branch -d feature/new-feature
```

## 요약

- **main 브랜치**: Git에서 기본적으로 제공하는 브랜치로, 안정적인 코드 버전을 유지하는 곳.
- **브랜치**: 서로 독립적으로 작업할 수 있는 코드의 버전으로, 새로운 기능 개발이나 버그 수정에 유용함.
- Git은 여러 브랜치를 만들어 작업한 후, 필요에 따라 다른 브랜치와 **병합(merge)**할 수 있는 매우 유연한 시스템을 제공합니다.

브랜치를 이용해 **병렬 작업**을 하여 다른 팀원들과 충돌을 최소화할 수 있고, 안정적인 코드 관리가 가능합니다.

하나의 branch로 계속해서 commit하기보다, branch를 나누어 commit을 하게된다면 어떻게 보면 branch는 commit(저장)하는 것들이 다른 방향성을 띠 때, 다른 버전이라고 할 수 있나?

네, 정확히 말하면 **\*\*브랜치를 나누어 커밋(commit)\*\***하는 것은 서로 다른 방향성을 가진 작업을 독립적으로 진행하는 방식입니다. 그리고 이로 인해 브랜치는 각각 다른 버전이라고 볼 수 있습니다. 브랜치를 사용하면 서로 다른 작업 흐름을 병렬적으로 진행할 수 있고, 각 브랜치는 서로 다른 기능이나 수정 사항을 포함하는 별도의 작업 영역이 됩니다.

## 브랜치를 나누어 커밋하는 경우가 왜 '다른 버전'으로 간주될 수 있는가?

### 1. 독립적인 작업 흐름

- 각 브랜치는 기본적으로 서로 독립된 작업 흐름을 가지고 있습니다.
- 예를 들어, `main` 브랜치는 현재 배포된 버전이나 안정적인 코드를 유지하고, `feature/login` 브랜치에서는 로그인 기능을 개발할 수 있습니다.
- **\*\*커밋(commit)\*\***은 해당 브랜치에서의 변경 기록을 저장하며, **\*\*각 브랜치는 서로 다른 이력(history)\*\***을 유지합니다.

### 2. 다른 목적을 위한 브랜치

- 브랜치마다 다른 목표나 방향성이 있기 때문에, 각각의 브랜치는 별도의 버전으로 간주될 수 있습니다.
- 예를 들어, `main` 브랜치가 현재 제품에 적용된 안정적인 버전이라면, `develop` 브랜치는 앞으로의 기능 개발을 위한 브랜치일 수 있습니다. 또, 기능별 브랜치는 개별 기능을 실험하거나 개발하는 용도로 사용될 수 있죠.
- 각 브랜치에서 커밋된 내용은 그 목적에 맞는 변경 사항을 포함하기 때문에, 기능 개발이 끝난 후에는 **\*\*병합(merge)\*\***되기 전까지 서로 다른 버전으로 존재하게 됩니다.

### 3. 병합 전까지 다른 버전으로 존재

- 브랜치는 기본적으로 병합(merge)되기 전까지 서로 독립적으로 발전합니다. 이 의미는 각 브랜치가 해당 시점까지 서로 다른 버전으로 존재한다는 것을 의미합니다.
- 병합하지 않는 한, 한 브랜치에서 한 커밋이 반영된다고 해도, 다른 브랜치에는 해당 커밋이 반영되지 않으므로 각 브랜치는 서로 다른 버전의 내용을 유지할 수 있습니다.

## 예시: 브랜치의 사용 흐름

1. `main` 브랜치: 프로젝트의 안정적인 버전 또는 배포 버전이 관리됩니다.
  - 주로 최종 사용자에게 전달될 코드를 담고 있습니다.
2. `feature/로그인` 브랜치: 로그인 기능을 개발하는 브랜치입니다.
  - 로그인 기능과 관련된 변경 사항들을 추가하면서 계속해서 커밋합니다.
  - 이 브랜치에서는 로그인 기능을 독립적으로 개발하며, 다른 브랜치와는 독립적으로 발전합니다.
3. `feature/결제` 브랜치: 결제 시스템을 개발하는 브랜치입니다.
  - 결제 관련 코드가 커밋되고 이 브랜치에서는 결제 기능에 대한 작업을 독립적으로 진행합니다.

각 브랜치는 서로 다른 기능이나 수정사항을 담고 있어, 결국에는 브랜치마다 각기 다른 버전의 코드를 가지게 됩니다.

## 브랜치가 다른 버전으로 여겨지는 이유

- 동시에 독립적으로 작업을 할 수 있으며, 각 브랜치가 자신만의 커밋 기록과 이력을 가지기 때문에, 각각의 브랜치는 서로 다른 상태로 존재하게 됩니다.
- 브랜치의 목표가 다르다면, 그것은 곧 서로 다른 기능이나 버전의 개발을 의미합니다. 따라서 하나의 프로젝트 내에서도 서로 다른 버전처럼 기능하거나 역할을 할 수 있습니다.

## 브랜치 간의 병합(Merge)

- 브랜치 간의 작업을 마친 후, **\*\*병합(merge)\*\***을 통해 두 브랜치를 하나로 합칠 수 있습니다.
  - 예를 들어, `feature/login` 브랜치에서 로그인 기능이 완료되면, 이 브랜치를 `main` 브랜치에 병합하여 최종 배포 코드에 통합할 수 있습니다.
  - 병합 후에는 하나의 브랜치가 전체 기능을 모두 포함하게 되며, 이때 비로소 여러 브랜치가 하나의 최종 버전으로 통합됩니다.

## 요약

- **\*\*브랜치(branch)\*\***는 서로 다른 방향성을 가진 작업을 독립적으로 진행할 수 있게 하며, 각각의 브랜치가 다른 버전이라고 생각할 수 있습니다.
- 브랜치를 나누어 작업을 하면, 각 브랜치에는 서로 다른 커밋과 변경 기록이 쌓이게 되고, 병합 전까지는 각각 독립적인 버전으로 존재합니다.
- 브랜치를 사용하면 병렬적으로 작업을 진행할 수 있어, 여러 기능이나 수정 사항을 동시에 개발하고 나중에 병합할 수 있습니다.

가려 하나는 regression으로 진행, 하나는 classification으로 같은 프로젝트를 다른 방향성을 가지고 진행하려할 때, branch를 나누는건가?

네, \*\*회귀(Regression)\*\*와 \*\*분류(Classification)\*\*는 각각 다른 목표와 방향성을 가진 작업이므로, 이 경우 브랜치를 나누는 것이 적절한 방법입니다. 이렇게 하면 각 작업을 독립적으로 진행하면서도, 나중에 필요에 따라 병합하거나 각 버전의 변경 사항을 쉽게 관리할 수 있습니다.

### 왜 브랜치를 나누어야 하는가?

#### 1. 다른 목표와 방향성:

- \*\*회귀(Regression)\*\*와 \*\*분류(Classification)\*\*는 데이터 처리 방식과 모델링 목표가 다릅니다. 회귀는 연속적인 값을 예측하는 반면, 분류는 이산적인 범주를 예측합니다.
- 따라서, 각각의 작업에서는 다른 모델링 기법, 평가 방법, 전처리 과정 등이 필요할 수 있습니다.

#### 2. 작업의 독립성:

- 브랜치를 나누면 회귀 모델과 분류 모델을 독립적으로 개발하고 테스트할 수 있습니다. 회귀 작업에서의 변경 사항이 분류 작업에 영향을 주지 않고, 반대로 분류 작업이 회귀 모델에 영향을 미치지 않습니다.
- 예를 들어, 회귀 모델을 위해 특정한 전처리 과정을 추가했을 때, 분류 모델에서는 그 전처리가 필요 없을 수도 있습니다.

#### 3. 별도의 커밋 기록 관리:

- 각 브랜치에서 작업한 내용은 서로 독립적으로 커밋되므로, 나중에 어떤 변경 사항이 각 작업에 어떤 영향을 미쳤는지 명확하게 추적할 수 있습니다.
- 예를 들어, `regression` 브랜치에서는 회귀 작업과 관련된 커밋만 남고, `classification` 브랜치에서는 분류 작업과 관련된 커밋만 기록됩니다.



## 실제 브랜치 전략

### 1. 메인 브랜치(main):

- 프로젝트의 기본 코드나 공통적으로 사용하는 코드는 `main` 브랜치에 보관할 수 있습니다.
- 공통적인 데이터 전처리, 데이터 로딩 코드, 프로젝트 설정 등은 이 브랜치에서 관리합니다.

### 2. 회귀 작업 브랜치(regression):

- 회귀 모델에 필요한 특정한 코드나 전처리 과정, 모델 선택, 평가 지표 등을 추가합니다.
- 이 브랜치에서 회귀 작업을 독립적으로 진행하며, 관련된 커밋을 이 브랜치에 기록합니다.

```
bash 코드 복사

git checkout -b regression
# 회귀 모델 작업 진행
git commit -m "Implement regression model"
```

### 3. 분류 작업 브랜치(classification):

- 분류 모델을 위한 별도의 코드나 전처리 과정, 모델 선택, 평가 방법 등을 관리합니다.
- 이 브랜치에서 분류 작업을 독립적으로 진행하며, 관련된 커밋을 이 브랜치에 기록합니다.

```
bash 코드 복사

git checkout -b classification
# 분류 모델 작업 진행
git commit -m "Implement classification model"
```

## 브랜치를 나누는 장점

- 병렬 개발 가능: 각 작업을 동시에 진행하면서도 서로 영향을 주지 않습니다.
- 버전 관리 용이: 각 작업에 대한 독립적인 커밋 기록이 남아, 추후에 어떤 부분이 어떻게 변경되었는지 쉽게 추적할 수 있습니다.
- 병합 가능성: 작업이 완료되면, 두 브랜치를 병합할 수 있습니다. 예를 들어, 공통된 전처리 과정을 `main` 브랜치에 병합하고, 모델별 코드만 따로 관리할 수도 있습니다.

## 예시 시나리오

1. `regression` 브랜치에서 회귀 작업을 진행하여, 연속형 변수를 예측하는 모델을 만듭니다.
2. `classification` 브랜치에서 분류 작업을 진행하여, 이산적인 클래스(예: 고양이 vs 개)를 예측하는 모델을 개발합니다.
3. 각 작업이 끝난 후, 필요에 따라 두 브랜치의 공통 코드를 `main` 브랜치에 병합할 수 있습니다.

## 요약

- Regression과 Classification은 서로 다른 방향성을 가진 작업이므로, 각각 별도의 브랜치로 작업을 나누는 것이 좋은 방법입니다.
- 이렇게 하면 각각의 작업이 독립적으로 진행되며, 필요 시 작업을 병합하거나 병렬로 유지할 수 있습니다.

- Branch -> 같은 프로젝트 내, 다른 버전의 모델을 적용해보고 싶을 때 진행
- commit -> save (저장하기)와 다를 것 없음
- 가령, main-branch에서 commit 3개 진행, 이후 첫번째 new branch를 만들어 branch를 첫번째로 옮기고 4번째 commit 진행, 이후 두번째 new branch를 만들어 branch를 두번째로 옮

기고, 5번째 commit 진행

-> 이 경우, main, 첫번째, 두번째 branch 모두 commit 3개에 대한 정보는 동일하게 가짐

-> 첫번째 branch의 경우, 4번째 commit 정보를 가짐

-> 두번째 branch의 경우, 5번째 commit 정보를 가짐

즉, main -> 1,2,3 / 첫번째 -> 1,2,3,4 / 두번째 -> 1,2,3,5

## **Clone은 언제사용?**

실제 local에서 작업 시 clone 활용이 많나? 아니면 그냥 계속 commit 하여 commit된 버전들을 역으로 찾아내 사용하나/

실제 로컬에서 작업할 때 clone과 commit은 모두 중요한 역할을 하지만, 그 사용 목적과 시점은 다릅니다. 일반적으로, clone은 처음 한 번 사용하고, 이후로는 commit과 push, pull 같은 명령어들을 반복적으로 사용하여 버전 관리를 합니다.

### 1. Clone: 첫 시작을 위한 과정

- `git clone`은 원격 저장소를 처음 로컬로 가져올 때 사용됩니다. 즉, 원격 저장소에 있는 프로젝트를 복제하여 로컬에서 동일한 작업 환경을 만들고자 할 때 사용하는 것이죠.
- 일반적으로 개발을 시작하거나 새로운 팀원이 프로젝트에 합류할 때, GitHub, GitLab 등의 원격 저장소에서 프로젝트를 clone하여 로컬 작업 환경을 설정합니다.

예를 들어:

```
bash
git clone https://github.com/username/repository.git
```

이후 로컬에 복제된 저장소를 사용하여 작업을 진행합니다.

### 2. Commit: 로컬에서의 주기적인 작업 기록

- `git commit`은 로컬에서 코드 작업을 진행할 때 주기적으로 변경 사항을 기록하는 과정입니다. 하나의 작업 단위(기능 추가, 버그 수정 등)가 완료될 때마다 `git commit`을 사용하여 현재 상태를 저장해 둡니다.
- 로컬에서 여러 번의 커밋(commit)을 한 후, 해당 커밋들을 원격 저장소에 push하여 팀원들과 변경 사항을 공유할 수 있습니다.
- commit을 자주 하여 작업의 중요한 시점마다 상태를 기록해 두면, 나중에 문제가 생겼을 때 이전 버전으로 쉽게 돌아갈 수 있습니다.

### 3. Push와 Pull: 원격 저장소와의 상호작용

- `git push`: 로컬에서 작업한 내용을 원격 저장소로 업로드합니다. push를 통해 원격 저장소에 반영되어 팀원들이 변경 사항을 확인할 수 있습니다.
- `git pull`: 원격 저장소에 있는 다른 팀원들의 변경 사항을 로컬로 가져와서 병합합니다. pull을 통해 팀 프로젝트에서 협업할 수 있게 됩니다.

## 4. Clone과 Commit의 실제 사용 시점

### 실제 개발 흐름

#### 1. 초기 작업:

- 프로젝트를 처음 로컬에 가져올 때 `git clone`을 사용합니다.
- 원격 저장소에 있는 프로젝트의 모든 커밋과 브랜치를 로컬에 복제합니다.

#### 2. 주기적 작업 기록:

- 개발을 진행하면서 코드를 변경한 후 `git commit`을 통해 변경 사항을 로컬에 기록합니다.
- 각 커밋은 프로젝트의 스냅샷처럼 중요한 작업 단위들을 기록해 둡니다.

#### 3. 팀 협업:

- 팀원들과 협업할 때는 변경 사항을 원격 저장소에 `git push`로 업로드합니다.
- 다른 팀원들이 push한 변경 사항을 로컬로 `git pull`하여 최신 상태를 유지합니다.

### 실제 작업 시 clone보다 commit이 더 많이 사용됨

- `git clone`은 프로젝트를 처음 복제할 때만 한 번 사용됩니다.
- 이후에는 `git commit`과 `git push`, `git pull`을 반복하면서 작업과 협업을 진행합니다.
- 개발 작업이 진행될수록 커밋 기록을 남기고, 버그 수정이나 이전 버전으로 돌아갈 때 커밋된 버전을 활용하는 일이 많습니다.

## 5. 커밋된 버전들로 되돌아가기

- 커밋된 버전들은 Git의 강력한 이점 중 하나로, 나중에 필요할 때 특정 커밋으로 돌아갈 수 있습니다.
- `git log` 명령어로 커밋 기록을 확인하고, 특정 커밋으로 돌아가고 싶을 때는 `git checkout`을 사용해 과거의 버전을 복원할 수 있습니다.
- 예시:

```
bash
git checkout <commit_hash>
```

- 혹은 `git revert`나 `git reset`을 사용하여 이전 커밋 상태로 되돌리거나, 특정 커밋을 취소할 수도 있습니다.

### 요약

- Clone:** 원격 저장소를 로컬로 복제할 때 사용하고, 처음 한 번만 실행합니다.
- Commit:** 로컬에서 작업한 내용을 버전별로 기록하는 과정으로, 자주 사용됩니다. 각 커밋은 프로젝트의 중요한 변경점이 됩니다.
- Push와 Pull:** 로컬 작업 내용을 원격 저장소에 반영하거나, 원격 저장소에서 최신 변경 사항을 가져옵니다.

따라서, 실제 로컬에서 작업을 할 때는 `git clone`은 처음 작업 환경을 구축할 때 한 번만 사용되고, 이후에는 커밋과 푸시/풀을 반복적으로 사용하면서 버전 관리를 하게 됩니다.

## 특정 폴더를 github에 올리는법

## git

공유하고자 하는 Vault 위치로 터미널을 이동한다

```
git init
git config --local user.name "깃허브 아이디 이름"
git config --local user.email "깃허브 이메일 이름"
```

## .gitignore 설정

필요한 .gitignore 설정

Vault 폴더 안에 `.gitignore` 파일을 만들어 다음과 같이 작성한다  
안드로이드와 컴퓨터 간의 플러그인 설정 파일 충돌을 막는다. [참고](#)

```
.obsidian/

.obsidian/workspace-mobile.json
.obsidian/workspace.json
.obsidian/app.json

.trash/
.DS_Store
```

## github에 올리기

공유하고자 하는 Vault 위치로 터미널을 이동한다

```
git add .
git commit -m "initial commit"
git branch -M main
git remote add origin "생성한 깃허브 주소"
git push -u origin main
```

```
git init
git config --local user.name "HAWONNA"
git config --local user.email "skgk4620@gmail.com"
```

```
git add .
```

```
git commit -m "24.09.12 commit"  
git branch -M main  
git remote add origin "https://github.com/HAWONNA/Finance.git"  
git push -u origin main
```