

## 1-2 Git 기초 및 명령어 실습 I

### Github 가입

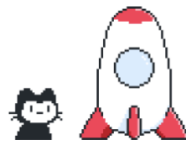
강의 시작 전에 [Github 가입](#) 을 완료 해 주세요.

[가입시 이메일 인증이 있으므로 정확한 이메일을 가입 해 주세요.]

[가입시 이메일로 인증코드 가입]



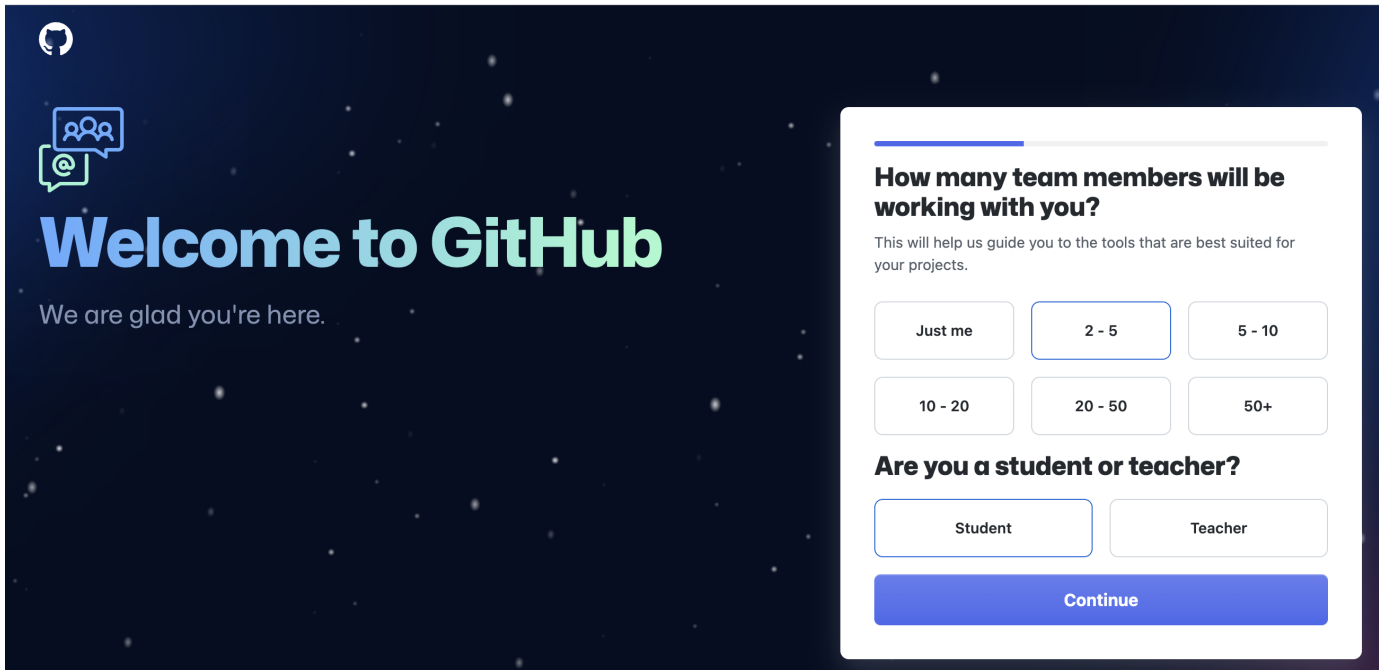
Here's your GitHub launch code, @SeungpilGcp1!



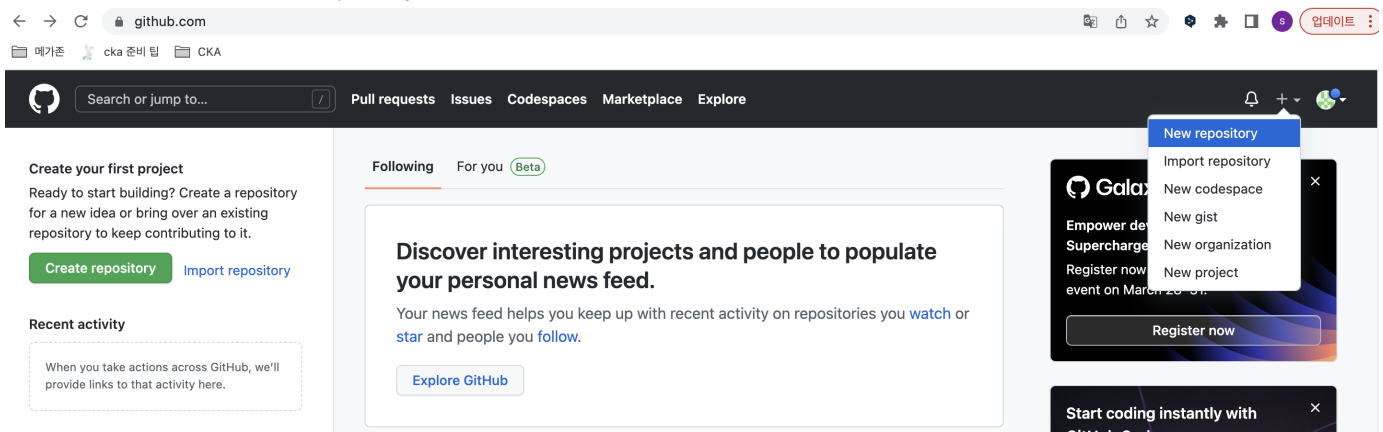
Continue signing up for GitHub by entering the code below:

97616101

Open GitHub



[가입 후 우측 상단 + 클릭 후 New repository]




[신규 레파지토리 생성]

- 레파지토리 이름 기입
- Add a README 체크박스

## Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? [Import a repository](#).

Owner \*

 SeungpilGcp1 ▾

Repository name \*

day1 ✓

Great repository names are short and memorable. Need inspiration? How about [automatic-adventure?](#)

Description (optional)



Public

Anyone on the internet can see this repository. You choose who can commit.



Private

You choose who can see and commit to this repository.

Initialize this repository with:

~~Skip this step if you're importing an existing repository.~~



Add a README file

This is where you can write a long description for your project. [Learn more](#).

Add .gitignore

Choose which files not to track from a list of templates. [Learn more](#).

.gitignore template: None ▾

Choose a license

A license tells others what they can and can't do with your code. [Learn more](#).

License: None ▾

This will set  main as the default branch. Change the default name in your [settings](#).

 You are creating a public repository in your personal account.

Create repository

- 만약 Add a README 체크 박스 하지 않았을 시에는 아무런 파일도 생성되지 않은 레파지토리 이기 때문에 초기 브랜치가 없습니다. 이 상태에서는 codespace 를 실행할 수 없으니, 그럴 경우 아래 영역을 클릭하셔서 파일을 추가 해 주세요.

Quick setup — if you've done this kind of thing before

Set up in Desktop or HTTPS SSH <https://github.com/SeungpilGcp1/day1.git>

Get started by [creating a new file](#) or [uploading an existing file](#). We recommend every repository include **a README, LICENSE, and .gitignore.**

...or create a new repository on the command line

```
echo "# day1" >> README.md
git init
git add README.md
git commit -m "first commit"
git branch -M main
git remote add origin https://github.com/SeungpilGcp1/day1.git
git push -u origin main
```

...or push an existing repository from the command line

```
git remote add origin https://github.com/SeungpilGcp1/day1.git
git branch -M main
git push -u origin main
```

- 레파지토리 생성 후에 아래 그림을 따라 Create codespace on main 클릭

main 1 branch 0 tags

Go to file Add file <> Code

Codespaces (New)

Codespaces

Your workspaces in the cloud

No codespaces

You don't have any codespaces with this repository checked out

Create codespace on main

Learn more about codespaces...

Codespace usage for this repository is paid for by SeungpilGcp1

About

No description, website, or topics provided.

Readme

0 stars

1 watching

0 forks

Releases

No releases published

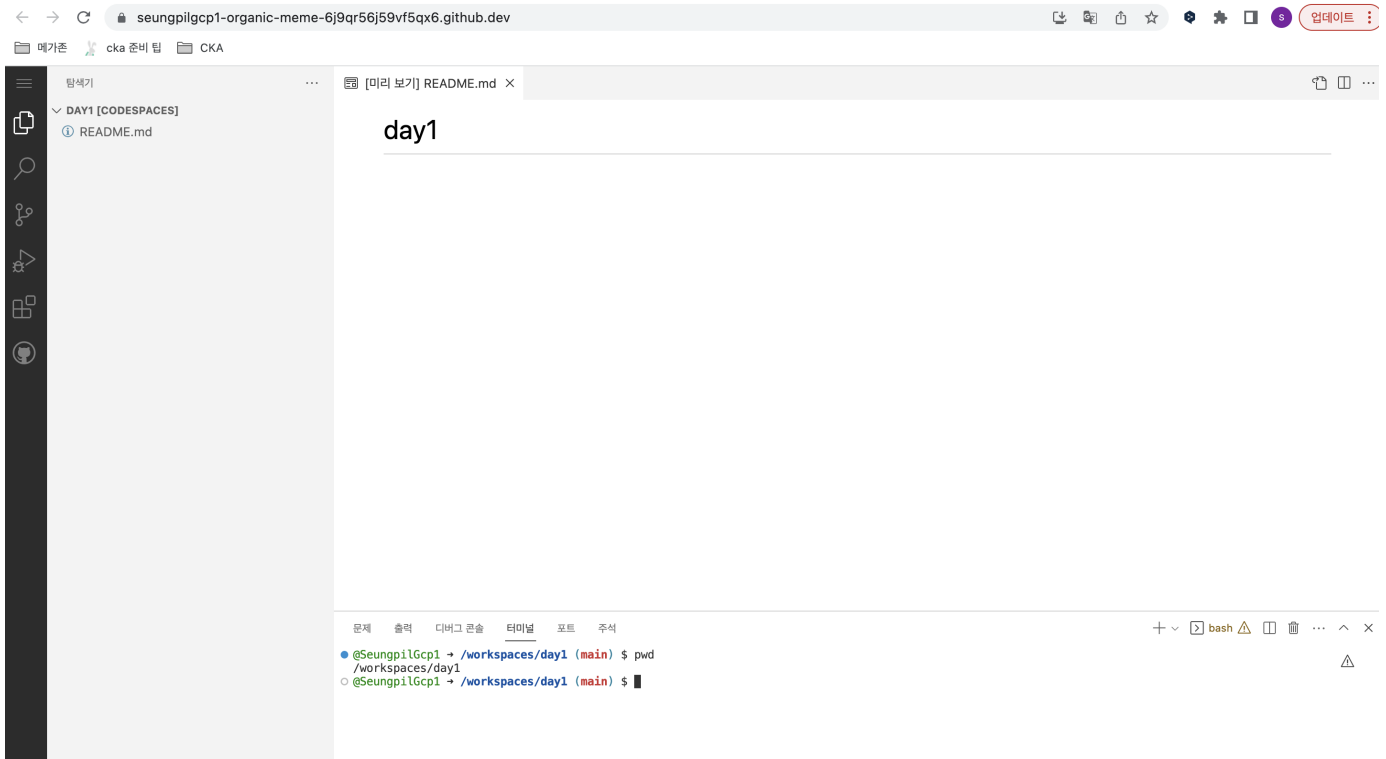
Create a new release

Packages

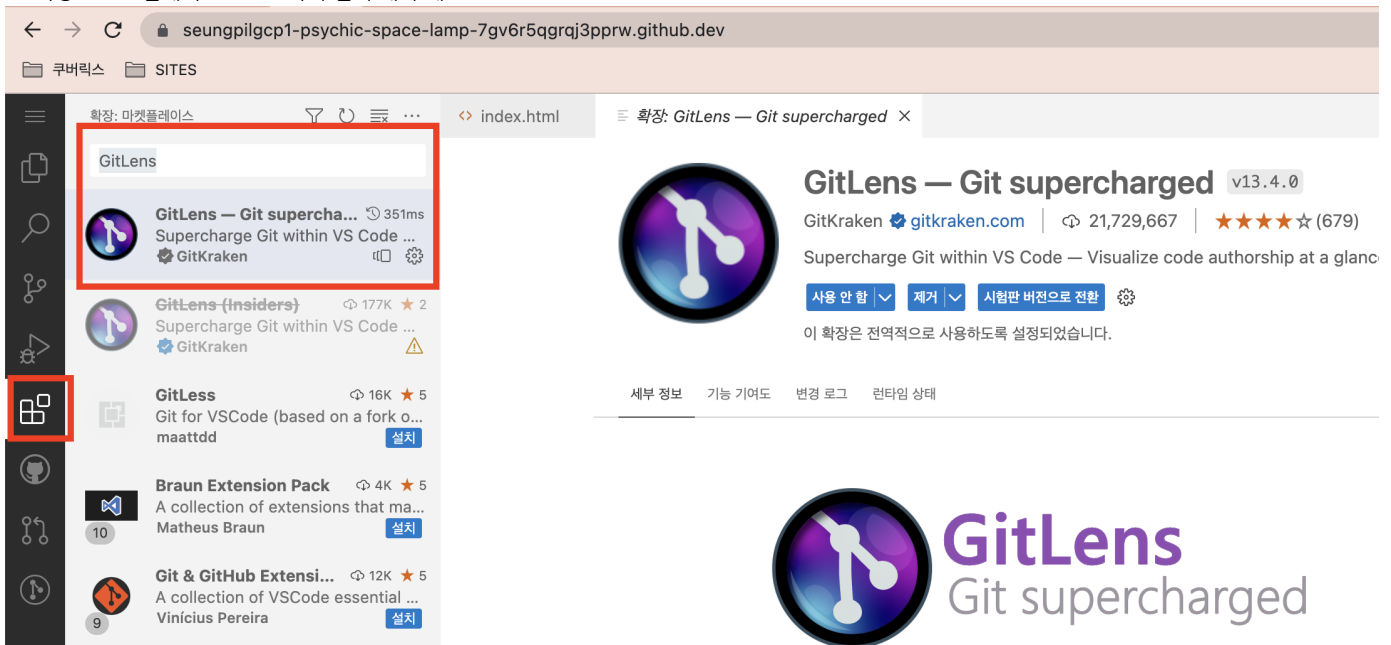
No packages published

Publish your first package

- 잠시 기다린 후 아래처럼 웹 터미널이 열리면 실습 환경 구성이 완료되었습니다.



- 확장 프로그램에서 GitLens 까지 설치 해 주세요.



## 2.1 Git의 기초 - Git 저장소 만들기

### Git 저장소 만들기

주로 다음 두 가지 중 한 가지 방법으로 Git 저장소를 쓰기 시작합니다.

1. 아직 버전관리를 하지 않는 기존 디렉토리에서 Git 저장소를 적용하는 방법
2. 기존의 Git 저장소를 Clone 하는 방법

### 기존 디렉토리를 Git 저장소로 만들기

기존 프로젝트를 Git 으로 관리하고 싶은 경우 우선 프로젝트의 디렉토리로 이동한다.

```
$ mkdir /workspaces/new-project
$ cd /workspaces/new-project
```

그리고 아래와 같은 명령을 실행한다:

```
$ git init
```

이 명령은 .git 이라는 하위 디렉토리를 만든다. .git 디렉토리에는 저장소에 필요한 뼈대 파일(Skeleton)이 들어 있다. 이 명령만으로는 아직 프로젝트의 어떤 파일도 관리하지 않는다. (.git 디렉토리가 막 만들어진 직후에 정확히 어떤 파일이 있는지에 대한 내용은 [Git의 내부](#)에서 다룬다)

Git 이 파일을 관리하게 하려면 저장소에 파일을 추가하고 커밋해야 한다. git add 명령으로 파일을 추가하고 git commit 명령으로 커밋한다:

```
$ vim LICENSE
$ git add LICENSE
$ git commit -m 'initial project version'
```

명령어 몇 개로 Git 저장소를 만들고 파일 버전 관리를 시작했다.

### 기존 저장소를 Clone 하기

다른 프로젝트에 참여 하려거나(Contribute) Git 저장소를 복사하고 싶을 때 git clone 명령을 사용한다.

- 이미 Subversion 같은 VCS에 익숙한 사용자에게는 "checkout" 이 아니라 "clone" 이라는 점
- Git이 Subversion과 다른 가장 큰 차이점은 서버에 있는 거의 모든 데이터를 복사.

git clone 을 실행하면 프로젝트 히스토리를 전부 받아온다. 실제로 서버의 디스크가 망가져도 클라이언트 저장소 중에서 아무거나 하나 가져다가 복구하면 된다 (서버에만 적용했던 설정은 복구하지 못하지만 모든 데이터는 복구된다)

git clone <url> 명령으로 저장소를 Clone 한다. libgit2 라이브러리 소스코드를 Clone 하려면 아래와 같이 실행한다.

```
$ cd /workspaces
$ git clone https://github.com/libgit2/libgit2
```

이 명령은 "libgit2" 라는 디렉토리를 만들고 그 안에 .git 디렉토리를 만든다. 그리고 저장소의 데이터를 모두 가져와서 자동으로 가장 최신 버전을 Checkout 해 놓는다. libgit2 디렉토리로 이동하면 Checkout으로 생성한 파일을 볼 수 있고 당장 하고자 하는 일을 시작할 수 있다.

아래와 같은 명령을 사용하여 저장소를 Clone 하면 libgit2 이 아니라 다른 디렉토리 이름으로 Clone 할 수 있다.

```
$ cd /workspaces
$ git clone https://github.com/libgit2/libgit2 mylibgit
```

디렉토리 이름이 mylibgit 이라는 것만 빼면 이 명령의 결과와 앞선 명령의 결과는 같다.

Git은 다양한 프로토콜을 지원한다. 이제까지는 https:// 프로토콜을 사용했지만 git:// 를 사용할 수도 있고 user@server:path/to/repo.git 처럼 SSH 프로토콜을 사용할 수도 있다.

## 2.2 Git의 기초 - 수정하고 저장소에 저장하기

### 수정하고 저장소에 저장하기

만질 수 있는 Git 저장소를 하나 만들었고 워킹 디렉토리에 Checkout도 했다. 이제는 파일을 수정하고 파일의 스냅샷을 커밋해 보자. 파일을 수정하다가 저장하고 싶으면 스냅샷을 커밋한다.

워킹 디렉토리의 모든 파일은 크게 Tracked(관리대상임)와 Untracked(관리대상이 아님)로 나눈다.

- Tracked 파일
  - 이미 스냅샷에 포함돼 있던 파일이다.
  - 간단히 말하자면 Git이 알고 있는 파일이라는 것이다.
  - Tracked 파일은 또 Unmodified(수정하지 않음)와 Modified(수정함) 그리고 Staged(커밋으로 저장소에 기록할) 상태 중 하나이다.
- 나머지 파일은 모두 Untracked 파일이다.
  - Untracked 파일은 워킹 디렉토리에 있는 파일 중 스냅샷에도 Staging Area에도 포함되지 않은 파일.

처음 저장소를 Clone 하면 모든 파일은 Tracked이면서 Unmodified 상태이다. Clone 을 하였다는 것은 이미 git 이 알고있는 파일(Tracked) 을 내려받았다는 것이고, 그 이후 아무것도 수정하지 않았기(Unmodified) 때문에 그렇다.

마지막 커밋 이후 아직 아무것도 수정하지 않은 상태에서 어떤 파일을 수정하면 Git은 그 파일을 **Modified** 상태로 인식한다. 실제로 커밋을 하기 위해서는 이 수정한 파일을 Staged 상태로 만들고, Staged 상태의 파일을 커밋한다. 이런 라이프사이클을 계속 반복한다.

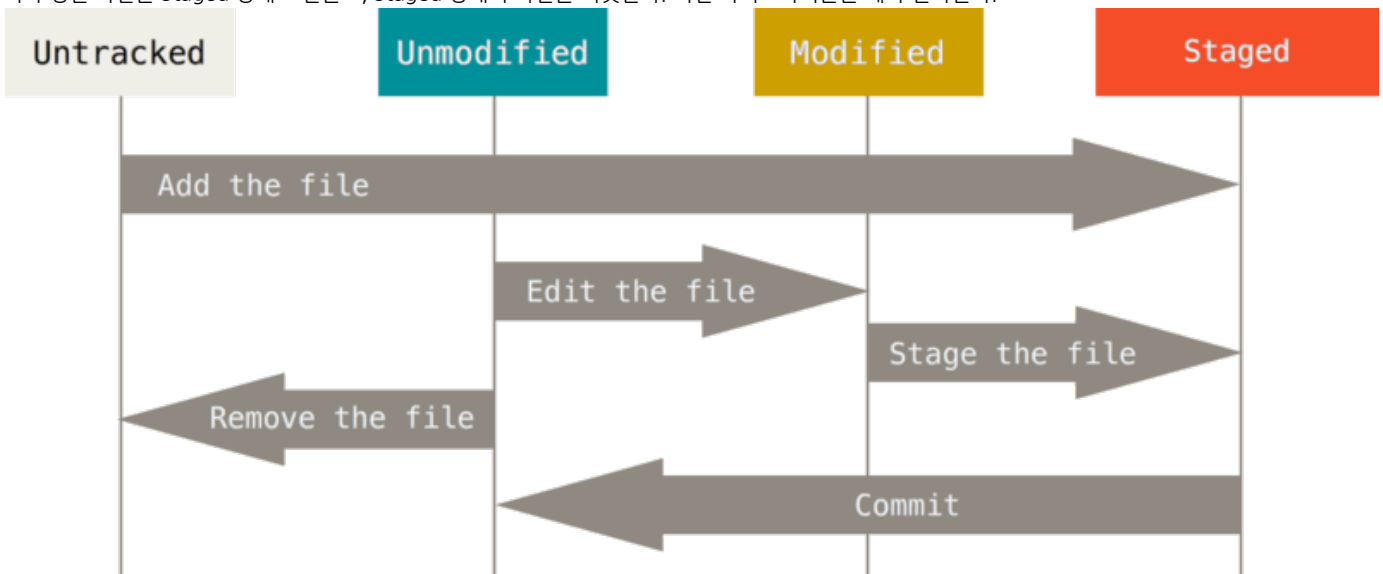


그림 8. 파일의 라이프사이클.

### 파일의 상태 확인하기

파일의 상태를 확인하려면 보통 `git status` 명령을 사용한다. Clone 한 후에 바로 이 명령을 실행하면 아래와 같은 메시지를 볼 수 있다.

```
$ cd /workspaces/day1
$ git status
On branch main
Your branch is up-to-date with 'origin/main'.
nothing to commit, working directory clean
```

위의 내용은 파일을 하나도 수정하지 않았다는 것을 말해준다. Tracked 파일은 하나도 수정되지 않았다는 의미다. Untracked 파일은 아직 없어서 목록에 나타나지 않는다. 그리고 현재 작업 중인 브랜치를 알려주며 서버의 같은 브랜치로부터 진행된 작업이 없는 것을 나타낸다. 기본 브랜치가 main이기 때문에 현재 브랜치 이름이 "main"로 나온다.

프로젝트에 README 파일을 만들어보자. README 파일은 새로 만든 파일이기 때문에 `git status`를 실행하면 'Untracked files'에 들어 있다:

```
$ echo 'My Project' > README
$ git status
On branch main
Your branch is up to date with 'origin/main'.

Untracked files:
  (use "git add <file>..." to include in what will be committed)
      README

nothing added to commit but untracked files present (use "git add" to track)
```

README 파일은 “Untracked files” 부분에 속해 있는데 이것은 README 파일이 Untracked 상태라는 것을 말한다.

Git은 Untracked 파일을 아직 스냅샷(커밋)에 넣어지지 않은 파일이라고 본다. 파일이 Tracked 상태가 되기 전까지는 Git은 그 파일을 커밋하지 않는다. README 파일을 add 하여 직접 Tracked 상태로 만들어 보자.

### 파일을 새로 추적하기

git add 명령으로 파일을 새로 추적할 수 있다. 아래 명령을 실행하면 Git은 README 파일을 추적한다.

```
$ git add README
```

git status 명령을 다시 실행하면 README 파일이 Tracked 상태이면서 커밋에 추가될 Staged 상태라는 것을 확인할 수 있다.

```
$ git status
On branch main
Your branch is up to date with 'origin/main'.

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

“Changes to be committed”에 들어 있는 파일은 Staged 상태라는 것을 의미한다.

- git add
  - git 을 커밋하면, git add 를 실행한 시점의 파일이 커밋되어 저장소 히스토리에 남는다.
  - 이 명령을 통해 디렉토리에 있는 파일을 추적하고 관리하도록 한다.
  - 파일 또는 디렉토리의 경로를 아규먼트로 받는다.
  - 디렉토리라면 아래에 있는 모든 파일들까지 재귀적으로 추가한다.

### Modified 상태의 파일을 Stage 하기

이미 Tracked 상태인 파일을 수정하는 법을 알아보자. CONTRIBUTING.md 라는 파일을 수정하고 나서 git status 명령을 다시 실행하면 결과는 아래와 같다.



```
$ git status
On branch main
Your branch is up to date with 'origin/main'.

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

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
```

이 CONTRIBUTING.md 파일은 “Changes not staged for commit”에 있다. 이것은 수정한 파일이 Tracked 상태이지만 아직 Staged 상태는 아니라는 것이다.

Staged 상태로 만들려면 git add 명령을 실행해야 한다. git add 명령은 파일을 새로 추적할 때도 사용하고 수정한 파일을 Staged 상태로 만들 때도 사용한다. Merge 할 때 충돌난 상태의 파일을 Resolve 상태로 만들 때도 사용한다.

add의 의미는 프로젝트에 파일을 추가한다기 보다는 다음 커밋에 추가한다고 받아들이는게 좋다. git add 명령을 실행하여 CONTRIBUTING.md 파일을 Staged 상태로 만들고 git status 명령으로 결과를 확인해보자.

```
$ git add CONTRIBUTING.md
$ git status
On branch main
Your branch is up to date with 'origin/main'.

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

두 파일 모두 Staged 상태이므로 다음 커밋에 포함된다. 하지만 아직 더 수정해야 한다는 것을 알게 되어 바로 커밋하지 못하는 상황이 되었다고 생각해보자. 이 상황에서 CONTRIBUTING.md 파일을 열고 수정한다. 이제 커밋할 준비가 다 됐다고 생각할 테지만, Git은 그렇지 않다. git status 명령으로 파일의 상태를 다시 확인해보자.

```
$ vim CONTRIBUTING.md
$ git status
On branch main
Your branch is up to date with 'origin/main'.

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

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 가 Staged 상태이면서 동시에 Unstaged 상태로 나온다.

git add 명령을 실행하면 Git은 파일을 바로 Staged 상태로 만든다. 지금 이 시점에서 커밋을 하면 git commit 명령을 실행하는 시점의 버전이 커밋 되는 것이 아니라 마지막으로 git add 명령을 실행했을 때의 버전이 커밋된다.

그러니까 git add 명령을 실행한 후에 또 파일을 수정하면 git add 명령을 다시 실행해서 최신 버전을 Staged 상태로 만들어야 한다.

```
$ git add CONTRIBUTING.md
$ git status
On branch main
Your branch is up to date with 'origin/main'.

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

#### 파일 상태를 짧막하게 확인하기

git status 명령으로 확인할 수 있는 내용이 좀 많아 보일 수 있다. 사실 그렇다. 좀 더 간단하게 변경 내용을 보여주는 옵션이 있다. git status -s 또는 git status --short 처럼 옵션을 주면 현재 변경한 상태를 짧막하게 보여준다.

```

# FILE1, FILE2
$ echo 'new' > FILE1
$ echo 'new' > FILE2

# FILE2 Staged
$ git add FILE2

# FILE2
$ echo 'changed again' > FILE2

#
$ git status -s
A  CONTRIBUTING.md
AM FILE2
A  README
?? FILE1

```

아직 추적하지 않는 새 파일 앞에는 ?? 표시가 붙는다. Staged 상태로 추가한 파일 중 새로 생성한 파일 앞에는 A 표시가, 수정한 파일 앞에는 M 표시가 붙는다. 위 명령의 결과에서 상태정보 컬럼에는 두 가지 정보를 보여준다. 왼쪽에는 Staging Area 에서의 상태를, 오른쪽에는 Working Tree 에서의 상태를 표시한다.

## 파일 무시하기

어떤 파일은 Git 이 관리할 필요가 없다. 보통 로그 파일이나 빌드 시스템이 자동으로 생성한 파일이 그렇다. 그런 파일을 무시하려면 .gitignore 파일을 만들고 그 안에 무시할 파일 패턴을 적는다. 아래는 .gitignore 파일의 예이다.

```

$ cat .gitignore
*. [oa]
*~

```

첫번째 라인은 확장자가 “.o” 나 “.a” 인 파일을 Git이 무시하라는 것이고 둘째 라인은 ~ 로 끝나는 모든 파일을 무시하라는 것이다. 보통 대부분의 텍스트 편집기에서 임시파일로 사용하는 파일 이름이기 때문이다. “.o” 와 “.a” 는 각각 빌드 시스템이 만들어내는 오브젝트와 아카이브 파일이고 ~ 로 끝나는 파일은 Emacs나 VI 같은 텍스트 편집기가 임시로 만들어내는 파일이다. 또 log, tmp, pid 같은 디렉토리나, 자동으로 생성하는 문서 같은 것들도 추가할 수 있다. .gitignore 파일은 보통 처음에 만들어 두는 것이 편리하다. 그래서 Git 저장소에 커밋하고 싶지 않은 파일을 실수로 커밋하는 일을 방지할 수 있다.

.gitignore 파일에 입력하는 패턴은 아래 규칙을 따른다.

- 아무것도 없는 라인이나, # 로 시작하는 라인은 무시한다.
- 표준 Glob 패턴을 사용한다. 이는 프로젝트 전체에 적용된다.
- 슬래시(/)로 시작하면 하위 디렉토리에 적용되지(Recursivity) 않는다.
- 디렉토리는 슬래시(/)를 끝에 사용하는 것으로 표현한다.
- 느낌표(!)로 시작하는 패턴의 파일은 무시하지 않는다.

Glob 패턴은 정규표현식을 단순하게 만든 것으로 생각하면 되고 보통 셸에서 많이 사용한다. 애스터리스크(\*)는 문자가 하나도 없거나 하나 이상을 의미하고, [abc] 는 중괄호 안에 있는 문자 중 하나를 의미한다(그러니까 이 경우에는 a, b, c). 물음표(?)는 문자 하나를 말하고, [0-9] 처럼 중괄호 안의 캐릭터 사이에 하이픈(-)을 사용하면 그 캐릭터 사이에 있는 문자 하나를 말한다. 애스터리스크 2개를 사용하여 디렉토리 안의 디렉토리 까지 지정할 수 있다. a/\*\*/z 패턴은 a/z, a/b/z, a/b/c/z 디렉토리에 사용할 수 있다.

아래는 .gitignore 파일의 예이다.

```
# .a
*.a

# .a lib.a
!lib.a

# TODO subdir/TODO
/TODO

# build/
build/

# doc/notes.txt doc/server/arch.txt
doc/*.txt

# doc *.pdf
doc/**/*.pdf
```

#### 힌트

GitHub은 다양한 프로젝트에서 자주 사용하는 .gitignore 예제를 관리하고 있다. 어떤 내용을 넣을지 막막하다면 <https://github.com/github/gitignore> 사이트에서 적당한 예제를 찾을 수 있다.

#### 노트

.gitignore 를 사용하는 간단한 방식은 하나의 .gitignore 파일을 최상위 디렉토리에 하나 두고 모든 하위 디렉토리에까지 적용시키는 방식이다. 물론 .gitignore 파일을 하나만 두는 것이 아니라 하위 디렉토리에 도 추가로 둘 수도 있다. .gitignore 정책은 현재 .gitignore 파일이 위치한 디렉토리와 그 하위 디렉토리에 적용된다. (리눅스 커널 소스 저장소에는 .gitignore 파일이 206개나 있음)

다수의 .gitignore 파일을 두고 정책을 적용하는 부분은 이 책에서 다루는 범위를 벗어난다. 자세한 내용은 `man gitignore` 에서 확인할 수 있다.

### Staged와 Unstaged 상태의 변경 내용을 보기

단순히 파일이 변경됐다는 사실이 아니라 어떤 내용이 변경됐는지 살펴보려면 `git status` 명령이 아니라 `git diff` 명령을 사용해야 한다. 보통 우리는 '수정했지만, 아직 Staged 파일이 아닌 것?'과 '어떤 파일이 Staged 상태인지?'가 궁금하기 때문에 `git status` 명령으로도 충분하다.

더 자세하게 볼 때는 `git diff` 명령을 사용하는데 Patch처럼 어떤 라인을 추가했고 삭제했는지가 궁금할 때 사용한다.

README 파일을 수정해서 Staged 상태로 만들고 CONTRIBUTING.md 파일은 그냥 수정만 해둔다. 이 상태에서 `git status` 명령을 실행하면 아래와 같은 메시지를 볼 수 있다.

```
$ echo 'changed again again' > CONTRIBUTING.md
$ git status
On branch main
Your branch is up to date with 'origin/main'.

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

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
```

git diff 명령을 실행하면 수정했지만 아직 staged 상태가 아닌 파일을 비교해 볼 수 있다.

```
$ git diff
diff --git a/CONTRIBUTING.md b/CONTRIBUTING.md
index 5f4cf76..fbb7ab1 100644
--- a/CONTRIBUTING.md
+++ b/CONTRIBUTING.md
@@ -1,1 @@
-changed again
\ No newline at end of file
+changed again again
```

이 명령은 워킹 디렉토리에 있는 것과 Staging Area에 있는 것을 비교한다. 그래서 수정하고 아직 Stage 하지 않은 것을 보여준다.

만약 커밋하려고 Staging Area에 넣은 파일의 변경 부분을 보고 싶으면 git diff --staged 옵션을 사용한다. 이 명령은 저장소에 커밋한 것과 Staging Area에 있는 것을 비교한다.

```
$ git diff --staged
diff --git a/CONTRIBUTING.md b/CONTRIBUTING.md
new file mode 100644
index 0000000..5f4cf76
--- /dev/null
+++ b/CONTRIBUTING.md
@@ -0,0 +1 @@
+changed again
\ No newline at end of file
diff --git a/README b/README
new file mode 100644
index 0000000..56266d3
--- /dev/null
+++ b/README
@@ -0,0 +1 @@
+My Project
```

꼭 잊지 말아야 할 것이 있는데 git diff 명령은 마지막으로 커밋한 후에 수정한 것들 전부를 보여주지 않는다. git diff 는 Unstaged 상태인 것들만 보여준다. 수정한 파일을 모두 Staging Area에 넣었다면 git diff 명령은 아무것도 출력하지 않는다.

노트	<p>외부 도구로 비교하기</p> <p>이 책에서는 계속 git diff 명령으로 여기저기서 써 먹는다. 즐겨 쓰거나 결과를 아름답게 보여주는 Diff 도구가 있으면 사용할 수 있다. git diff 대신 git difftool 명령을 사용해서 emerge, vimdiff 같은 도구로 비교할 수 있다. 상용 제품도 사용할 수 있다. git difftool --tool-help 라는 명령은 사용가능한 도구를 보여준다.</p>
----	--

## 변경사항 커밋하기

수정한 것을 커밋하기 위해 Staging Area에 파일을 정리했다. (git add 를 통해서)

Unstaged 상태의 파일은 커밋되지 않는다는 것을 기억해야 한다. 커밋하기 전에 git status 명령으로 모든 것이 Staged 상태인지 확인할 수 있다. 그 후에 git commit 을 실행하여 커밋한다.

```
$ git status
$ git add CONTRIBUTING.md
$ git commit
```

Git 설정에 지정된 편집기가 실행되고, 아래와 같은 텍스트가 자동으로 포함된다 (아래 예제는 Vim 편집기의 화면이다. 이 편집기는 쉘의 EDITOR 환경 변수에 등록된 편집기이고 보통은 Vim이나 Emacs를 사용한다. 또 [시작하기](#) 에서 설명했듯이 git config --global core.editor 명령으로 어떤 편집기를 사용할지 설정할 수 있다).

편집기는 아래와 같은 내용을 표시한다(아래 예제는 Vim 편집기).

```
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
#
# On branch main
# Your branch is up to date with 'origin/main'.
#
# Changes to be committed:
#       new file:   CONTRIBUTING.md
#       new file:   README
#
```

- 자동으로 생성되는 커밋 메시지의 첫 라인은 비어 있고 둘째 라인부터 `git status` 명령의 결과가 채워진다.
- 커밋한 내용을 쉽게 기억할 수 있도록 이 메시지를 포함할 수도 있고 메시지를 전부 지우고 새로 작성할 수 있다
- 정확히 뭘 수정했는지도 보여줄 수 있는데, `git commit` 에 `-v` 옵션을 추가하면 편집기에 diff 메시지도 추가된다.
- 내용을 저장하고 편집기를 종료하면 Git은 입력된 내용(#로 시작하는 내용을 제외한)으로 새 커밋을 하나 완성한다.

메시지를 인라인으로 첨부할 수도 있다. `commit` 명령을 실행할 때 아래와 같이 `-m` 옵션을 사용한다.

```
$ git commit -m "Story 182: Fix benchmarks for speed"
[main 73e66cf] Story 182: Fix benchmarks for speed
 2 files changed, 2 insertions(+)
 create mode 100644 CONTRIBUTING.md
 create mode 100644 README
```

이렇게 첫번째 커밋을 작성해보았다. `commit` 명령은 몇 가지 정보를 출력하는데 위 예제는 (main) 브랜치에 커밋했고 체크섬은 (463dc4f)이라고 알려 준다. 그리고 수정한 파일이 몇 개이고 삭제됐거나 추가된 라인이 몇 라인인지 알려준다.

Git은 Staging Area에 속한 스냅샷을 커밋한다는 것을 기억해야 한다. 수정은 했지만, 아직 Staging Area에 넣지 않은 것은 다음에 커밋할 수 있다. 커밋 할 때마다 프로젝트의 스냅샷을 기록하기 때문에 나중에 스냅샷끼리 비교하거나 예전 스냅샷으로 되돌릴 수 있다.

### Staging Area 생략하기

Staging Area는 커밋할 파일을 정리한다는 점에서 매우 유용하지만 복잡하기만 하고 필요하지 않은 때도 있다. 아주 쉽게 Staging Area를 생략할 수 있다.

`git commit` 명령을 실행할 때 `-a` 옵션을 추가하면 Git은 Tracked 상태의 파일을 자동으로 Staging Area에 넣는다. 그래서 `git add` 명령을 실행하는 수고를 덜 수 있다.

[단, Untracked file, 워킹 디렉터리에서 신규로 추가하거나 한 파일은 해당 명령어로 자동으로 Staging Area에 들어가지 않는다.]

```

$ echo 'changed again again again' > CONTRIBUTING.md
$ git status
On branch main
Your branch is ahead of 'origin/main' by 1 commit.
  (use "git push" to publish your local commits)

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

no changes added to commit (use "git add" and/or "git commit -a")
$ git commit -a -m 'added new benchmarks'
[main 7f73ba0] added new benchmarks
 1 file changed, 1 insertion(+), 1 deletion(-)

```

이 예제에서는 커밋하기 전에 `git add` 명령으로 `CONTRIBUTING.md` 파일을 추가하지 않았다는 점을 눈여겨보자. `-a` 옵션을 사용하면 모든 파일이 자동으로 추가된다.

### 파일 삭제하기

Git에서 파일을 제거하려면 `git rm` 명령으로 Tracked 상태의 파일을 삭제한 후에(정확하게는 Staging Area에서 삭제하는 것) 커밋해야 한다. 이 명령은 워킹 디렉토리에 있는 파일도 삭제하기 때문에 실제로 파일도 지워진다.

Git 명령을 사용하지 않고 단순히 워킹 디렉터리에서 파일을 삭제하고 `git status` 명령으로 상태를 확인하면 Git은 현재 “Changes not staged for commit” (즉, *Unstaged* 상태)라고 표시해준다.

```

$ echo 'text' > PROJECTS.md
$ git add PROJECTS.md
$ git commit -m 'add PROJECTS.md'
$ rm PROJECTS.md
$ git status
On branch main
Your branch is ahead of 'origin/main' by 3 commits.
  (use "git push" to publish your local commits)

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

no changes added to commit (use "git add" and/or "git commit -a")

```

그리고 `git rm` 명령을 실행하면 삭제한 파일은 Staged 상태가 된다.



```
$ git rm PROJECTS.md
rm 'PROJECTS.md'
$ git status
On branch main
Your branch is ahead of 'origin/main' by 3 commits.
  (use "git push" to publish your local commits)

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

커밋하면 파일은 삭제되고 Git은 이 파일을 더는 추적하지 않는다. 이미 파일을 수정했거나 Staging Area에(역주 - Git Index라고도 부른다) 추가했다면 -f 옵션을 주어 강제로 삭제해야 한다. 이 점은 실수로 데이터를 삭제하지 못하도록 하는 안전장치다. 커밋 하지 않고 수정한 데이터는 Git으로 복구할 수 없기 때문이다.

또 Staging Area에서만 제거하고 워킹 디렉토리에 있는 파일은 지우지 않고 남겨둘 수 있다. 다시 말해서 하드디스크에 있는 파일은 그대로 두고 Git만 추적하지 않게 한다. 이것은 .gitignore 파일에 추가하는 것을 빼먹었거나 대용량 로그 파일이나 컴파일된 파일인 .a 파일 같은 것을 실수로 추가했을 때 쓴다. --cached 옵션을 사용하여 명령을 실행한다.

```
$ git rm --cached README
```

여러 개의 파일이나 디렉토리를 한꺼번에 삭제할 수도 있다. 아래와 같이 git rm 명령에 file-glob 패턴을 사용한다.

```
$ git rm log/\*.log
```

\* 앞에 \을 사용한 것을 기억하자. 파일명 확장 기능은 셸에만 있는 것이 아니라 Git 자체에도 있기 때문에 필요하다. 이 명령은 log/ 디렉토리에 있는 .log 파일을 모두 삭제한다. 아래의 예제처럼 할 수도 있다.

```
$ git rm \*~
```

이 명령은 ~로 끝나는 파일을 모두 삭제한다.

### 파일 이름 변경하기

아래와 같이 파일 이름을 변경할 수 있다.

```
$ git mv file_from file_to
```

잘 동작한다. 이 명령을 실행하고 Git의 상태를 확인해보면 Git은 이름이 바뀐 사실을 알고 있다.

```
$ git mv README.md README
$ git status
On branch main
Your branch is ahead of 'origin/main' by 4 commits.
  (use "git push" to publish your local commits)

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

사실 `git mv` 명령은 아래 명령어를 수행한 것과 완전 똑같다.

```
$ mv README.md README
$ git rm README.md
$ git add README
```

`git mv` 명령은 일종의 단축 명령어이다. 이 명령으로 파일 이름을 바뀌도 되고 `mv` 명령으로 파일 이름을 직접 바뀌도 된다. 단지 `git mv` 명령은 편리하게 명령을 세 번 실행해주는 것 뿐이다. 어떤 도구로 이름을 바뀌도 상관없다. 중요한 것은 이름을 변경하고 나서 꼭 `rm/add` 명령을 실행해야 한다는 것 뿐이다.

## 2.3 Git의 기초 - 커밋 히스토리 조회하기

### 커밋 히스토리 조회하기

새로 저장소를 만들어서 몇 번 커밋을 했을 수도 있고, 커밋 히스토리가 있는 저장소를 Clone 했을 수도 있다. 어쨌든 가끔 저장소의 히스토리를 보고 싶을 때가 있다. Git에는 히스토리를 조회하는 명령어인 `git log` 가 있다.

이 예제에서는 “simplegit”이라는 매우 단순한 프로젝트를 사용한다. 아래와 같이 이 프로젝트를 Clone 한다.

```
$ cd /workspace
$ git clone https://github.com/schacon/simplegit-progit
$ cd simplegit-progit/
```

이 프로젝트 디렉토리에서 `git log` 명령을 실행하면 아래와 같이 출력된다.

```
$ git log
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    changed the version number

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 16:40:33 2008 -0700

    removed unnecessary test

commit allbef06a3f659402fe7563abf99ad00de2209e6
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 10:31:28 2008 -0700

    first commit
```

특별한 아규먼트 없이 `git log` 명령을 실행하면 저장소의 커밋 히스토리를 시간순으로 보여준다. 즉, 가장 최근의 커밋이 가장 먼저 나온다. 그리고 이어서 각 커밋의 SHA-1 체크섬, 저자 이름, 저자 이메일, 커밋한 날짜, 커밋 메시지를 보여준다.

원하는 히스토리를 검색할 수 있도록 `git log` 명령은 매우 다양한 옵션을 지원한다. 여기에서는 자주 사용하는 옵션을 설명한다.

여러 옵션 중 `-p`, `--patch` 는 굉장히 유용한 옵션이다. `-p` 는 각 커밋의 diff 결과를 보여준다. 다른 유용한 옵션으로 `-2` 가 있는데 최근 두 개의 결과만 보여주는 옵션이다:

```

$ git log -p -2
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    changed the version number

diff --git a/Rakefile b/Rakefile
index a874b73..8f94139 100644
--- a/Rakefile
+++ b/Rakefile
@@ -5,7 +5,7 @@ require 'rake/gempackagetask'
spec = Gem::Specification.new do |s|
  s.platform = Gem::Platform::RUBY
  s.name = "simplegit"
-  s.version = "0.1.0"
+  s.version = "0.1.1"
  s.author = "Scott Chacon"
  s.email = "schacon@gee-mail.com"
  s.summary = "A simple gem for using Git in Ruby code."

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 16:40:33 2008 -0700

    removed unnecessary test

diff --git a/lib/simplegit.rb b/lib/simplegit.rb
index a0a60ae..47c6340 100644
--- a/lib/simplegit.rb
+++ b/lib/simplegit.rb
@@ -18,8 +18,3 @@ class SimpleGit
  end

end

-
-if $0 == __FILE__
-  git = SimpleGit.new
-  puts git.show
-end

```

이 옵션은 직접 diff를 실행한 것과 같은 결과를 출력하기 때문에 동료가 무엇을 커밋했는지 리뷰하고 빨리 조회하는데 유용하다. 또 git log 명령에는 히스토리의 통계를 보여주는 옵션도 있다. --stat 옵션으로 각 커밋의 통계 정보를 조회할 수 있다.

```
$ git log --stat
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700
```

```
    changed the version number
```

```

Rakefile | 2 +-
1 file changed, 1 insertion(+), 1 deletion(-)
```

```
commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 16:40:33 2008 -0700
```

```
    removed unnecessary test
```

```

lib/simplegit.rb | 5 -----
1 file changed, 5 deletions(-)
```

```
commit a11bef06a3f659402fe7563abf99ad00de2209e6
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 10:31:28 2008 -0700
```

```
    first commit
```

```

README          | 6 ++++++
Rakefile        | 23 +++++++++++++++++++++
lib/simplegit.rb | 25 +++++++++++++++++++++
3 files changed, 54 insertions(+)
```

이 결과에서 --stat 옵션은 어떤 파일이 수정됐는지, 얼마나 많은 파일이 변경됐는지, 또 얼마나 많은 라인을 추가하거나 삭제했는지 보여준다. 요약정보는 가장 뒤쪽에 보여준다.

다른 또 유용한 옵션은 --pretty 옵션이다. 이 옵션을 통해 히스토리 내용을 보여줄 때 기본 형식 이외에 여러 가지 중에 하나를 선택할 수 있다. 몇개 선택할 수 있는 옵션의 값이 있다. oneline 옵션은 각 커밋을 한 라인으로 보여준다. 이 옵션은 많은 커밋을 한 번에 조회할 때 유용하다. 추가로 short, full, fuller 옵션도 있는데 이것은 정보를 조금씩 가감해서 보여준다.

```
$ git log --pretty=oneline
ca82a6dff817ec66f44342007202690a93763949 changed the version number
085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7 removed unnecessary test
a11bef06a3f659402fe7563abf99ad00de2209e6 first commit
```

format 옵션은 나만의 포맷으로 결과를 출력하고 싶을 때 사용한다. 특히 결과를 다른 프로그램으로 파싱하고자 할 때 유용하다. 이 옵션을 사용하면 포맷을 정확하게 일치시킬 수 있기 때문에 Git을 새 버전으로 바뀌어도 결과 포맷이 바뀌지 않는다.

```
$ git log --pretty=format:"%h - %an, %ar : %s"
ca82a6d - Scott Chacon, 6 years ago : changed the version number
085bb3b - Scott Chacon, 6 years ago : removed unnecessary test
allbef0 - Scott Chacon, 6 years ago : first commit
```

git log --pretty=format 에 쓸 몇가지 유용한 옵션` 포맷에서 사용하는 유용한 옵션.

옵션	설명
%H	커밋 해시
%h	짧은 길이 커밋 해시
%T	트리 해시
%t	짧은 길이 트리 해시
%P	부모 해시
%p	짧은 길이 부모 해시
%an	저자 이름
%ae	저자 메일
%ad	저자 시각 (형식은 --date=옵션 참고)
%ar	저자 상대적 시각
%cn	커미터 이름
%ce	커미터 메일
%cd	커미터 시각
%cr	커미터 상대적 시각
%s	요약

oneline 옵션과 format 옵션은 --graph 옵션과 함께 사용할 때 더 효과적이다. 이 명령은 브랜치와 머지 히스토리를 보여주는 아스키 그래프를 출력한다. SpringBoot 오픈소스 프로젝트를 clone 받아 살펴보자.

```
$ git clone https://github.com/spring-projects/spring-boot
$ cd spring-boot
```

```
$ git log --pretty=format:"%h %s" --graph
* b91f814e42 Fix incomplete assertions
* 7828fbfdab Merge branch '3.0.x'
|\
| * d4ef5dc151 Merge branch '2.7.x' into 3.0.x
| |\
| | * 17a4d303f8 Switch Java 20 CI to Bellsoft Liberica
* | | e8e29ecfb8 Merge branch '3.0.x'
|\| |
| * | 1933e9a28e Merge branch '2.7.x' into 3.0.x
| |\|
| | * 04e7e70ebd Remove JDK 19 CI
* | | 718e2ff276 Merge branch '3.0.x'
|\| |
| * | 1da4271f81 Merge branch '2.7.x' into 3.0.x
| |\|
| | * e79ac4b24e Upgrade CI images to ubuntu:jammy-20230308
* | | 54a623f4c9 Merge branch '3.0.x'
|\| |
| * | 042c8e94a9 Merge branch '2.7.x' into 3.0.x
| |\|
| | * 0d82729a29 Upgrade to Gradle Enterprise Gradle plugin 3.12.5
* | | fd9b8fe020 Merge branch 'gh-34658'
|\ \ \
| * | | 95f45eab1f Create service connections from Testcontainers-
managed containers
| * | | 8ec266bea4 Add infrastructure for pluggable connection details
factories
| / / /
* | | 8e4b8a869e Merge branch 'gh-34657'
|\ \ \
| * | | 8721c0e64f Add ConnectionDetail support to Zipkin auto-
configuration
```

다음 장에서 살퍼볼 브랜치나 Merge 결과의 히스토리를 이런 식으로 살퍼보면 훨씬 흥미롭다.

git log 명령의 기본적인 옵션과 출력물의 형식에 관련된 옵션을 살퍼보았다. git log 명령은 앞서 살퍼본 것보다 더 많은 옵션을 지원한다. git log 주요 옵션 는 지금 설명한 것과 함께 유용하게 사용할 수 있는 옵션이다. 각 옵션으로 어떻게 log 명령을 제어할 수 있는지 보여준다.

옵션	설명
-p	각 커밋에 적용된 패치를 보여준다.
--stat	각 커밋에서 수정된 파일의 통계정보를 보여준다.
--shortstat	--stat 명령의 결과 중에서 수정한 파일, 추가된 라인, 삭제된 라인만 보여준다.
--name-only	커밋 정보중에서 수정된 파일의 목록만 보여준다.
--name-status	수정된 파일의 목록을 보여줄 뿐만 아니라 파일을 추가한 것인지, 수정한 것인지, 삭제한 것인지도 보여준다.

--abbrev-commit	40자 짜리 SHA-1 체크섬을 전부 보여주는 것이 아니라 처음 몇 자만 보여준다.
--relative-date	정확한 시간을 보여주는 것이 아니라 "2 weeks ago" 처럼 상대적인 형식으로 보여준다.
--graph	브랜치와 머지 히스토리 정보까지 아스키 그래프로 보여준다.
--pretty	지정한 형식으로 보여준다. 이 옵션에는 oneline, short, full, fuller, format이 있다. format은 원하는 형식으로 출력하고자 할 때 사용한다.
--oneline	--pretty=oneline --abbrev-commit 두 옵션을 함께 사용한 것과 같다.

## 조회 제한조건

출력 형식과 관련된 옵션을 살펴봤지만 git log 명령은 조회 범위를 제한하는 옵션들도 있다. 히스토리 전부가 아니라 부분만 조회한다. 이미 최근 두 개만 조회하는 -2 옵션은 살펴봤다. 실제 사용법은 -<n> 이고 n은 최근 n개의 커밋을 의미한다. 사실 이 옵션을 자주 쓰진 않는다. Git은 기본적으로 출력을 pager류의 프로그램을 거쳐서 내보내므로 한 번에 한 페이지씩 보여준다.

반면 --since 나 --until 같은 시간을 기준으로 조회하는 옵션은 매우 유용하다. 지난 2주 동안 만들어진 커밋들만 조회하는 명령은 아래와 같다.

```
$ git log --since=2.weeks
```

이 옵션은 다양한 형식을 지원한다. "2008-01-15" 같이 정확한 날짜도 사용할 수 있고 "2 years 1 day 3 minutes ago" 같이 상대적인 기간을 사용할 수도 있다.

또 다른 기준도 있다. --author 옵션으로 저자를 지정하여 검색할 수도 있고 --grep 옵션으로 커밋 메시지에서 키워드를 검색할 수도 있다

노트	--author 와 --grep 옵션을 함께 사용하여 모두 만족하는 커밋을 찾으려면 --all-match 옵션도 반드시 함께 사용해야 한다.
----	--

이제 살펴볼 예제는 Merge 커밋을 제외한 순수한 커밋을 확인해보는 명령이다. Andy Wilkinson 이 2023-03-23 ~ 24 에 커밋한 이력들이다.



```
$ git log --pretty="%h - %s" --author="Andy Wilkinson" --since="2023-03-23" \
--before="2023-03-24" --no-merges
b91f814e42 - Fix incomplete assertions
17a4d303f8 - Switch Java 20 CI to Bellsoft Liberica
04e7e70ebd - Remove JDK 19 CI
e79ac4b24e - Upgrade CI images to ubuntu:jammy-20230308
0d82729a29 - Upgrade to Gradle Enterprise Gradle plugin 3.12.5
95f45eab1f - Create service connections from Testcontainers-managed
containers
8ec266bea4 - Add infrastructure for pluggable connection details
factories
8721c0e64f - Add ConnectionDetail support to Zipkin auto-configuration
ac55caa463 - Add ConnectionDetail support to Redis auto-configuration
69f31cb6c0 - Add ConnectionDetail support to Rabbit auto-configuration
de8fb04814 - Add ConnectionDetail support to Neo4J auto-configuration
2ef33dc81f - Add ConnectionDetail support to Mongo auto-configuration
042f0c8520 - Add ConnectionDetail support to Kafka auto-configuration
d860d875b9 - Add ConnectionDetail support to Influx auto-configuration
4cc7958c0b - Add ConnectionDetail support to Elasticsearch auto-
configuration
9f187bb13a - Add ConnectionDetail support to Couchbase auto-
configuration
4307fdc0a0 - Add ConnectionDetail support to Cassandra auto-
configuration
61e9fe8cd4 - Add ConnectionDetail support to R2DBC auto-configuration
d09ac00824 - Add ConnectionDetail support to JDBC auto-configuration
aa91f2b8b6 - Introduce ConnectionDetails interface
```

#### 힌트

머지 커밋 표시하지 않기

저장소를 사용하는 워크플로우에 따라 머지 커밋이 차지하는 비중이 클 수도 있다. --no-merges 옵션을 사용하면 검색 결과에서 머지 커밋을 표시하지 않도록 할 수 있다.