

4장 커밋

- 4.1 코드의 변화
- 4.2 새 파일 생성 및 감지
- 4.3 깃에 새 파일 등록
- 4.4 첫 번째 커밋
- 4.5 커밋 확인
- 4.6 두 번째 커밋
- 4.7 메시지가 없는 빈 커밋

1

4장 커밋

- 4.8 커밋 아이디
- 4.9 커밋 로그
- 4.10 diff 명령어
- 4.11 정리

2

4.1 코드의 변화

3

1. 코드의 변화

➤ 코드의 변화

- 깃은 개발 중인 코드의 이력을 만들 수 있음
- 커밋(commit):
 - 깃이 코드 변화를 기록하는 것
- 영어로 commit은 여러 의미가 있음
- 그중 깃의 동작과 가장 유사한 의미는 '~를 적어 두다'임
- 커밋은 의미 있는 변경 작업들을 저장소에 기록하는 동작임

1. 코드의 변화

➤ 코드의 변화

- 개발 과정에서 소스 코드는 수없이 수정됨
- 일반적으로는 새로운 기능을 추가하는 코드를 삽입
- 버그를 수정하려고 많은 코드를 이동하거나 대체함
- 이러한 코드 수정은 개발 목적을 달성하는 작업들임

변경 전

안녕하세요.
반갑습니다.

변경 후



안녕하세요. 지니입니다.
이렇게 만나서 반갑습니다.

1. 코드의 변화

➤ 코드의 변화

- 개발자는 만일의 경우에 대비하여 중간에 코드 변경 과정을 기록하길 원함
- 변경 시점을 저장해 두면 잘못된 동작을 발견했을 때 특정 시점으로 되돌아갈 수
이요

최종본

안녕하세요. 지니입니다.
이렇게 만나서 반갑습니다.

이전으로 복귀



안녕하세요.
반갑습니다.

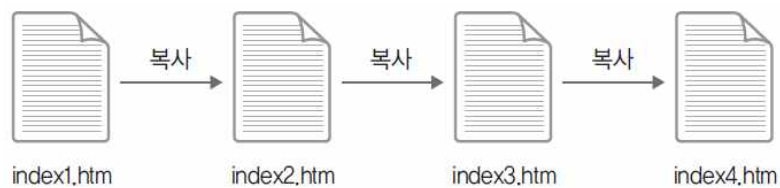
- 이때 필요한 것이 깃의 버전 관리임
- 깃은 코드의 변경 이력과 시점을 커밋으로 기록함
- 사용자가 일일이 기억하지 않아도 됨
- 이전 시점으로 쉽게 되돌아갈 수 있으며, 실수도 없음

1. 코드의 변화

➤ 파일 관리 방법

- 보통 우리는 의미 있는 변경을 할 때 파일을 복사함
- 복사한 새 파일에는 추가하거나 변경하고 싶은 내용을 적용함
- 파일을 복사하는 형태는 파일의 변경 내역을 기록하는 것보다 더 많은 파일을 생성하고 관리해야 하는 부작용이 있음
- 모든 내용이 중복되기 때문에 용량도 많이 차지함

▼ 그림 4-1 파일 복사로 파일 관리

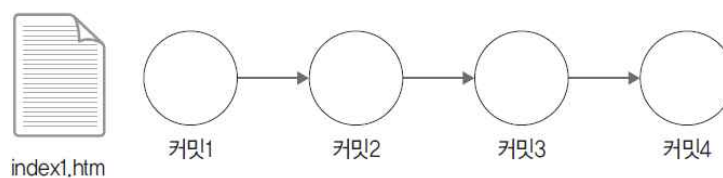


1. 코드의 변화

➤ 파일 관리 방법

- 깃의 커밋은 새로 변경된 부분만 추출하여 저장함
- 파일 이름을 변경하지 않고도 동일한 파일 이름으로 하나로 관리가 가능함
- 커밋:
시간에 따라 변화되는 내용만 관리하고, 코드가 변화된 시간 순서에 따라서 영구적으로 저장함

▼ 그림 4-2 깃으로 파일 관리



1. 코드의 변화

➤ 파일 관리 방법

- 개발자 입장에서는 복잡한 구조의 파일을 관리하지 않아도 되고, 여러 개의 파일보다는 파일 하나로 모든 이력을 처리하기 때문에 유용함
- 커밋은 부모 커밋(parent commit)을 기반으로 변화된 부분만 새로운 커밋으로 생성함
- 커밋은 파일의 시간적 변화도 함께 저장함

4.2 새 파일 생성 및 감지

2. 새 파일 생성 및 감지

➤ 새 파일 생성

- 실습을 위해 간단한 HTML 파일을 하나 작성함
- 에디터를 이용하여 코드를 작성하면 됨
- 필자는 VS Code를 이용함

```
$ mkdir gitstudy04 ----- 새 폴더 만들기
$ cd gitstudy04 ----- 만든 폴더로 이동
$ git init ----- 저장소를 것으로 초기화
Initialized empty Git repository in E:/gitstudy04/.git/

infoh@hojin MINGW64 /e/gitstudy04 (master)
$ code index.htm ----- VS Code를 사용하여 파일 작성
```

2. 새 파일 생성 및 감지

➤ 새 파일 생성

index.htm

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <title>Page Title</title>
</head>
<body>

</body>
</html>
```

2. 새 파일 생성 및 감지

➤ 새 파일 생성

- 작성한 예제 파일은 기본이 되는 HTML의 뼈대 페이지임

▼ 그림 4-3 파일 생성 과정



2. 새 파일 생성 및 감지

➤ 깃에서 새 파일 생성 확인

- 워킹 디렉터리에 새 파일이 생성됨
- 워킹 디렉터리에 새 파일이 추가되면 깃은 변화된 상태를 자동으로 감지함
- 이때 깃 상태를 확인할 수 있는 명령어가 status임
- status 명령어를 입력하면 메시지가 출력되는 것을 볼 수 있음

```
infoh@hojin MINGW64 /e/gitstudy04 (master)
$ git status ----- 상태 확인
On branch master
No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        index.htm ----- 새로운 파일이 등록된 것을 확인
nothing added to commit but untracked files present (use "git add" to track)
```

2. 새 파일 생성 및 감지

➤ 깃에서 새 파일 생성 확인

- 깃의 상태 메시지에서 Untracked files 표시 부분을 확인함
- 깃 배시 터미널로 실행하면 추적되지 않은 파일은 빨간색으로 표시함
- “Untracked files” 메시지는 워킹 디렉터리에 새로운 파일이 등록되었다고 알려 주는 것임
- 깃은 워킹 디렉터리에 새 파일이 추가되면 상태를 감지하고 향후 이력을 추적할지 여부를 결정함

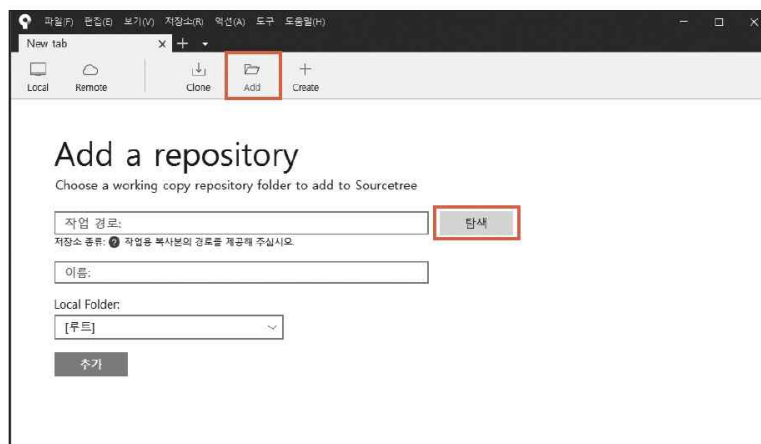
2. 새 파일 생성 및 감지

➤ 소스트리에서 새 파일 감지

- 소스트리를 사용하여 깃의 status 명령어와 동일한 상태를 확인할 수 있음
- 아직 gitstudy04 폴더와 소스트리를 연동하지 않음
- 새 탭에서 Add 버튼을 클릭함
- 탐색을 눌러 깃 배시에서 만든 gitstudy04 폴더를 찾아 선택한 후 추가를 누름

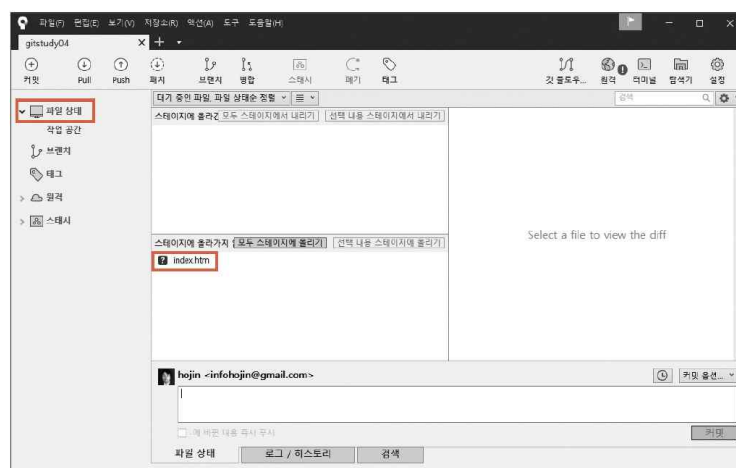
2. 새 파일 생성 및 감지

▼ 그림 4-4 소스트리에 저장소 추가



2. 새 파일 생성 및 감지

▼ 그림 4-5 소스트리에서 새 파일 감지



2. 새 파일 생성 및 감지

➤ 소스트리에서 새 파일 감지

- 소스트리는 GUI로 상태를 직관적으로 표시해 줌
- 사실 내부적으로는 소스트리가 status 명령어를 대신 깃에 입력하는 것임
- 직접 status 명령어를 실행하지 않아도 쉽게 확인할 수 있음

4.3 깃에 새 파일 등록

3. 깃에 새 파일 등록

➤ 깃에 새 파일 등록

- 워킹 디렉터리에 있는 파일은 깃이 자동으로 추적 관리하지 않음
- 커밋을 하려면 **파일의 상태가 추적 가능해야 함**
- **등록:**
워킹 디렉터리에 새로 추가된 untracked 상태의 파일을 추적 가능 상태로 변경하는 것
- 파일을 등록하면 워킹 디렉터리의 파일이 스테이지 영역에 추가됨
- 스테이지 영역의 관리 목록에 추가된 파일만 깃에서 이력을 추적할 수 있음

3. 깃에 새 파일 등록

▼ 그림 4-6 새 파일 등록



3. 깃에 새 파일 등록

➤ 스테이지에 등록

- 등록:
워킹 디렉터리에 있는 파일을 스테이지(stage) 영역으로 복사하는 것을 의미
- '복사'는 실제 파일을 복사하는 것을 의미하지는 않음
- 깃 내부에서 논리적인 기록을 변경하는 과정일 뿐임
- 복사라고 표현한 것은 이해하기 쉽게 풀어 쓴 것임
- 워킹 디렉터리에 추가된 모든 파일을 커밋할 때는 반드시 이 과정을 거쳐야 함
- 깃에서 버전 이력을 관리할 수 있음
- 스테이지에 등록되지 않은 unstage 상태의 파일들은 커밋할 수 없음
- 깃은 커밋하기 전에 파일들이 stage 상태인지 unstage 상태인지를 판단함
- 스테이지 영역으로 등록된 파일들은 tracked 상태로 자동 변경됨

3. 깃에 새 파일 등록

➤ 스테이지에 등록

명령어로 등록: add 명령어

- 현재는 커밋 명령어를 실행하기 이전의 중간 단계임
- 깃의 add 명령어는 워킹 디렉터리의 파일을 스테이지 영역으로 등록
- 깃은 안정적인 커밋을 할 수 있도록 add 명령어를 기준으로 이전과 이후 단계를 구별함
- 터미널에서는 다음 형태의 명령어를 입력

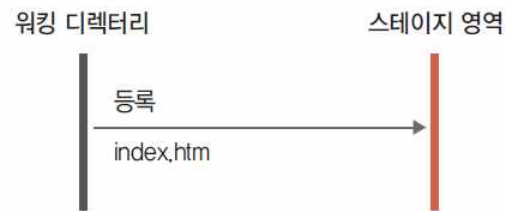
```
$ git add 파일이름
```

```
infoh@hojin MINGW64 /e/gitstudy04 (master)
```

```
$ git add index.htm ----- 스테이지에 등록
```

3. 깃에 새 파일 등록

▼ 그림 4-7 스테이지 영역에 등록



3. 깃에 새 파일 등록

➤ 스테이지에 등록

- add 명령어를 실행하면 지정한 파일은 스테이지 영역으로 등록됨
- 스테이지 영역에 파일이 등록되면 파일은 **tracked** 상태로 변경됨
- 파일 이름 대신 점(.)을 이용하면 전체 파일과 폴더를 모두 등록할 수 있음
- 점(.)은 리눅스와 같은 운영 체제에서 현재 디렉터리를 의미하는 기호임

예

```
$ git add .
```

3. 깃에 새 파일 등록


➤ 스테이지에 등록

- 워킹 디렉터리에 생성된 모든 파일을 스테이지 영역에 추가할 필요는 없음
- 필요한 파일만 스테이지 영역에 등록하여 이력을 추적하면 됨
- **스테이지 영역에 등록하지 않은 파일은 커밋 작업에 포함되지 않음**
- 등록 명령으로 파일들의 이력을 커밋 기록에 포함할지 여부를 결정할 수 있음
- 정보 이력을 추적하고 싶은 파일만 스테이지 영역에 추가함
- 단 빈 폴더는 스테이지 영역에 등록할 수 없음
- 폴더 안에 파일이 하나 이상 있어야 등록이 가능함

3. 깃에 새 파일 등록

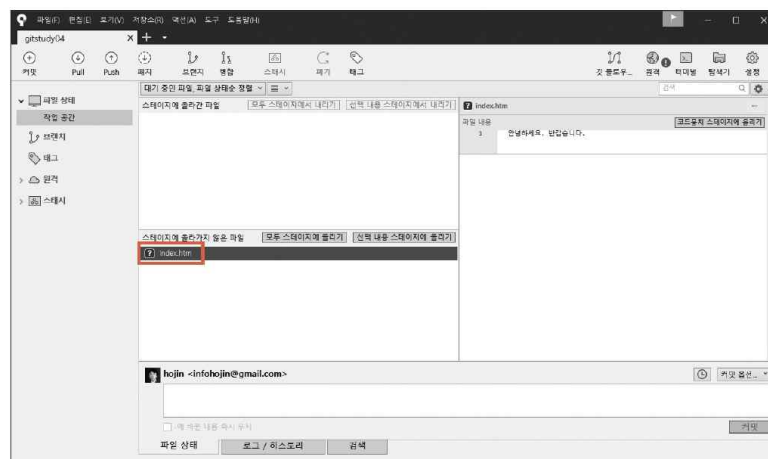
➤ 스테이지에 등록

소스트리에서 등록

- 소스트리는 스테이지 영역에 등록되는 파일을 직관적으로 확인할 수 있음
- **untracked** 상태인 파일은 소스트리의 **스테이지에 올라가지 않은 파일** 영역에서 확인할 수 있음
- 파일을 선택하여 상위 영역의 **스테이지에 올라간 파일** 부분으로 옮김
- 소스트리는 추적 상태와 추적하지 않음을 쉽게 구별할 수 있도록 파일 이름 앞에 아이콘을 함께 표시함
- 보라색 아이콘  index.htm 은 untracked 상태의 파일임
- 새로 생성된 파일을 의미하기도 함

3. 깃에 새 파일 등록

▼ 그림 4-8 스테이지에 올라가지 않은 파일



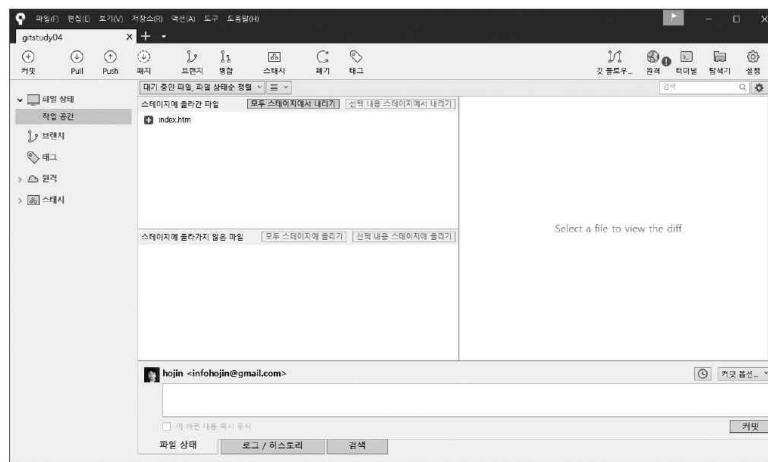
3. 깃에 새 파일 등록

➤ 스테이지에 등록

- untracked 상태의 파일 여러 개를 **모두 스테이지에 올리기** 로 한 번에 등록할 수 있음
- 전체 파일을 스테이지 영역에 등록할 때는 **모두 스테이지에 올리기를** 누름
- 스테이지 영역에 등록된 파일은 스테이지에 올라간 파일 목록에서 확인할 수 있음
- 스테이지에 등록되면 파일 이름 앞에 녹색 아이콘 **+ index.htm** 을 표시함

3. 깃에 새 파일 등록

▼ 그림 4-9 스테이지에 파일이 등록됨



3. 깃에 새 파일 등록

➤ 스테이지에 등록

- **선택 내용 스테이지에 올리기**를 사용하면 한 번에 하나씩 파일을 등록할 수 있음
- 등록할 파일 이름을 목록에서 선택한 후 **선택 내용 스테이지에 올리기**를 누름

3. 깃에 새 파일 등록

➤ 파일 등록 취소

- 워킹 디렉터리에 있는 새로운 파일이 스테이지 영역에 등록됨
- 콘솔창에서 status 명령어를 사용하여 등록 상태를 다시 한 번 확인해보자

```

infoh@hohjin MINGW64 /e/gitstudy04 (master)
$ git status ----- 상태 확인
On branch master
No commits yet
Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   index.htm ----- 스테이지에 등록, 새 파일 상태

```

3. 깃에 새 파일 등록

➤ 파일 등록 취소

- 이번에는 tracked 상태의 파일을 **untracked 상태로 변경**해보자
- 스테이지에 등록하는 것과 반대 과정임
- 등록 취소는 워킹 디렉터리와 스테이지 영역을 서로 왔다 갔다 할 수 있는 방법
- unstage 상태로 변경하려면 삭제(rm)나 리셋(reset) 명령어를 사용함

▼ 그림 4-10 스테이지 영역의 파일 등록 취소



3. 깃에 새 파일 등록

➤ 파일 등록 취소

- rm 명령어로 삭제해보자
- 스테이지 영역에서만 등록된 파일을 삭제하려고 --cached 옵션을 함께 사용함

```
infoh@hojin MINGW64 /e/gitstudy04 (master)
$ git rm --cached index.htm ----- 스테이지 삭제
rm 'index.htm'
```

3. 깃에 새 파일 등록

➤ 파일 등록 취소

- 스테이지의 캐시 목록에서 파일이 삭제됨
- 다시 status 명령어를 실행하여 확인함

```
infoh@hojin MINGW64 /e/gitstudy04 (master)
$ git status ----- 상태 확인
On branch master
No commits yet
Untracked files: ----- 추적하지 않음
  (use "git add <file>..." to include in what will be committed)
    index.htm ----- 스테이지 삭제
nothing added to commit but untracked files present (use "git add" to track)
```

3. 깃에 새 파일 등록

➤ 파일 등록 취소

- 등록하기 이전의 untracked 상태로 변경됨
- 다음 실습에 대비하여 다시 tracked 상태로 변경해 놓음

```
infoh@hojin MINGW64 /e/gitstudy04 (master)
$ git add index.htm ----- 스테이지에 다시 등록
```

3. 깃에 새 파일 등록

➤ 파일 등록 취소

- 파일을 등록한 후 커밋하지 않고 바로 삭제하려면 **rm --cached** 명령어를 사용함
- 한번이라도 커밋을 했다면 **reset** 명령어를 사용해야 함

예

```
infoh@hojin MINGW64 /e/gitstudy04 (master)
$ git rm --cached index.htm
rm 'index.htm'
```

3. 깃에 새 파일 등록

> 파일 등록 취소

- 삭제한 후 status 명령어를 실행하면 다음과 같이 이전과 다른 결과가 나옴

예

```
infoh@hojin MINGW64 /e/gitstudy04 (master)
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)
        deleted:   index.htm
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        index.htm  ----- 스테이지 삭제
```

3. 깃에 새 파일 등록

> 파일 등록 취소

- 파일이 untracked 상태가 되고, 스테이지 영역에서 파일이 삭제 처리됨
- 커밋 후 삭제는 파일이 **삭제 또는 변화된 것으로 간주함**
- 커밋된 파일은 리셋으로 삭제한 후 정리해 주어야 함

예

```
infoh@hojin MINGW64 /e/gitstudy04 (master)
$ git reset HEAD index.htm ----- 리셋 시도
```

3. 깃에 새 파일 등록

➤ 파일 등록 취소

- 다시 status 명령어로 확인하면 정상적으로 커밋이 정리됨

예

```
infoh@hojin MINGW64 /e/gitstudy04 (master)
```

```
$ git status ----- 상태 확인
```

```
On branch master
```

```
nothing to commit, working tree clean
```

- 터미널에서 unstage 상태 및 untracked 상태로 변경하는 것은 복잡함
- 소스트리를 이용하면 스테이지 영역에 등록된 파일을 좀 더 쉽게 등록 취소할 수 있음
- 모두 스테이지에서 내리기와 선택 내용 스테이지에서 내리기를 사용하면 untracked 상태로 쉽게 변경할 수 있음

3. 깃에 새 파일 등록

➤ 등록된 파일 이름이 변경되었을 때

- 작업 도중 파일 이름도 변경할 수 있음
- 파일 이름을 변경했다고 별도로 깃에 통보할 필요는 없음
- 깃은 똑똑해서 변경된 파일 이름을 자동으로 알고 있음
- 리눅스나 macOS에서는 mv 명령어로 파일 이름을 변경할 수 있음
- 깃에서도 파일 이름을 변경할 때 mv 명령어를 사용함

```
$ git mv 파일이름 새파일이름
```

3. 깃에 새 파일 등록

➤ 등록된 파일 이름이 변경되었을 때

- 다음과 같이 index.htm 파일의 이름을 변경하고 상태를 확인해 보면 깃에서 변경된 파일을 계속 추적하는 것을 알 수 있음

```
infoh@hojin MINGW64 /e/gitstudy04 (master)
$ git mv index.htm home.htm ----- 파일 이름 변경
$ git status
On branch master
No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
    new file:   home.htm ----- 변경된 파일 이름
```

3. 깃에 새 파일 등록

➤ 등록된 파일 이름이 변경되었을 때

- 굳이 git mv 명령어를 사용하지 않고, 운영 체제의 mv 명령어를 사용해도 됨
- 깃의 git mv 명령어를 여러 단계의 명령으로 풀면 다음과 같음

예

```
infoh@hojin MINGW64 /e/gitstudy04 (master)
$ mv index.htm home.htm
$ git rm index.htm
$ git add home.htm
```

3. 깃에 새 파일 등록

➤ 등록된 파일 이름이 변경되었을 때

- 이름을 변경한다는 의미는 기존 파일을 삭제하고 새 파일을 다시 스테이지 영역에 등록하는 과정과 유사함
- 풀어 쓴 명령에서 이름을 변경한 후에는 rm과 add 명령어를 실행해야 한다는 사실을 잊지 말자
- 다음 실습을 위해 이전 이름으로 되돌려 놓음

```
infoh@hojin MINGW64 /e/gitstudy04 (master)  
$ git mv home.htm index.htm
```

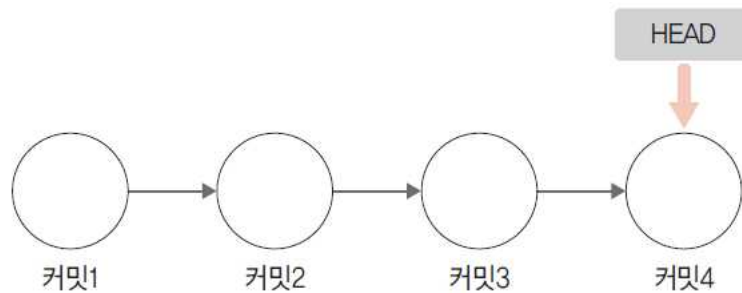
4.4 첫 번째 커밋

4. 첫 번째 커밋

➤ HEAD

- 깃에는 HEAD라는 포인터 개념이 있음
- HEAD는 커밋을 가리키는 묵시적 참조 포인터임

▼ 그림 4-11 HEAD



4. 첫 번째 커밋

➤ HEAD

- HEAD는 최종적인 커밋 작업의 위치를 가리킴
- 앞에서 새로운 커밋은 이전 부모 커밋을 기반으로 새로운 커밋을 만든다고 했음
- HEAD는 바로 부모 커밋을 가리킴
- 단 깃을 설치하고 처음 커밋할 때는 HEAD의 포인터가 없음
- 최소한 한 번 이상 커밋을 해야만 HEAD가 존재함
- HEAD는 커밋될 때마다 한 단계씩 이동함
- 마지막 커밋 위치를 가리킴
- HEAD는 커밋이 변화한 최종 시점을 의미함

4. 첫 번째 커밋

➤ 스냅샷

- 커밋은 파일 변화를 깃 저장소에 영구적으로 기록함
- 커밋은 이전에 파일을 복사하여 관리하던 방식과는 큰 차이가 있음
- 깃이 다른 버전 관리 도구와 다른 점은 스냅샷(snapshot) 방식을 이용한다는 것임
- 파일을 복사하는 방식으로 수정분을 관리하면 같은 내용을 반복해서 저장하기에 많은 용량을 차지함
- 수정된 부분들을 일일이 찾아야 하기 때문에 검색할 때도 매우 불편함

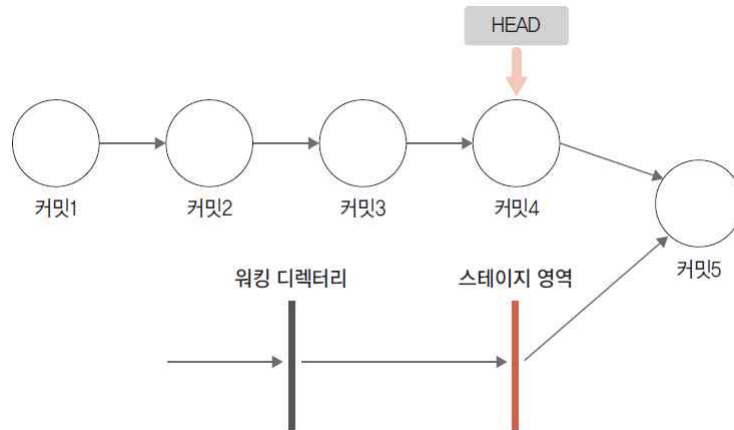
4. 첫 번째 커밋

➤ 스냅샷

- 깃은 이러한 시스템적인 단점을 해결하려고 변경된 파일 전체를 저장하지 않고, 파일에서 변경된 부분을 찾아 수정된 내용만 저장함
- 스냅샷 방식:
마치 변화된 부분만 찾아 사진을 찍는 것과 같음

4. 첫 번째 커밋

▼ 그림 4-12 파일에서 변경된 부분을 찾아 사진을 찍듯이 저장



4. 첫 번째 커밋

➤ 스냅샷

- 깃의 스냅샷은 HEAD가 가리키는 커밋을 기반으로 사진을 찍음
- 스테이지 영역과 비교하여 새로운 커밋으로 기록함
- 깃은 스냅샷 방식을 이용하여 빠르게 버전의 차이점을 처리하고, 용량을 적게 사용함

4. 첫 번째 커밋

➤ 파일 상태와 커밋

- 커밋은 변화된 내용을 영구적으로 깃 저장소에 기록함
- 새롭게 생성된 파일을 커밋하려면 반드시 **tracked** 상태로 변경해 주어야 함
- tracked 상태로 파일이 변경됨과 동시에 스테이지 영역에 등록함
- tracked 상태인 파일을 수정하면 다시 modified 상태로 변경됨
- modified는 untracked 상태임
- untracked 상태의 파일은 반드시 등록 명령으로 다시 스테이지 상태로 재등록해야 함
- 재등록하면 다시 tracked 상태로 변경됨

4. 첫 번째 커밋

➤ 파일 상태와 커밋

- 커밋하기 전에는 status 명령어로 항상 상태를 확인하는 습관이 필요함
- 워킹 디렉터리가 깨끗하게 정리되어 있지 않으면 커밋 명령어를 수행할 수 없음
- 커밋을 하려면 스테이지 영역에 새로운 변경 내용이 있어야 함
- 수정된 내용이 스테이지 영역으로 등록되지 않으면 커밋을 할 수 없음
- 커밋은 수정된 내용을 한 번만 등록함
- 스테이지 영역의 파일이 변경되지 않았다면 커밋을 두 번 실행할 수 없음
- 깃은 스테이지 영역의 변경된 내용을 기준으로 스냅샷을 만들어 커밋하기 때문임

4. 첫 번째 커밋

➤ 파일 상태와 커밋

명령어로 커밋: **commit** 명령어

- 수정된 파일 이력을 커밋하려면 **commit** 명령어를 사용함

```
$ git commit
```

4. 첫 번째 커밋

➤ 파일 상태와 커밋

- **commit** 명령어는 독립적으로 사용할 수 있음
- 옵션을 추가하여 여러 동작을 같이 수행할 수도 있음
- 커밋 옵션은 **-help** 명령어를 입력하면 볼 수 있음

예

```
infoh@hojin MINGW64 /e/gitstudy04 (master)
```

```
$ git commit -help
```

```
usage: git commit [<options>] [--] <paths-spec>...
```

-q, --quiet	suppress summary after successful commit
-v, --verbose	show diff in commit message template

Commit message options

4. 첫 번째 커밋

➤ 파일 상태와 커밋

```
-F, --file <file>      read message from file
--author <author>      override author for commit
--date <date>          override date for commit
-m, --message <message>
                        commit message
-c, --reedit-message <commit>
                        reuse and edit message from specified commit
```

이하 생략

4. 첫 번째 커밋

➤ 파일 상태와 커밋

- 깃의 커밋은 HEAD와 스테이지 영역 간 차이를 비교하여 새로운 객체를 생성함
- 생성된 객체를 깃 저장소에 기록함

▼ 그림 4-13 객체 생성



4. 첫 번째 커밋

➤ 파일 상태와 커밋

커밋 메시지

- 커밋은 변경된 파일 차이를 깃 저장소에 기록함
- 커밋을 할 때 생성된 객체를 기록하는 것과 동시에 이를 **구별할 수 있는 메시지를 같이 작성해야 함**
- 변화된 각 커밋 객체에 꼬리표처럼 설명을 달아 놓는다고 생각하면 됨

4. 첫 번째 커밋

➤ 파일 상태와 커밋

- 복사하는 형태로 백업할 때는 일일이 파일 이름을 수정하여 구분함
- index.htm 파일을 수정했다면 index_레이아웃수정.htm처럼 파일 이름을 변경해서 저장함
- **커밋은 파일 이름을 여러 개 사용하지 않고 하나만 가짐**
- 기존처럼 파일 이름으로 변화된 객체를 구별할 수 없음
- 그 대신 깃은 변화된 객체를 구별하고자 메시지 시스템을 도입함
- 파일 이름을 사용하지 않고, 별도로 작성한 메시지 문자열로 각 변경 객체들을 쉽게 구분할 수 있음
- **모든 커밋은 반드시 커밋 메시지를 작성해야 함**

4. 첫 번째 커밋

➤ 파일 상태와 커밋

- 커밋할 때는 commit 명령어만 사용함
- 단독으로 명령어를 입력하면 커밋 메시지 작성을 요구하며, 메시지를 작성할 수 있는 화면이 나옴

```
$ git commit
```

- 기본적으로 커밋 메시지는 vi 에디터를 사용함
- 여기에 수정 내역을 요약하여 작성함
- 메시지를 저장하면 커밋이 완료됨

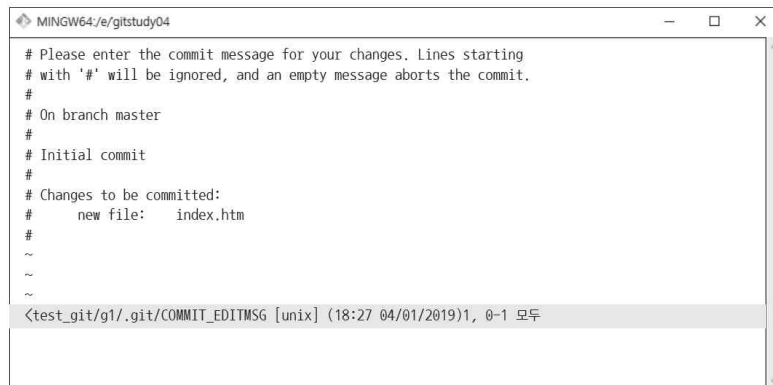
4. 첫 번째 커밋

➤ 파일 상태와 커밋

- 초보자에게 vi 에디터는 익숙하지 않음
- 커밋 메시지를 작성하려면 기본 사용법 정도는 알고 있는 것이 좋음
- 깃 배스에서 git commit 명령어를 입력하면 다음과 같이 커밋 메시지를 입력할 수 있는 창이 뜬다

4. 첫 번째 커밋

▼ 그림 4-14 vi 에디터를 사용한 커밋 메시지



```

MINGW64/e/gitstudy04
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
#
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   new file:   index.htm
#
# ~
# ~
# ~
<test_git/g1/.git/COMMIT_EDITMSG [unix] (18:27 04/01/2019)1, 0~1 모두

```

4. 첫 번째 커밋

➤ 파일 상태와 커밋

- 원하는 메시지를 입력한 후 저장함
- vi 에디터에서 새로운 내용을 입력할 때는 **[Esc]**를 누른 후 **[i]**를 누름
- 작성한 후 저장과 종료는 **[Esc]**를 누른 후 **:[w]**, **[q]**를 입력함

4. 첫 번째 커밋

➤ 파일 상태와 커밋

- 가끔씩 커밋 메시지를 작성하다 vi 에디터를 중지하고 싶을 때도 있을 것임
- 이때는 아무것도 작성하지 않고 에디터를 종료함
- `[Esc]`를 누른 후 `:[q]`를 누름
- 에디터에서 # 기호는 주석임
- 커밋할 때 꼭 주석을 삭제할 필요는 없음

예)

```
infoh@hojin MINGW64 /e/gitstudy04 (master)
$ git commit
Aborting commit due to empty commit message.
```

- 커밋 메시지를 작성하지 않아 커밋이 거부되었다는 메시지임
- vi 에디터에 아무 내용도 넣지 않고 종료하면 커밋 명령은 취소됨

4. 첫 번째 커밋

➤ 파일 상태와 커밋

- vi 에디터에서 커밋 메시지를 작성할 때는 요약 내용과 상세 내용을 분리하여 기록하면 좋음
- 보통 첫째 줄에는 '제목'을 적고, 다음 줄에는 상세 내용을 작성하곤 함
- 중간에 빈 줄로 구분해 주는 것도 좋음
- 첫째 줄을 분리하여 작성하는 것은 로그 출력을 간단하게 하기 위해서임
- 소스트리나 일부 간략한 로그들은 커밋 메시지의 첫째 줄만 표시하기 때문임

4. 첫 번째 커밋

➤ 파일 상태와 커밋

파일 등록과 커밋을 동시에 하는 방법

- 커밋을 하려면 반드시 워킹 디렉터리를 정리해야 함
- add 명령어로 추가되거나 수정된 파일들을 스테이지 영역에 등록해야 함
- 가끔씩 add 명령어를 미리 수행하는 것을 깜빡 잊을 때가 있는데, commit 명령어를 바로 수행하면 오류가 발생함
- -a 옵션을 commit 명령어와 같이 사용하면 이를 한 번에 해결할 수 있음

```
$ git commit -a
```

- -a 옵션은 커밋을 하기 전에 자동으로 모든 파일을 등록하는 과정을 미리 수행함
- 파일 등록과 커밋을 동시에 실행하는 것임

4. 첫 번째 커밋

➤ 파일 상태와 커밋

소스트리에서 커밋 메시지 작성

- 소스트리를 이용하면 좀 더 쉽게 커밋할 수 있음
- 소스트리에서 왼쪽의 파일 상태 탭을 선택하면 아래쪽에 그림 4-15와 같이 커밋 메시지를 입력할 수 있는 창이 열림

▼ 그림 4-15 소스트리에서 커밋 메시지 입력



4. 첫 번째 커밋

➤ 파일 상태와 커밋

- 커밋 메시지를 입력하고 커밋을 누름
- 바로 앞에서 제대로 커밋되지 않은 내용을 소스트리에서 커밋해보자
- 원하는 내용을 입력한 후 커밋을 누름
- 필자는 '인덱스 페이지 레이아웃'이라고 입력함
- 소스트리를 이용하면 한글 커밋 메시지도 쉽게 작성할 수 있음

4.5 커밋 확인

5. 커밋 확인

➤ 스테이지 초기화

- 먼저 터미널에서 status 명령어를 실행해서 상태를 확인함

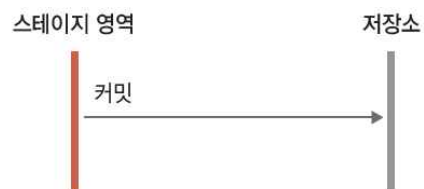
```
infoh@hojin MINGW64 /e/gitstudy04 (master)
$ git status ----- 상태 확인
On branch master
nothing to commit, working tree clean ----- 워킹 트리 정리됨
```

5. 커밋 확인

➤ 스테이지 초기화

- 이전과 달리 working tree clean 메시지를 볼 수 있음
- 커밋을 하면 스테이지 영역은 초기화됨
- 더 이상 추가된 새로운 파일과 수정된 파일이 없다는 의미임

▼ 그림 4-16 스테이지 초기화



- 항상 커밋 전후에 status 명령어로 상태를 확인하는 것이 좋음

5. 커밋 확인

➤ 로그 기록 확인

- 깃은 커밋 목록을 확인할 수 있는 log 명령어를 별도로 제공함

```
$ git log
```

- log 명령어는 시간 순으로 커밋 기록을 출력하는데, 최신 커밋 기록부터 내림차순으로 나열함

```
infoh@hojin MINGW64 /e/gitstudy04 (master)
```

```
$ git log
```

```
commit e2bce41380691b0a34aeab7db889a6c30fed8287 (HEAD -> master)
```

```
Author: hojin <infohojin@gmail.com>
```

```
Date: Sat Jan 5 18:24:50 2019 +0900
```

인덱스 페이지 레이아웃

5. 커밋 확인

➤ 로그 기록 확인

- 커밋한 후에는 습관적으로 한 번씩 log 명령어를 실행하여 기록을 확인하는 것이 좋음
- log 명령어는 다양한 커밋 기록을 확인할 수 있도록 여러 옵션을 제공함
- -help 옵션으로 확인할 수 있음

5. 커밋 확인

➤ 소스트리에서 로그 기록 확인

- 터미널로 로그 기록을 확인하는 것은 가독성이 좋지 않음
- 커밋 횟수가 많을수록 보기도 어려움
- 소스트리를 이용하면 좀 더 직관적으로 커밋 기록을 확인할 수 있음
- 소스트리 같은 GUI 도구를 사용하도록 추천하는 이유이기도 함

▼ 그림 4-17 소스트리에서의 브랜치



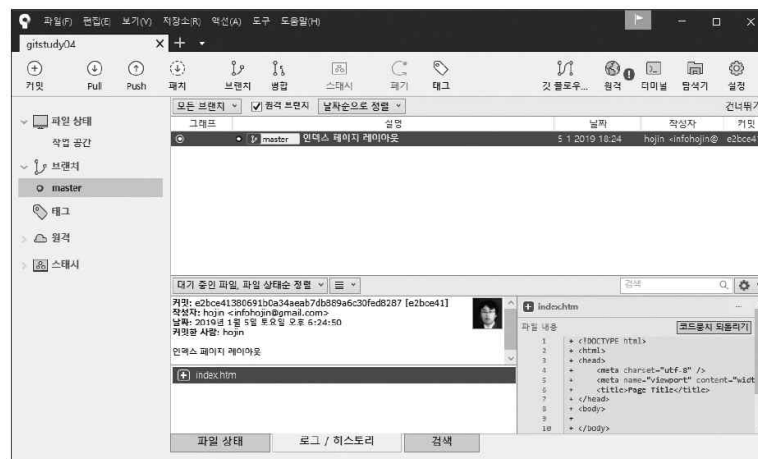
5. 커밋 확인

➤ 소스트리에서 로그 기록 확인

- 깃을 처음 생성하면 자동으로 master 브랜치 1개를 생성함
- 커밋은 master 브랜치 안에 기록됨

5. 커밋 확인

▼ 그림 4-18 소스트리에서 브랜치 확인



4.6 두 번째 커밋

6. 두 번째 커밋

> 파일 수정

- index.htm 파일의 <body></body> 태그 안에 <h1> 태그를 추가하여 간단한 이시마의 녀

infoh@hojin MINGW64 /e/gitstudy04 (master)

\$ **code index.htm** ----- VS Code 실행

index.htm

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <title>Page Title</title>
</head>
<body>
  <h1>hello GIT world!</h1>
</body>
</html>
```

6. 두 번째 커밋

> 파일 변경 사항 확인

- 터미널에서 status 명령어를 다시 한 번 실행함

infoh@hojin MINGW64 /e/gitstudy04 (master)

\$ **git status** ----- 상태 확인

On branch master

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

modified: index.htm ----- 파일 수정

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

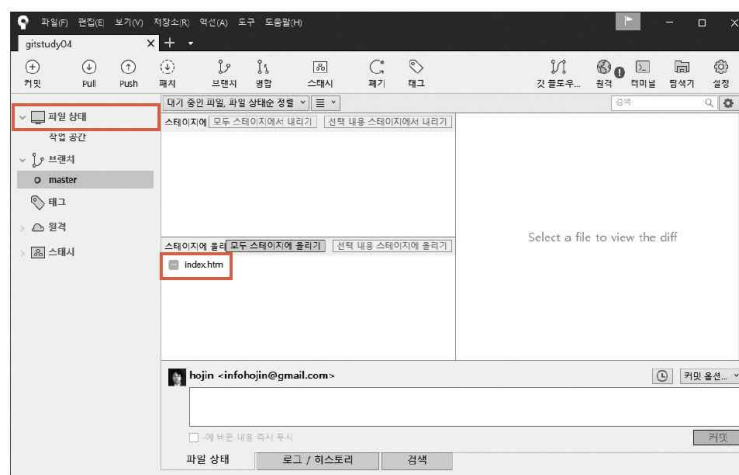
6. 두 번째 커밋

▶ 파일 변경 사항 확인

- 소스트리는 좀 더 직관적으로 화면에 출력함
- 왼쪽의 파일 상태 탭을 선택하면 스테이지에 올라가지 않은 파일 목록에 방금 수정된 파일이 다시 등록된 것을 확인할 수 있음


6. 두 번째 커밋

▼ 그림 4-19 소스트리에서 수정된 파일 감지



6. 두 번째 커밋

➤ 파일 변경 사항 확인

- 수정된 파일은 노란색 아이콘  index.htm 으로 표시함
- 소스트리는 각 파일의 생성, 변경 등 상태에 따라서 여러 가지 색의 아이콘으로 표현하므로 상태를 확인하기 편리함

6. 두 번째 커밋

➤ 수정된 파일 되돌리기

- 깃을 이용하면 수정한 파일을 커밋 전 마지막 내용으로 쉽게 되돌릴 수 있음

```
$ git checkout -- 수정파일이름
```

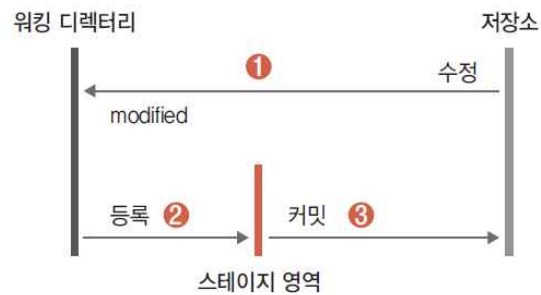
- 수정 파일을 되돌리면 이전 커밋 이후에 작업한 수정 내역은 모두 삭제함

6. 두 번째 커밋

➤ 스테이지에 등록

- 변경된 소스 코드를 커밋하는 것은 처음 파일을 생성하고 등록하는 과정과 매우 유사함

▼ 그림 4-20 스테이지에 등록



6. 두 번째 커밋

➤ 스테이지에 등록

- ❶ 기존 파일을 수정하면 해당 파일은 modified 상태로 변경됨
- 다시 워킹 디렉터리로 이동함
- ❷ 파일이 수정되면 반드시 **add** 명령어로 스테이지 영역에 재등록해야 함
- 파일을 수정할 때마다 등록 작업을 반복해야 한다는 것
- 소스트리에서 모두 스테이지에 올리기를 사용함

```
$ git add 수정파일이름
```

6. 두 번째 커밋

➤ 스테이지에 등록

- 앞 실습에 이어 수정한 파일을 스테이지에 재등록함
- 다시 한 번 status 명령어를 실행함

```
infoh@hojin MINGW64 /e/gitstudy04 (master)
$ git add index.htm ----- 스테이지에 재등록
$ git status ----- 상태 확인
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)
        modified:   index.htm ----- 파일 수정
```

- 수정된 파일 이름이 빨간색에서 녹색으로 변경된 것을 확인할 수 있음

6. 두 번째 커밋

➤ 두 번째 커밋

- vi 에디터나 소스트리에서는 커밋 메시지를 여러 줄 작성할 수 있음
- 커밋과 동시에 간단하게 한 줄짜리 커밋 메시지도 작성할 수 있음
- 커밋할 때는 -m 옵션을 사용함

```
$ git commit -m "커밋메시지"
```

6. 두 번째 커밋

> 두 번째 커밋

- 다음과 같이 커밋과 동시에 "hello git world 추가"라고 커밋 메시지도 함께 작성해보자

```
infoh@hojin MINGW64 /e/gitstudy04 (master)
$ git commit -m "hello git world 추가" ----- 커밋 메시지를 같이 입력
[master aa1dd51] hello git world 추가
1 file changed, 1 insertion(+), 1 deletion(-)
```

- 실제 작업할 때는 에디터를 여는 대신 간편한 -m 옵션을 많이 사용함

6. 두 번째 커밋

> 두 번째 커밋 확인

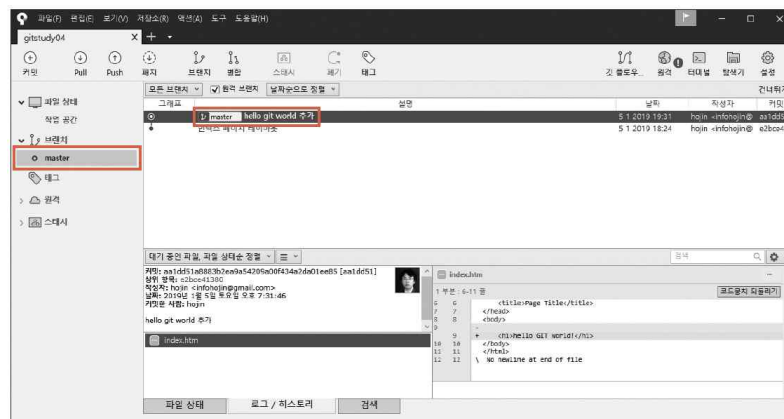
- 터미널에서 log 명령어를 실행함

```
infoh@hojin MINGW64 /e/gitstudy04 (master)
$ git log ----- 로그 확인
commit aa1dd51a8883b2ea9a54209a00f434a2da01ee85 (HEAD -> master)
Author: hojin <infohojin@gmail.com>
Date: Sat Jan 5 19:31:46 2019 +0900
    hello git world 추가 ----- 추가된 커밋 확인

commit e2bce41380691b0a34aeab7db889a6c30fed8287
Date: Sat Jan 5 18:24:50 2019 +0900
인덱스 페이지 레이아웃
```

6. 두 번째 커밋

▼ 그림 4-21 브랜치에서 추가된 로그 기록 확인



6. 두 번째 커밋

➤ 깃허브에서 확인

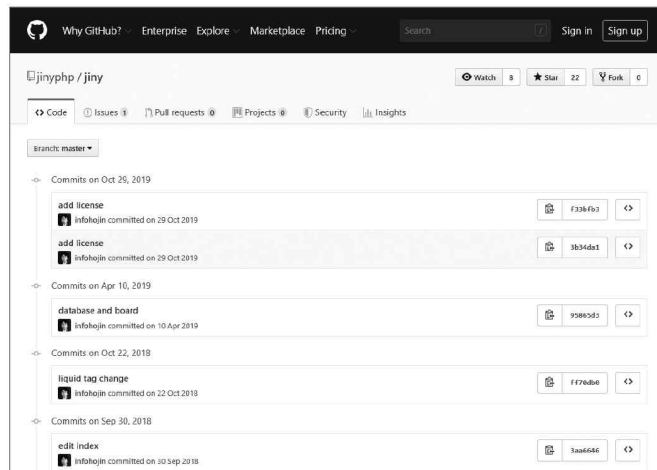
- 깃은 자신의 컴퓨터뿐만 아니라 원격 저장소를 같이 연동할 수 있음
- 대표적인 원격 저장소로는 깃허브가 있음
- 원격 저장소인 깃허브는 커밋된 횟수를 다음과 같이 표시함

▼ 그림 4-22 커밋 횟수 표시

 65 commits

6. 두 번째 커밋

▼ 그림 4-23 깃허브에서 커밋 상세 목록 확인



4.7 메시지가 없는 빈 커밋

7. 메시지가 없는 빈 커밋

➤ 메시지가 없는 빈 커밋

- 커밋을 할 때는 반드시 커밋 메시지를 같이 작성해야 함
- 의미가 없는 커밋이라 커밋 메시지를 생략하고 싶을 때도 있음
- 빈 커밋:
특별한 상황에 대비하여 깃은 메시지가 없는 커밋 작성도 허용함

7. 메시지가 없는 빈 커밋

➤ 세 번째 커밋

- 실습을 위해 index.htm 파일 내용을 좀 더 수정하여 세 번째 커밋을 하겠음
- 간략하게 <title>~</title> 사이의 내용을 JINYGIT으로 수정한 후 저장함

infoh@hojin MINGW64 /e/gitstudy04 (master)

\$ **code index.htm** ----- VS Code 실행

index.htm

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <title>JINYGIT</title>
</head>
<body>
  <h1>hello GIT world!</h1>
</body>
</html>
```


7. 메시지가 없는 빈 커밋

➤ 세 번째 커밋

- 파일을 수정한 후에는 반드시 다음과 같이 다시 수정된 파일을 스테이지 영역에 재등록함

```
infoh@hojin MINGW64 /e/gitstudy04 (master)
$ git add index.htm ----- 스테이지에 등록
```

7. 메시지가 없는 빈 커밋

➤ 세 번째 커밋

--allow-empty-message 옵션

- 터미널에서 메시지가 없는 빈 커밋을 작성하려면 --allow-empty-message 옵션을 사용함

```
infoh@hojin MINGW64 /e/gitstudy04 (master)
$ git commit --allow-empty-message -m "" ----- 커밋 메시지를 작성하지 않음
[master 42250c6] ----- 빈 커밋
1 file changed, 1 insertion(+), 1 deletion(-)
```

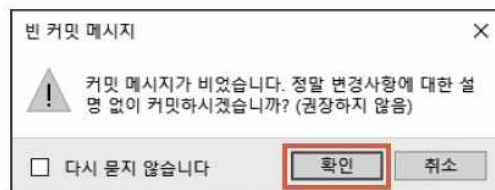
- 메시지가 없는 빈 커밋이 정상적으로 실행된 것을 확인할 수 있음

7. 메시지가 없는 빈 커밋

➤ 소스트리에서 빈 커밋

- 소스트리에서는 메시지가 없는 빈 커밋을 하기가 더 쉬움
- 예를 들어 **파일 상태** 탭을 선택한 후 아래쪽의 커밋 메시지 입력란을 비워 두고 커밋을 누르면 됨
- 경고문을 알리는 것은 커밋 작업이 반드시 메시지를 작성하는 것을 원칙으로 하고 있기 때문임

▼ 그림 4-24 빈 커밋 알림창



7. 메시지가 없는 빈 커밋

➤ 빈 커밋 확인

- 터미널에서 log 명령어를 입력함

```
infoh@hojin MINGW64 /e/gitstudy04 (master)
$ git log ----- 로그 확인
commit aa92947d350db27b604d1351930d4f809f96886e (HEAD -> master)
Author: hojin <infohojin@gmail.com> ----- 빈 커밋
Date:   Sat Jan 5 20:09:48 2019 +0900

commit aa1dd51a8883b2ea9a54209a00f434a2da01ee85
Author: hojin <infohojin@gmail.com>
Date:   Sat Jan 5 19:31:46 2019 +0900
```

7. 메시지가 없는 빈 커밋

➤ 빈 커밋 확인

hello git world 추가

commit e2bce41380691b0a34aeab7db889a6c30fed8287

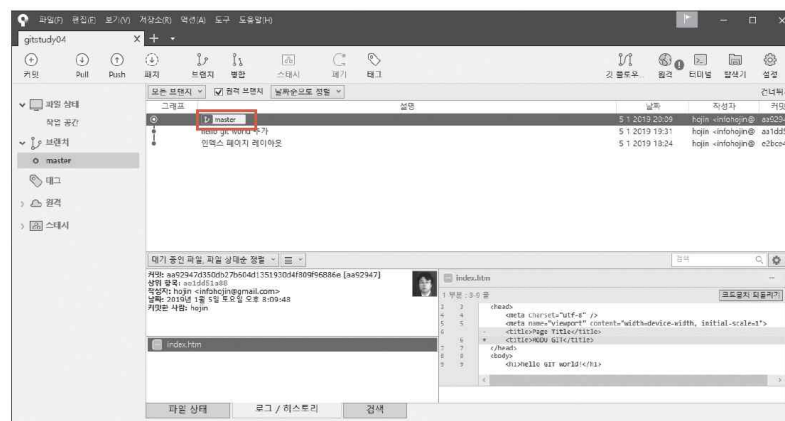
Author: hojin <infohojin@gmail.com>

Date: Sat Jan 5 18:24:50 2019 +0900

인덱스 페이지 레이아웃

7. 메시지가 없는 빈 커밋

▼ 그림 4-25 소스트리에서 빈 커밋 확인



4.8 커밋 아이디

103

8. 커밋 아이디

> 커밋 아이디

- 다음과 같이 터미널에서 log 명령어를 실행하면 로그 정보를 볼 수 있음

예

```
infoh@hojin MINGW64 /e/gitstudy04 (master)
$ git log
commit aa92947d350db27b604d1351930d4f809f96886e (HEAD -> master) ----- 아이디
Author: hojin <infohojin@gmail.com>
Date: Sat Jan 5 20:09:48 2019 +0900
이하 생략
```

- 각 커밋에는 aa92947d350db27b604d1351930d4f809f96886e 같은 이상한 영문과 숫자가 있는데 이를 **커밋 아이디**라고 함
- 커밋 아이디는 특정 커밋을 가리키는 절대적 이름이고, 명시적 참조 값임
- 커밋 아이디는 다수의 커밋을 구분할 수 있는 키이며, 브랜치나 태그 등에도 많이 사용함

8. 커밋 아이디

➤ SHA1

- 커밋 아이디가 이렇게 복잡한 영어와 숫자로 된 이유는 것이 SHA1이라는 해시 알고리즘을 사용하기 때문임
- SHA1 해시키 값은 40자리의 복잡한 hexa 값으로 되어 있음
- 같은 스테이지 영역의 변경된 내용을 기반으로 SHA1 해시키를 생성함
- SHA1 해시는 **중복되지 않은 고유의 키**를 생성할 수 있는 장점이 있음
- 것이 SHA1 해시를 이용하는 것은 콘텐츠 추적과 분산형 저장 관리를 운영하면서 충돌을 방지하기 위함

8. 커밋 아이디

➤ 단축키

- SHA1 해시키는 매우 복잡한 모양의 영어와 숫자로 되어 있음
- 해시는 40자리의 16진수로 입력하다 실수로 잘못 입력할 가능성이 높음
- SHA1 해시키는 매우 큰 숫자이기 때문에 고유 접두사로 간략하게 사용할 수 있는데, 해시의 앞쪽 **7자만으로도 중복을 방지**하면서 전체 키 값을 사용할 수 있음
- 해시는 매우 큰 값으로 웬만해서는 앞쪽 숫자 값이 변경되는 경우가 드뭄
- 전체의 키 값을 다 사용하는 것이 좋겠지만, 앞자리 몇 개만 사용해도 충분함

4.9 커밋 로그

107

9. 커밋 로그

➤ 커밋 로그

- 깃은 터미널 기반의 응용 프로그램임
- 깃의 로그는 저장소 커밋 기록들을 확인할 수 있음
- 커밋 메시지, 아이디도 확인할 수 있고, 브랜치 경로 등을 분석할 수 있는 옵션들도 제공함

9. 커밋 로그

➤ 간략 로그

- 커밋 메시지를 여러 줄 작성했다면 일반적인 로그 정보를 복잡하게 느낄 수 있음
- 로그 옵션 중에서 **--pretty=short**를 사용하면 로그를 출력할 때 첫 번째 줄의 커밋 메시지만 출력함

```
infoh@hojin MINGW64 /e/gitstudy04 (master)
$ git log --pretty=short ----- 로그 확인
commit aa92947d350db27b604d1351930d4f809f96886e (HEAD -> master)
Author: hojin <infohojin@gmail.com>

commit aa1dd51a8883b2ea9a54209a00f434a2da01ee85
Author: hojin <infohojin@gmail.com>
    hello git world 추가

commit e2bce41380691b0a34aeab7db889a6c30fed8287
Author: hojin <infohojin@gmail.com>
    인덱스 페이지 레이아웃
```

9. 커밋 로그

➤ 간략 로그

- 특정 커밋의 상세 정보도 확인할 수 있음
- 특정 커밋의 상세 정보를 확인하고 싶다면 show 명령어를 사용함

```
$ git show 커밋ID
```

9. 커밋 로그

➤ 특정 파일의 로그

- 전체 커밋과 달리 특정 파일의 로그 기록만 볼 수도 있음
- log 명령어 뒤에 파일 이름을 적어 주면 됨

```
infoh@hojin MINGW64 /e/gitstudy04 (master)
$ git log index.htm ----- 파일의 로그 확인
commit aa92947d350db27b604d1351930d4f809f96886e (HEAD -> master)
Author: hojin <infohojin@gmail.com>
Date: Sat Jan 5 20:09:48 2019 +0900
```

9. 커밋 로그

➤ 특정 파일의 로그

- log 명령어의 옵션은 매우 다양함
- 소스트리를 이용하면 log 명령어와 복잡한 옵션을 사용하지 않아도 쉽게 로그 흐름을 파악할 수 있는 장점이 있음

log 명령어의 여러 옵션

- -p 옵션: diff 기능(수정한 라인 비교)을 같이 포함하여 출력할 수 있음
- --stat 옵션: 히스토리를 출력함
- --pretty=oneline 옵션: 각 커밋을 한 줄로 표시함

4.10 diff 명령어

113

10. diff 명령어

➤ diff 명령어

- diff 명령어는 커밋 간 차이를 확인함
- 보통 리눅스나 macOS 같은 유닉스 계열의 운영 체제에는 유용한 diff 명령어가 있음
- 깃 또한 초기 시작은 리눅스 커널을 개발하려는 것이었기 때문에 유사한 기능을 하는 diff 명령어를 제공함

10. diff 명령어

➤ 파일 간 차이

- git의 장점은 파일들의 수정 이력을 커밋이라는 형태로 구분할 수 있다는 것임
- git은 커밋으로 파일들의 수정 내역을 추적함
- 파일 수정:
 - 파일 내용 일부가 수정, 추가, 삭제되는 것을 의미함
- 개발하면서 수많은 소스 코드가 수정, 추가, 삭제되곤 함
- git은 커밋을 기준으로 이러한 파일들의 수정 이력을 비교해 볼 수 있는 diff 기능을 제공함
- diff 기능으로 파일의 수정 및 변경 내역을 쉽게 파악할 수 있음

10. diff 명령어

➤ 워킹 디렉터리 vs 스테이지 영역

- 아직 add 명령어로 파일을 추가하지 않은 경우, 워킹 디렉터리와 스테이지 영역 간 변경 사항을 비교할 수 있음
- index.htm 파일을 열어 <h1> 태그 밑에 <h2> 태그와 내용을 추가함

```
infoh@hojin MINGW64 /e/gitstudy04 (master)
```

```
$ code index.htm
```

10. diff 명령어

➤ 워킹 디렉터리 vs 스테이지 영역

index.htm

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <title>JINYGIT</title>
</head>
<body>
  <h1>hello GIT world!</h1>
  <h2>깃을 이용하면 소스의 버전 관리를 쉽게 할 수 있습니다.</h2>
</body>
</html>
```

10. diff 명령어

➤ 워킹 디렉터리 vs 스테이지 영역

- diff 명령어를 실행해보자

infoh@hojin MINGW64 /e/gitstudy04 (master)

\$ **git diff** ----- 스테이지 vs 워킹 디렉터리 비교

diff --git a/index.htm b/index.htm

index f5097d9..56af0de 100644

--- a/index.htm

+++ b/index.htm

@@ -7,5 +7,6 @@

</head>

<body>

<h1>hello GIT world!</h1>

+ <h2>깃을 이용하면 소스의 버전 관리를 쉽게 할 수 있습니다.</h2> ----- 추가

</body>

</html>

\ No newline at end of file

10. diff 명령어

➤ 워킹 디렉터리 vs 스테이지 영역

- 워킹 디렉터리 내용과 스테이지 내용의 차이점을 출력함
- 이번에는 변경한 파일을 add 명령어로 추가하여 스테이지 영역에 등록함

```
infoh@hojin MINGW64 /e/gitstudy04 (master)
$ git add index.htm
```

- 다음 다시 diff 명령어를 수행함
- 아무 내용도 출력되지 않음

```
infoh@hojin MINGW64 /e/gitstudy04 (master)
$ git diff
```

- 이것은 등록 과정을 거쳐 워킹 디렉터리의 수정 내역을 스테이지 영역에 반영했기 때문임
- 아무 내용도 출력되지 않음

10. diff 명령어

➤ 커밋 간 차이

- 스테이지 영역에 있는 수정된 파일을 아직 커밋하지 않았다면, **최신 커밋과 변경 내용을 비교하여 볼 수 있음**
- HEAD는 마지막 커밋을 가지고 있는 포인터임

10. diff 명령어

➤ 커밋 간 차이

- 워킹 디렉터리와 마지막 커밋을 비교해보자

```
infoh@hojin MINGW64 /e/gitstudy04 (master)
$ git diff head ----- head 포인터 입력
diff --git a/index.htm b/index.htm
index f5097d9..56af0de 100644
--- a/index.htm
+++ b/index.htm
@@ -7,5 +7,6 @@
</head>
<body>
    <h1>hello GIT world!</h1>
+   <h2>깃을 이용하면 소스의 버전 관리를 쉽게 할 수 있습니다.</h2>
</body>
</html>
\ No newline at end of file
```

10. diff 명령어

➤ 커밋 간 차이

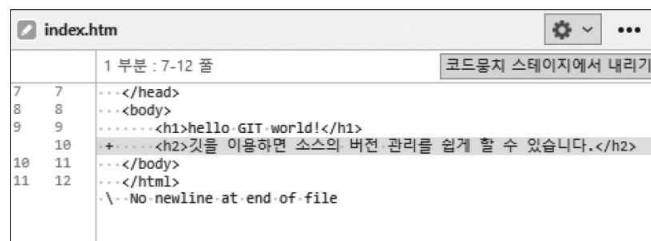
- HEAD:
 - 최근의 커밋 중 가장 마지막 커밋의 위치를 가리키는 값임
- HEAD를 이용하면 최신 커밋과 이전 커밋을 비교하여 출력할 수 있음

10. diff 명령어

➤ 소스트리에서 간단하게 변경 이력 확인

- 소스트리를 이용하면 이를 좀 더 직관적으로 알아볼 수 있게 표현할 수 있음
- 예를 들어 소스트리에서 수정한 파일 이름을 클릭하면 오른쪽 창에 파일 변경 내역들이 출력됨

▼ 그림 4-26 소스트리에서 변경 이력 확인 1

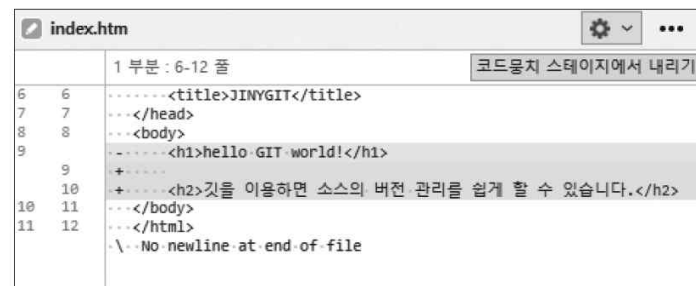


10. diff 명령어

➤ 소스트리에서 간단하게 변경 이력 확인

- 예를 들어 index.htm에서 <h1>hello GIT world!</h1>을 삭제한 후 다시 확인해 보면 삭제된 항목은 붉은색으로 표시함

▼ 그림 4-27 소스트리에서 변경 이력 확인 2



10. diff 명령어

➤ 소스트리에서 간단하게 변경 이력 확인

- + 표시는 새롭게 추가된 내용을 의미함
색상은 녹색으로 표시됨
- - 표시는 삭제된 내용을 의미함
색상은 붉은색으로 표시됨
- diff 기능은 나중에 자신이 수정한 부분들을 쉽게 찾을 수 있도록 도와줌
- 또 다른 사람들과 협업하여 개발하는 과정에서 코드를 리뷰할 때 전체를 파악하는 것보다 수정된 부분만 확인하면 되므로 좀 더 쉽게 검토할 수 있음

10. diff 명령어

➤ diff 내용을 추가하여 커밋

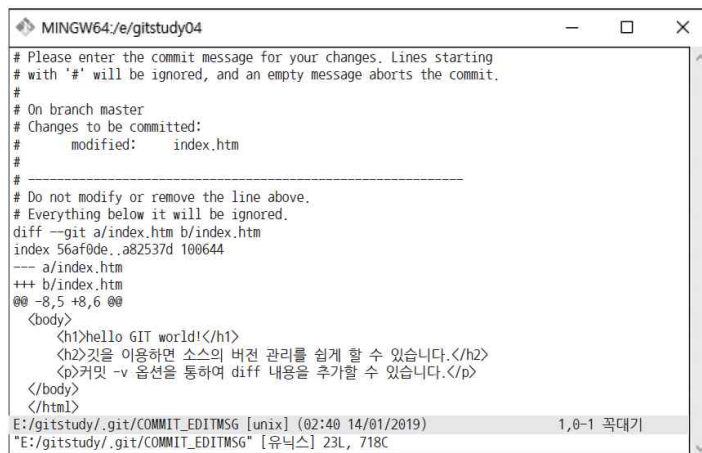
- 커밋 메시지를 작성할 때 -v 옵션을 같이 사용하면 vi 에디터에서 diff 내용을 추가할 수 있음

예

```
infoh@hojin MINGW64 /e/gitstudy04 (master)
$ git commit -v ----- diff 내용 추가
```

10. diff 명령어

▼ 그림 4-28 커밋 메시지를 작성할 때 diff 내용 추가



```

# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
#
# On branch master
# Changes to be committed:
#   modified:   index.htm
#
-----
# Do not modify or remove the line above.
# Everything below it will be ignored.
diff --git a/index.htm b/index.htm
index 56af0de..a82537d 100644
--- a/index.htm
+++ b/index.htm
@@ -8,5 +8,6 @@
<body>
  <h1>hello GIT world!</h1>
  <h2>깃을 이용하면 소스의 버전 관리를 쉽게 할 수 있습니다.</h2>
  <p>커밋 -v 옵션을 통하여 diff 내용을 추가할 수 있습니다.</p>
</body>
</html>
E:/gitstudy/.git/COMMIT_EDITMSG [unix] (02:40 14/01/2019) 1,0-1 쪽대기
"E:/gitstudy/.git/COMMIT_EDITMSG" [유닉스] 23L, 718C
  
```

4.11 정리

11. 정리

➤ 정리

- 커밋 작업은 깃에서 소스 코드를 관리하는 첫 단추임
- 너무 많은 코드를 수정한 후 커밋하는 것보다는 작은 단위로 코드를 수정한 후 커밋하는 것을 추천함
- 커밋의 수정 부분이 적을수록 검토하기 쉽고, 오류도 쉽게 찾을 수 있음