# 3. Git 기초 및 명령어 II

## 1. 되돌리기 명령어

Git 작업을 하다보면 모든 단계에서 어떤 것은 되돌려야 (Undo) 할 때가 있음.
한 번 되돌리면 복구할 수 없기에 주의.

### Commit 덮어쓰기

완료한 커밋을 수정해야 할 때 사용. 예)

- 너무 일찍 커밋했거나
- 어떤 파일을 빼먹었을 때
- 커밋 메시지를 잘못 적었을 때 한다.

다시 커밋 해야 하면 파일 수정 작업을 하고 Staging Area 에 추가한 다음 `--amend` 옵션을 사용하여 커밋을 재작성 할 수 있다.

```
$ git commit --amend
```

아래는 커밋을 했는데 Stage 하는 것을 깜빡하고 빠트린 파일이 있을 경우 예시이다.

```
$ echo 'hello' > init_file
$ echo 'hello' > forgotten_file
$ git add init_file
$ git commit -m 'initial commit'
[main 5964c67] initial commit
 1 file changed, 1 insertion(+)
 create mode 100644 init_file

$ git add forgotten_file

$ git commit --amend
[main a399df2] initial commit
 Date: Sat Mar 25 23:55:21 2023 +0000
 2 files changed, 2 insertions(+)
 create mode 100644 forgotten_file
 create mode 100644 init_file
```

여기서 실행한 명령어 3개는 모두 커밋 한 개로 기록된다. 두 번째 커밋은 첫 번째 커밋을 덮어쓴다.

| | |
|---|---|
| 노트 | 이렇게 `--amend` 옵션으로 커밋을 고치는 작업은, 추가로 작업한 일이 작다고 하더라도 이전의 커밋을 완전히 새로 고쳐서 새 커밋으로 변경하는 것을 의미한다. 이전의 커밋은 일어나지 않은 일이 되는 것이고 당연히 히스토리에도 남지 않는다.<br><br>`--amend` 옵션으로 커밋을 고치는 작업이 주는 장점은 마지막 커밋 작업에서 아주 살짝 뭔가 빠뜨린 것을 넣거나 변경하는 것을 새 커밋으로 분리하지 않고 하나의 커밋에서 처리하는 것이다. “앗차, 빠진 파일 넣었음”, “이전 커밋에서 오타 살짝 고침” 등의 커밋을 만들지 않겠다는 말이다. |

## 파일 상태를 Unstage 로 변경하기

실수로 Staging Area 에 들어간 파일을 워킹 디렉토리로 되돌리는 방법이다.

다음은 2가지 파일이 Staging Area 에 들어갔는데, 그중 `CONTRIBUTING.md` 는 커밋 대상이 아닌 상황이다.

```
$ git mv README README.md
$ echo 'reset test' > CONTRIBUTING.md
$ git add *
$ git status
On branch main
Your branch is ahead of 'origin/main' by 6 commits.
  (use "git push" to publish your local commits)

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   CONTRIBUTING.md
        renamed:    README -> README.md
```

`Changes to be commited` 밑에 `git restore --staged <file>…` 메시지가 보인다. 이 명령으로 Unstaged 상태로 변경할 수 있다.
`CONTRIBUTING.md` 파일을 Unstaged 상태로 변경해보자.

```
$ git restore --staged CONTRIBUTING.md
$ git status
On branch main
Your branch is ahead of 'origin/main' by 6 commits.
  (use "git push" to publish your local commits)

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        renamed:    README -> README.md

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   CONTRIBUTING.md
```

`CONTRIBUTING.md` 파일은 Unstaged 상태가 됐다.

---

## Modified 파일 되돌리기

어떻게 해야 `CONTRIBUTING.md` 파일을 수정하고 나서 다시 되돌릴 수 있을까?

그러니까 마지막으로 커밋된 버전으로 되돌리는 방법이 무얼까? 위에 있는 예제에서 Unstaged 부분을 보자.

```
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   CONTRIBUTING.md
```

위의 메시지는 수정한 파일을 되돌리는 방법을 꽤 정확하게 알려준다. 알려주는 대로 한 번 해보자.

```
$ git restore CONTRIBUTING.md
$ git status
On branch main
Your branch is ahead of 'origin/main' by 6 commits.
  (use "git push" to publish your local commits)

```

```
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        renamed:    README -> README.md
```

정상적으로 복원된 것을 알 수 있다.

- Git 으로 *커밋* 한 모든 것은 언제나 복구할 수 있다. 삭제한 브랜치에 있었던 것도, `--amend` 옵션으로 다시 커밋한 것도 복구할 수 있다.
- 하지만 커밋하지 않고 잃어버린 것은 되돌릴 수 없다.

---

# 2. 리모트 저장소

## 리모트 저장소

리모트 저장소를 관리할 줄 알아야 다른 사람과 함께 일할 수 있다.

- 리모트 저장소는 인터넷이나 네트워크 어딘가에 있는 저장소를 말한다.
- 저장소는 여러 개가 있을 수 있다.
- 다른 사람들과 함께 일한다는 것은 리모트 저장소를 관리하면서 데이터를 거기에 Push 하고 Pull 하는 것이다.
- 리모트 저장소를 관리한다는 것은 저장소를 추가, 삭제하는 것뿐만 아니라 브랜치를 관리하고 추적할지 말지 등을 관리하는 것을 말한다.

| 노트 | 원격 저장소라 하더라도 로컬 시스템에 위치할 수도 있다.<br><br>“remote” 저장소라고 이름이 붙어있어도 이 원격 저장소가 사실 같은 로컬 시스템에 존재할 수도 있다. 여기서 “remote” 라는 이름은 반드시 저장소가 네트워크나 인터넷을 통해 어딘가 멀리 떨어져 있어야만 한다는 것을 의미하지 않는다. 물론 일반적인 원격 저장소와 마찬가지로 Push, Pull 등의 기능은 동일하게 사용한다. |
|---|---|

## 리모트 저장소 확인하기

`git remote` 명령으로 현재 프로젝트에 등록된 리모트 저장소를 확인할 수 있다. 이 명령은 리모트 저장소의 단축 이름을 보여준다. 저장소를 Clone 하면 `origin` 이라는 리모트 저장소가 자동으로 등록되기 때문에 `origin` 이라는 이름을 볼 수 있다.

```
$ git remote
origin
```

`-v` 옵션을 주어 단축이름과 URL을 함께 볼 수 있다.

```
$ git remote -v
origin  https://github.com/SeungpilGcp1/day1 (fetch)
origin  https://github.com/SeungpilGcp1/day1 (push)
```

---

## 리모트 저장소에 Push 하기

프로젝트를 공유하고 싶을 때 Upstream 저장소에 Push 할 수 있다. 이 명령은 `git push <리모트 저장소 이름> <브랜치 이름>` 으로 단순하다. main 브랜치를 `origin` 서버에 Push 하려면(다시 말하지만 Clone 하면 보통 자동으로 origin 이름이 생성된다) 아래와 같이 서버에 Push 한다.

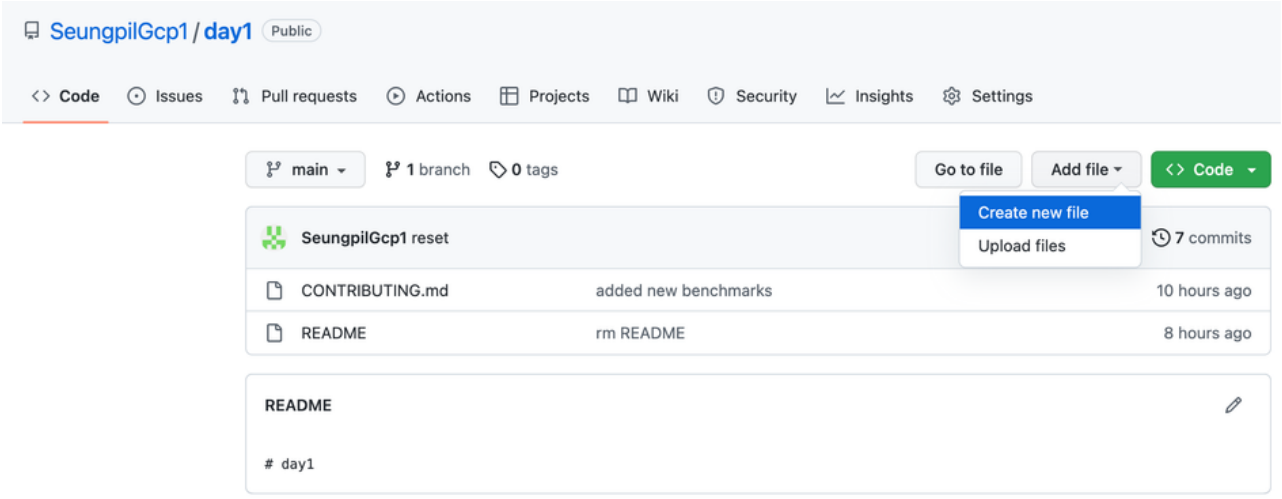
```
$ git push origin main

Enumerating objects: 17, done.
Counting objects: 100% (17/17), done.
Delta compression using up to 2 threads
Compressing objects: 100% (11/11), done.
Writing objects: 100% (16/16), 1.50 KiB | 769.00 KiB/s, done.
Total 16 (delta 3), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (3/3), done.
To https://github.com/SeungpilGcp1/day1
   928924c..2a082e9  main -> main
```

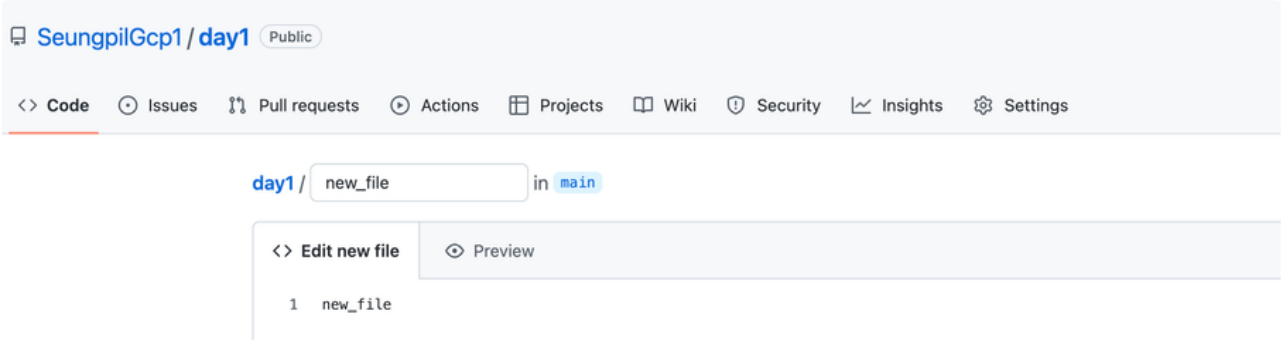
이 명령은 Clone 한 리모트 저장소에 쓰기 권한이 있고, Clone 하고 난 이후 아무도 Upstream 저장소에 Push 하지 않았을 때만 사용할 수 있다. 다시 말해서 Clone 한 사람이 여러 명 있을 때, 다른 사람이 Push 한 후에 Push 하려고 하면 Push 할 수 없다. 먼저 다른 사람이 작업한 것을 가져와서 Merge 한 후에 Push 할 수 있다.

---

## 리모트 저장소 Fetch

실습을 위해 리모트 저장소에 Github UI 를 통해서 임의의 데이터를 직접 변경한다.




앞서 설명했듯이 리모트 저장소에서 데이터를 가져오려면 간단히 아래와 같이 실행한다.

```
$ git fetch <remote>
```

이 명령은 로컬에는 없지만, 리모트 저장소에는 있는 데이터를 모두 가져온다. 그러면 리모트 저장소의 모든 브랜치를 로컬에서 접근할 수 있어서 언제든지 Merge를 하거나 내용을 살펴볼 수 있다.

저장소를 Clone 하면 명령은 자동으로 리모트 저장소를 “origin” 이라는 이름으로 추가한다. 그래서 나중에 `git fetch origin` 명령을 실행하면 Clone 한 이후에(혹은 마지막으로 가져온 이후에) 수정된 것을 모두 가져온다.

`git fetch` 명령은 리모트 저장소의 데이터를 모두 로컬로 가져오지만, 자동으로 Merge 하지 않는다.

### 리모트 저장소 Pull

`git pull` 명령으로 리모트 저장소 브랜치에서 데이터를 가져올 뿐만 아니라 자동으로 로컬 브랜치와 Merge 시킬 수 있다. 먼저 `git clone` 명령은 자동으로 로컬의 main 브랜치가 리모트 저장소의 main 브랜치를 추적하도록 한다.

그리고 `git pull` 명령은 Clone 한 서버에서 데이터를 가져오고 그 데이터를 자동으로 현재 작업하는 코드와 Merge 시킨다.

```
$ git pull origin
Updating 2a082e9..7f5ad25
Fast-forward
 new_file | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 new_file
```

---

### 리모트 저장소 살펴보기

`git remote show <리모트 저장소 이름>` 명령으로 리모트 저장소의 구체적인 정보를 확인할 수 있다. `origin` 같은 단축이름으로 이 명령을 실행하면 아래와 같은 정보를 볼 수 있다.

```
$ git remote show origin
* remote origin
  Fetch URL: https://github.com/SeungpilGcp1/day1
  Push  URL: https://github.com/SeungpilGcp1/day1
  HEAD branch: main
  Remote branch:
    main tracked
  Local branch configured for 'git pull':
    main merges with remote main
  Local ref configured for 'git push':
    main pushes to main (up to date)
```

리모트 저장소의 URL과 추적하는 브랜치를 출력한다. 이 명령은 `git pull` 명령을 실행할 때 main 브랜치와 Merge 할 브랜치가 무엇인지 보여 준다. `git pull` 명령은 리모트 저장소 브랜치의 데이터를 모두 가져오고 나서 자동으로 Merge 할 것이다. 그리고 가져온 모든 리모트 저장소 정보도 출력한다.

---

## 3. 태그

다른 VCS처럼 Git도 태그를 지원한다. 사람들은 보통 릴리즈할 때 사용한다(v1.0, 등등). 이번에는 태그를 조회하고 생성하는 법과 태그의 종류를 설명한다.

### 태그 조회하기

우선 `git tag` 명령으로 (`-l`, `--list` 는 옵션) 이미 만들어진 태그가 있는지 확인할 수 있다.

```
$ git tag
v1.0
v1.1
```

이 명령은 알파벳 순서로 태그를 보여준다.

검색 패턴을 사용하여 태그를 검색할 수 있다.

```
$ git tag -l "v1.*"
v1.0
v1.1
```

| 노트 | 와일드카드를 사용하여 Tag 리스트를 확인하려면 `-l`, `--list` 옵션을 지정<br><br>단순히 모든 Tag 목록을 확인하기 위해 `git tag` 명령을 실행했을 때 `-l` 또는 `--list` 옵션이 적용된 것과 동일한 결과가 출력된다.<br><br>하지만 와일드카드를 사용하여 태그 목록을 검색하는 경우에는 반드시 `-l` 또는 `--list` 옵션을 같이 써 줘야 원하는 결과를 얻을 수 있다. |
|---|---|

## 태그 붙이기

Git의 태그는 *Lightweight* 태그와 *Annotated* 태그로 두 종류가 있다.

- Lightweight 태그
  - 브랜치와 비슷한데 브랜치처럼 가리키는 지점을 최신 커밋으로 이동시키지 않는다. 단순히 특정 커밋에 대한 포인터일 뿐이다.
- Annotated 태그는
  - Git 데이터베이스에 태그를 만든 사람의 이름, 이메일과 태그를 만든 날짜, 그리고 태그 메시지도 저장한다.
  - GPG(GNU Privacy Guard)로 서명할 수도 있다.
  - 일반적으로 Annotated 태그를 만들어 이 모든 정보를 사용할 수 있도록 하는 것이 좋다. 하지만 임시로 생성하는 태그거나 이러한 정보를 유지할 필요가 없는 경우에는 Lightweight 태그를 사용할 수도 있다.

### Annotated 태그

Annotated 태그를 만드는 방법은 간단하다. `tag` 명령을 실행할 때 `-a` 옵션을 추가한다.

```
$ git tag -a v1.0 -m "my version 1.0"
$ git tag
v1.0
```

`-m` 옵션으로 태그를 저장할 때 메시지를 함께 저장할 수 있다. 명령을 실행할 때 메시지를 입력하지 않으면 Git은 편집기를 실행시킨다.

`git show` 명령으로 태그 정보와 커밋 정보를 모두 확인할 수 있다.

```
$ git show v1.0
tag v1.0
Tagger: SeungpilGcp1 <128713886+SeungpilGcp1@users.noreply.github.com>
Date:   Sun Mar 26 08:32:04 2023 +0000

my version 1.0

commit 7f5ad259c893ae0fa1c06e1b2dc6734c81c62018 (HEAD -> main, tag: v1.0, origin/main, origin/HEAD)
Author: SeungpilGcp1 <128713886+SeungpilGcp1@users.noreply.github.com>
Date:   Sun Mar 26 17:22:02 2023 +0900

    Create new_file

diff --git a/new_file b/new_file
```

```
new file mode 100644
index 0000000..d4fa860
--- /dev/null
+++ b/new_file
@@ -0,0 +1 @@
+new_file
```

커밋 정보를 보여주기 전에 먼저 태그를 만든 사람이 누구인지, 언제 태그를 만들었는지, 그리고 태그 메시지가 무엇인지 보여준다.

### Lightweight 태그

Lightweight 태그는 기본적으로 파일에 커밋 체크섬을 저장하는 것뿐이다. 다른 정보는 저장하지 않는다. Lightweight 태그를 만들 때는 `-a`, `-s`, `-m` 옵션을 사용하지 않는다. 이름만 달아줄 뿐이다.

```
$ echo 'new_file' > new_file2
$ git add new_file2
$ git commit -m 'add new file'
[main fe07710] add new file
 2 files changed, 1 insertion(+)
 rename README => README.md (100%)
 create mode 100644 new_file2

$ git tag v1.1
```

이 태그에 `git show` 를 실행하면 별도의 태그 정보를 확인할 수 없다. 이 명령은 단순히 커밋 정보만을 보여준다.

```
$ git show v1.1
commit fe07710f4df32f1d9e732774696ebb5116f59cde (HEAD -> main, tag: v1.1)
Author: SeungpilGcp1 <128713886+SeungpilGcp1@users.noreply.github.com>
Date:   Sun Mar 26 08:33:48 2023 +0000

    add new file

diff --git a/README b/README.md
similarity index 100%
rename from README
rename to README.md
diff --git a/new_file2 b/new_file2
new file mode 100644
index 0000000..d4fa860
--- /dev/null
+++ b/new_file2
@@ -0,0 +1 @@
+new_file
```

---

## 나중에 태그하기

예전 커밋에 대해서도 태그할 수 있다. 커밋 히스토리는 아래와 같다고 가정한다.

```
$ git log --pretty=oneline
fe07710f4df32f1d9e732774696ebb5116f59cde (HEAD -> main, tag: v1.1) add new file
7f5ad259c893ae0fa1c06e1b2dc6734c81c62018 (tag: v1.0, origin/main, origin/HEAD) Create new_file
2a082e94db3fbab78a00e8824ca01c5f73b89c5b reset
a399df21bf81cb40cef55ea3df831ec20f5187ed initial commit
359e6e8aeee9782e8dadedc9172dbe89dad8fe6d rm README
a39442e749d5782ba2c361ff9a2dfacbafd39b3b add PROJECTS.md
```

```
7f73ba02341270272c1d56b55bb265e1278acb23 added new benchmarks
73e66cf6402b8a94ad664e353574e78868afa81b Story 182: Fix benchmarks for speed
928924c5d14301c36729a7536117054c57a1eec8 Initial commit
```

특정 커밋에 태그하기 위해서 명령의 끝에 커밋 체크섬을 명시한다(긴 체크섬을 전부 사용할 필요는 없다).

```
$ git tag -a v0.9 928924c
```

이제 아래와 같이 만든 태그를 확인한다.

```
$ git tag
v0.9
v1.0
v1.1
```

---

### 태그 공유하기

`git push` 명령은 자동으로 리모트 서버에 태그를 전송하지 않는다. 태그를 만들었으면 서버에 별도로 Push 해야 한다. 브랜치를 공유하는 것과 같은 방법으로 할 수 있다. `git push origin <태그 이름>` 을 실행한다.

```
$ git push origin v0.9
Enumerating objects: 1, done.
Counting objects: 100% (1/1), done.
Writing objects: 100% (1/1), 177 bytes | 177.00 KiB/s, done.
Total 1 (delta 0), reused 0 (delta 0), pack-reused 0
To https://github.com/SeungpilGcp1/day1
 * [new tag]         v0.9 -> v0.9
```

만약 한 번에 태그를 여러 개 Push 하고 싶으면 `--tags` 옵션을 추가하여 `git push` 명령을 실행한다. 이 명령으로 리모트 서버에 없는 태그를 모두 전송할 수 있다.

```
$ git push origin --tags
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 2 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 436 bytes | 436.00 KiB/s, done.
Total 3 (delta 1), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
To https://github.com/SeungpilGcp1/day1
 * [new tag]         v1.0 -> v1.0
 * [new tag]         v1.1 -> v1.1
```

이제 누군가 저장소에서 Clone 하거나 Pull을 하면 모든 태그 정보도 함께 전송된다.

---

## 태그를 Checkout 하기

예를 들어 태그가 특정 버전을 가리키고 있고, 특정 버전의 파일을 체크아웃 해서 확인하고 싶다면 다음과 같이 실행한다.

단 태그를 체크아웃하면 “detached HEAD”(떨어져나온 HEAD) 상태가 되며 일부 Git 관련 작업이 브랜치에서 작업하는 것과 다르게 동작할 수 있다.

```
$ git checkout v0.9
Note: switching to 'v0.9'.

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:

  git switch -

Turn off this advice by setting config variable advice.detachedHead to false

HEAD is now at 928924c Initial commit
```

---

## 4. 파일 무시하기 (gitignore)

`.gitignore` 파일은 Git에서 파일이나 디렉터리를 무시하도록 설정할 수 있는 파일. 이 파일에 지정된 패턴과 일치하는 파일이나 디렉터리는 Git 저장소에 추가되지 않는다.

### 파일 생성 및 기본 사용법

1. 프로젝트 디렉터리에서 `.gitignore` 파일을 생성합니다.
2. 파일을 열고 무시하고 싶은 파일이나 디렉터리의 이름을 입력합니다.
   - 예: `node_modules/` (디렉터리 무시), `*.log` (모든 .log 파일 무시)

### 파일 예제

```
# Node.js 관련
node_modules/
npm-debug.log*

# macOS 관련
.DS_Store
.AppleDouble

# Windows 관련
Thumbs.db
ehthumbs.db

# IntelliJ 관련
.idea/
*.iml

# VSCode 관련
.vscode/
```

위 예제는 Node.js, macOS, Windows, IntelliJ, 그리고 VSCode와 관련된 일반적인 파일과 디렉터리를 무시하기 위한 `.gitignore` 파일의 내용을 보여줍니다.

## 주의사항

- 이미 저장소에 커밋된 파일은 `.gitignore`에 추가한다고 해서 자동으로 삭제되지 않습니다. 이러한 파일을 삭제하려면 별도의 Git 명령어를 사용해야 합니다.
- `.gitignore` 파일에는 정규 표현식을 사용할 수 없지만, 간단한 패턴 매칭을 사용할 수 있습니다.