3장 깃 개념 잡기

1

3.1 깃 저장소 생성

2

1. 깃 저장소 생성

➢ **깃 저장소 생성**

- 깃은 작성된 소스 코드 파일의 모든 변경 사항을 관리함
- 깃은 이러한 변경 사항을 전용 저장소(repository)(리포지터리)에 저장함
- 이 저장소는 일반적으로 사용하는 폴더와 유사하지만, 조금 차이가 있음
- 깃의 동작 방식을 이해하려면 저장소 동작 원리를 확실히 알아야 함

1. 깃 저장소 생성

➢ **폴더와 깃 저장소**

- 컴퓨터의 파일과 폴더는 운영 체제의 파일 시스템에 의존하여 동작함
- 파일 시스템은 하드디스크 같은 장치에 데이터를 저장하고 관리함
- 그중 폴더는 파일 여러 개를 하나로 관리할 수 있는 논리적 개념임
- 마치 파일을 그룹으로 묶어 놓은 것과 같음

1. 깃 저장소 생성

➢ **폴더와 깃 저장소**

- 깃 저장소는 외형적으로 폴더와 유사함
- 사용자 입장에서는 일반 폴더와 깃 저장소를 구별 없이 모두 동일하게 사용할 수 있음
- 깃 저장소는 내부적으로 구조가 다름
- 깃 저장소에는 별도의 숨겨진 영역(숨겨진 폴더)이 있는데, 여기에 버전 관리 시스템(VCS, Version Control System)에 필요한 파일 변경 이력을 기록함
- 저장소는 프로젝트의 모든 리비전(revision)(개정)과 히스토리를 가진 데이터베이스와 같음
- 일반적인 폴더와 깃 저장소 차이점은 숨겨진 영역이 있는지 여부임

1. 깃 저장소 생성

➢ **초기화**

- 저장소를 생성하려면 먼저 초기화 작업이 필요함
- 초기화란:

  이미 존재하는 폴더에 초기화 명령어로 VCS 관리를 위한 숨겨진 영역을 생성하는 작업
- 깃 초기화를 해보자
- 먼저 터미널(깃 배시)을 실행함

1. 깃 저장소 생성

➢ **초기화**

- 터미널은 텍스트로 명령어를 입력할 수 있는 대화창임
- 깃 배시 터미널 프로그램 외에도 윈도에 기본 내장된 CMD, powerShell 등을 사용해도 됨
- 윈도 메뉴 말고 바탕화면의 깃 배시 아이콘으로 실행해도 됨

▼ **그림 3-1 깃 배시 아이콘**

1. 깃 저장소 생성

➢ **초기화**

- 실습에 필요한 새 폴더를 하나 만듦
- 기존에 있던 폴더에서 시작해도 괜찮음

```
$ mkdir jinygit03 -------- 새 폴더 만들기
$ cd jinygit03 -------- 만든 폴더로 이동
```

1. 깃 저장소 생성

➢ **초기화**

- mkdir 명령어:
  make directory의 약어로 셸(터미널)에서 폴더를 만드는 명령
- cd 명령어:
  change directory의 약어로 디렉터리를 이동하는 명령
- 명령 프롬프트를 사용하는 것이 익숙하지 않다면, 윈도 탐색기에서 마우스 오른쪽 버튼을 누른 후 **새로 만들기 > 폴더** 메뉴를 선택하여 만들어도 좋음

1. 깃 저장소 생성

➢ **초기화**

- 명령 프롬프트에서 원하는 경로로 이동하기 불편하다면, 윈도 탐색기를 사용하여 원하는 폴더로 이동한 후 해당 폴더에서 터미널을 열 수 있음
- 원하는 폴더에서 마우스 오른쪽 버튼을 누른 후 Git Bash Here 메뉴를 선택함

1. 깃 저장소 생성

▼ 그림 3-2 지정된 폴더에서 깃 배시 열기

1. 깃 저장소 생성

➢ **초기화**

- 이제 터미널에서 다음 초기화 명령어를 입력함
- 깃 명령어는 보통 git 키워드와 명령어를 함께 입력함
- 옵션을 추가할 수도 있음

```
$ git init 경로명
```

- git init 명령어는 기존 폴더에 **숨겨진 영역(숨겨진 폴더)을 추가**함
- 숨겨진 영역을 추가함으로써 **깃 저장소로 변경**되는 것임

1. 깃 저장소 생성

➢ **초기화**

- 초기화 명령어를 입력할 때 **경로명**을 입력하지 않으면, **현재 폴더에서 초기화**됨
- 우리는 원하는 폴더로 이동한 상태이므로 경로명을 입력하지 않고 실행할 것임
- 정상적으로 초기화되었다면 "Initialized empty~" 같은 메시지를 출력함

```
$ git init -------- 저장소 초기화
Initialized empty Git repository in E:/jinygit03/.git/
```

- 현재 폴더를 의미하는 .을 사용할 수도 있음

```
$ git init .
```

1. 깃 저장소 생성

➢ **초기화**

- 깃 초기화는 완전히 비어 있는 폴더나 기존에 사용하던 폴더에서 모두 가능함

▼ **그림 3-3 깃 초기화**

1. 깃 저장소 생성

➢ **초기화**

- 초기화된 저장소에서 '숨긴 항목'을 볼 수 있게 허용하면, 숨겨진 영역을 확인할 수 있음
- 윈도 탐색기에서 숨겨진 파일을 표시하려면 설정을 변경해야 함
- **탐색기 > 보기** 메뉴를 선택한 후 **숨긴 항목**을 체크함

▼ **그림 3-4 숨긴 항목 허용**

- git init 명령어는 기본적으로 로컬 저장소를 생성하며, 다양한 옵션을 추가로 제공함
- 추가 옵션을 사용하여 원격 저장소도 초기화할 수 있음

1. 깃 저장소 생성

➢ **숨겨진 폴더 = .git 폴더**

- git init 명령어를 사용하여 일반적인 폴더를 깃이 관리할 수 있는 저장소로 변경함
- 깃 저장소를 초기화한다는 것은 별도의 숨겨진 폴더를 하나 추가하고 환경 설정 파일을 생성하는 것임
- 생성된 숨겨진 폴더를 확인해보자
- 먼저 깃 배시 터미널이 실행된 상태에서 ls 명령어로 파일 목록을 출력함

```
$ ls
```

- ls 명령어는 파일 목록을 출력하는 리눅스 명령어임
- 초기화된 폴더에서 ls 명령어를 입력하면 아무 내용도 출력되지 않음

1. 깃 저장소 생성

➢ **숨겨진 폴더 = .git 폴더**

- 일반적인 ls 명령어는 숨겨진 폴더까지 확인할 수 없음
- 이번에는 -a 옵션을 추가하여 입력
- -a 옵션은 폴더 안의 숨겨진 파일을 같이 출력하라는 의미

```
$ ls -a
./  ../  .git/
```

1. 깃 저장소 생성

➢ **숨겨진 폴더 = .git 폴더**

- 이전 결과와 달리 목록이 화면에 출력됨
- 목록을 보니 **.git**이라는 숨겨진 폴더가 하나 있음
- 보통 폴더 이름 앞에 **점(.)이 있으면 숨겨진 폴더**를 의미
- 숨겨진 폴더인 **.git** 폴더에는 깃 저장소에 필요한 모든 **뼈대 파일**이 담겨 있음
- 뼈대 파일들은 깃 초기화를 통하여 자동 생성됨

1. 깃 저장소 생성

➢ **숨겨진 폴더 = .git 폴더**

- 깃의 숨겨진 폴더(.git)는 매우 중요함
- 이 폴더에는 깃으로 관리되는 모든 파일 및 브랜치 등 이력을 기록함
- 컴퓨터에서 깃 저장소를 통째로 복사하고자 할 때는 숨겨진 **.git 폴더까지 같이 복사해야 함**
- 로컬 컴퓨터에서 .git 폴더를 삭제하거나 함께 복제하지 않으면 깃의 모든 이력은 없어짐
- 일반적인 폴더 파일과 동일함

1. 깃 저장소 생성

➢ **소스트리와 연결**

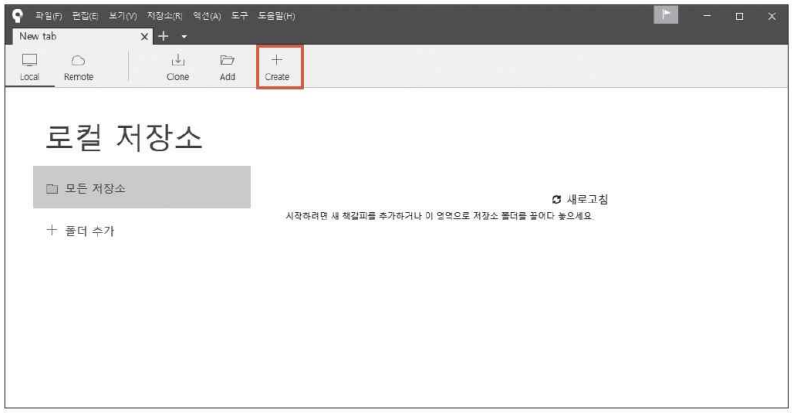
- 로컬 컴퓨터에 생성된 깃 저장소를 소스트리와 연동하여 이력을 쉽게 관리할 수 있음

새 저장소를 생성하여 연결

- 위쪽 + Create 버튼을 클릭하여 새 저장소를 생성함
- 새 저장소 생성은 폴더 생성 및 초기화 명령어인 git init를 동시에 실행하는 것과 같음

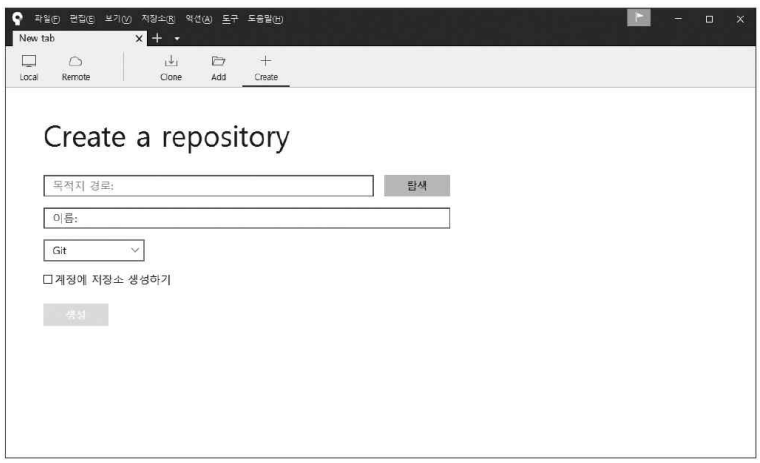
1. 깃 저장소 생성

▼ 그림 3-5 새 저장소 생성



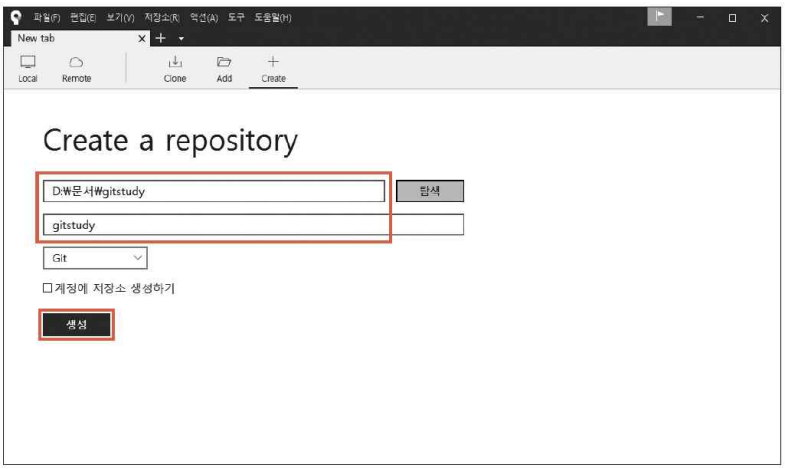
1. 깃 저장소 생성

▼ 그림 3-6 새 저장소 생성



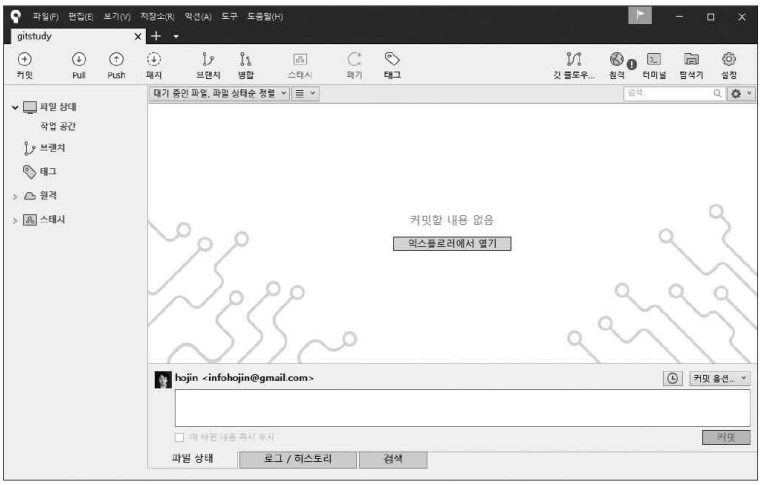
1. 깃 저장소 생성

▼ 그림 3-7 gitstudy 폴더 생성



1. 깃 저장소 생성

▼ 그림 3-8 깃 상태 확인



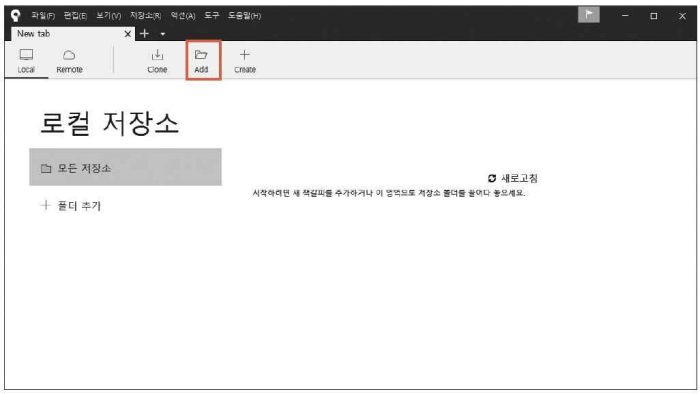
1. 깃 저장소 생성

➢ **소스트리와 연결**

기존 저장소와 연결

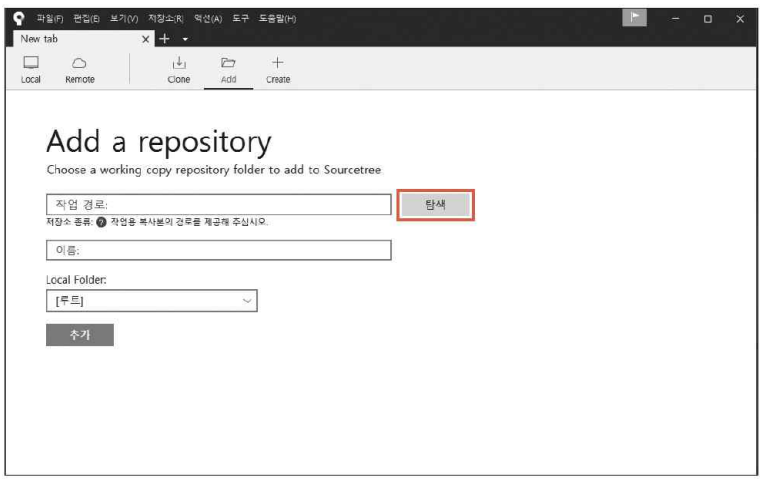
- 기존에 사용하던 로컬 저장소를 소스트리와 연결할 수 있음

▼ **그림 3-9 Add 버튼 클릭**



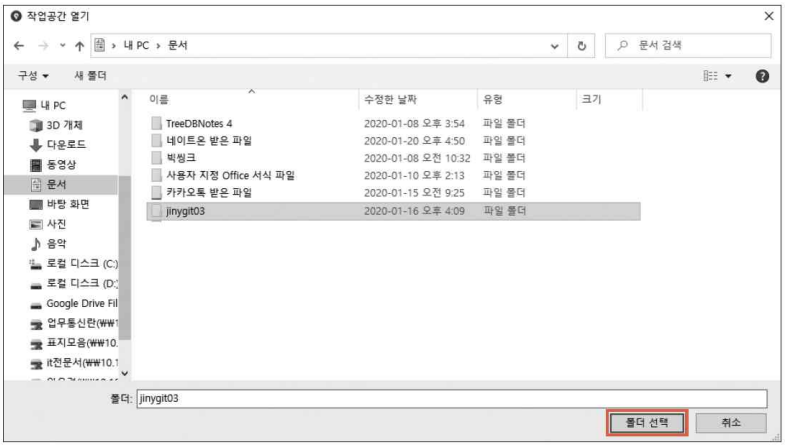
1. 깃 저장소 생성

▼ **그림 3-10 [탐색] 누르기**



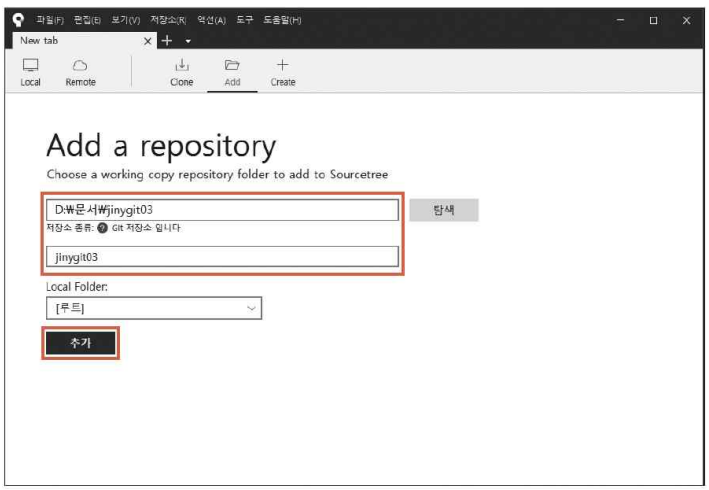
1. 깃 저장소 생성

▼ 그림 3-11 원하는 폴더 선택



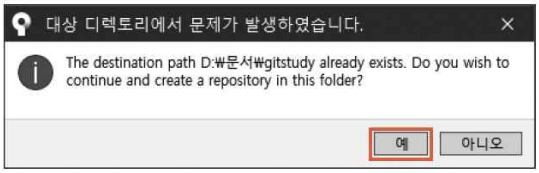
1. 깃 저장소 생성

▼ 그림 3-12 로컬 저장소를 선택한 상태



1. 깃 저장소 생성

▼ 그림 3-13 기존 깃 저장소 폴더를 선택할 때 알림 화면



1. 깃 저장소 생성

▼ 그림 3-14 깃 상태 확인

1. 깃 저장소 생성

➢ **소스트리와 연결**

- 이제 깃 저장소를 터미널과 소스트리를 모두 사용하여 동시에 관리할 수 있음
- 깃 명령어가 익숙하지 않다면 처음에는 소스트리를 사용하는 것이 편리함
- 깃 명령어에 익숙하다면 터미널이 좀 더 빠름
- 깃 터미널로 작업한 모든 결과 역시 소스트리로 확인할 수 있음

3.2 워킹 디렉터리

32

2. 워킹 디렉터리

➢ **워킹 디렉터리**

- 깃의 동작을 이해하려면 먼저 워킹 디렉터리(working directory) 개념을 알아야 함
- 워킹 디렉터리는 다른 용어로 워킹 트리(working tree)라고도 함

2. 워킹 디렉터리

➢ **워킹 디렉터리란?**

- 깃은 VCS의 특성 때문에 저장 공간을 논리적으로 분리함
- 깃을 처음 학습할 때는 이러한 논리적인 공간 분리 때문에 어렵게 느끼곤 함
- 깃은 저장 공간을 크게 '작업을 하는 공간(working)'과 '임시로 저장하는 공간(stage)', '실제로 저장하여 기록하는 공간(repository)'으로 나뉨
- 논리적으로 공간을 분리하는 것은 깃의 동작과 이력을 좀 더 효율적으로 처리하기 위함
- 워킹 디렉터리는 '작업을 하는 공간'을 의미함
- 말 그대로 로컬 저장소에 접근할 수 있으며, 실제로 파일을 생성하고 수정하는 공간임
- 단순하게 파일을 저장하는 공간이라고 생각하면 됨

2. 워킹 디렉터리

➢ **파일의 untracked 상태와 tracked 상태**

- 깃이 다른 VCS보다 뛰어난 것은 지정된 파일들의 모든 것을 추적하는 관리 시스템이기 때문임
- 깃은 워킹 디렉터리에 있는 파일들을 '추적됨'과 '추적되지 않음' 상태로 구분함

2. 워킹 디렉터리

➢ **파일의 untracked 상태와 tracked 상태**

untracked 상태

- 실제 작업 중인 파일은 워킹 디렉터리 안에 있음
- 워킹 디렉터리는 현재 작업 중인 소스 코드를 담고 있으며, 운영 체제도 워킹 디렉터리 안에 있는 파일들만 접근하고 수정할 수 있음
- 워킹 디렉터리는 사용자 작업 공간이라고 생각하면 됨
- 이 공간에 파일을 추가하거나 수정했다고 해서 깃이 자동으로 관리해 주지는 않음
- 워킹 디렉터리에 새로 생성된 파일은 모두 **추적되지 않음(untracked)** 상태
- 이 파일을 관리하려면 깃에 추적하라고 통지해 주어야 함
- 통지하지 않은 파일은 깃에서 따로 추적하지 않음

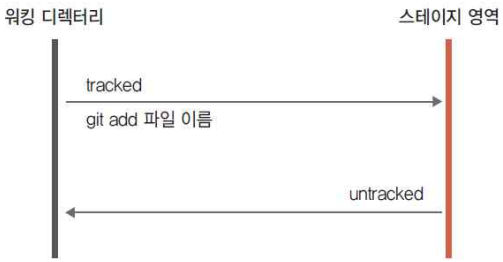
2. 워킹 디렉터리

➢ **파일의 untracked 상태와 tracked 상태**

tracked 상태

- 워킹 디렉터리에 새 파일을 추가하면 **'추적되지 않음(untracked)' 상태**
- 워킹 디렉터리 안에 추적되지 않는 상태의 파일들은 별도로 명령어를 실행하여 **추적(tracked) 상태로 변경**해 주어야 함

▼ **그림 3-15 git add 명령어를 실행하여 추적 상태로 변경**



2. 워킹 디렉터리

➢ **파일의 untracked 상태와 tracked 상태**

- 깃이라고 모든 파일을 자동으로 완벽하게 관리할 수는 없음
- 수많은 파일을 모두 자동으로 처리해야 한다면 시스템에 엄청난 부하가 발생됨
- 깃은 요청받은 파일들만 **추적 관리**함
- 사실 깃 입장에서는 어떤 파일이 정말로 추적 관리가 필요한지 알 수 없음
- 추적하는 파일들은 **tracked** 상태로 표시함
- 깃이 tracked와 untracked 개념을 사용하는 것은 시스템 부하를 줄이고, 좀 더 효율적으로 파일 이력을 관리하기 위해서임

2. 워킹 디렉터리

➢ **파일의 untracked 상태와 tracked 상태**

- 실제로 개발 작업을 할 때 워킹 디렉터리의 파일들은 아주 빈번하게 수정됨
- 개발자에게 관리할 파일 목록들을 제출해 달라고 요청하는 것이 더 현명함
- 요청받은 파일들만 이력을 관리한다면 매번 파일 목록을 추적하려고 많은 리소스를 낭비할 필요가 없음
- 추적 상태:
  관리할 파일 목록에 등록된 상태

3.3 스테이지

40

3. 스테이지

➢ **스테이지**

- 깃은 여러 단계의 논리적인 저장 공간을 가지고 있음
- 스테이지(stage):
  '임시로 저장하는 공간'을 의미함
- 스테이지 영역은 워킹 디렉터리에서 제출된 **tracked 파일들을 관리**함
- **이 영역은 나중에 배울 커밋 작업과 연관이 매우 많음**

3. 스테이지

➢ **스테이지 = 임시 영역**

- 스테이지는 워킹 디렉터리와 '실제로 저장하여 기록하는 공간' 사이에 있는 **임시 영역**임
- 깃은 워킹 디렉터리에서 작업이 끝난 파일을 스테이지로 잠시 복사함
- 스테이지가 임시 영역이라고 해서 파일의 콘텐츠 내용을 직접 가지고 있지는 않음
- 단지 커밋하려는 파일의 추적 상태 정보들만 기록함

▼ **그림 3-16 스테이지는 커밋하려는 파일의 추적 정보만 기록**

3. 스테이지

➢ **스테이지 = 임시 영역**

- 임시 영역인 스테이지를 별도로 운영하는 것은 커밋을 빠르게 처리하기 위해서임
- '실제로 저장하여 기록하는 공간'인 저장소는 스테이지 영역에서 가리키는 파일 내용을 기반으로 변경된 차이점만 기록함
- 파일들의 스테이지 상태는 status 명령어로 확인 가능함
- 깃의 git ls-files 같은 명령어로도 확인 가능함

예

```
$ git status
$ git ls-files --stage
```

- 스테이지 영역에 등록된 파일들은 또 다시 stage 상태와 unstage 상태로 구분됨
- 버전 관리에서 제외하고 싶은 파일이 있다면 .gitignore 파일에 등록함

3. 스테이지

➢ **파일의 stage 상태와 unstage 상태**

- 워킹 디렉터리에 있는 tracked 상태의 파일들은 스테이지 영역과 긴밀한 상관관계를 맺음
- 스테이지 영역으로 등록된 모든 파일은 untracked 상태에서 tracked 상태로 변경됨
- 스테이지는 워킹 디렉터리 안에 있는 파일들의 추적 상태를 관리하는 역할을 수행함
- 스테이지 영역은 파일을 stage 상태와 unstage 상태로 구분함
- 깃이 변화 이력을 기록하려면 파일들의 **최종 상태가 stage 상태**여야 함
- unstage 상태라면 파일에 변화가 있다는 것을 의미함
- 스테이지 영역에 있는 파일과 워킹 디렉터리 안에 있는 **파일 내용에 차이가 있을 때는 unstage 상태**가 됨

3. 스테이지

➢ **파일의 stage 상태와 unstage 상태**

- 넓게 보면 아직 스테이지 영역으로 등록하지 않은 워킹 디렉터리 안의 파일도 unstage 상태라고 생각할 수 있음
- 이때는 unstage 상태이자 동시에 untracked 상태임
- unstage 상태라고 해서 실제 파일이 없어지는 것은 아님
- 단지 파일이 수정되어 임시적으로 **스테이지 목록에서 제외**된 것임
- git add 명령어를 사용하면 스테이지에 다시 추가할 수 있음

3. 스테이지

➢ **파일의 modified 상태와 unmodified 상태**

- 코드를 변경한다는 것은 워킹 디렉터리에서 파일을 **수정**하는 것을 의미함
- 파일이 수정되면 워킹 디렉터리와 스테이지 간 내용이 일치하지 않음
- 스테이지는 수정한 파일과 원본 파일을 구분하려고 **수정함(modified) 상태와 수정하지 않음(unmodified) 상태**로 표현함
- 파일 수정 작업은 스테이지 영역과 긴밀한 상관관계를 맺음

3. 스테이지

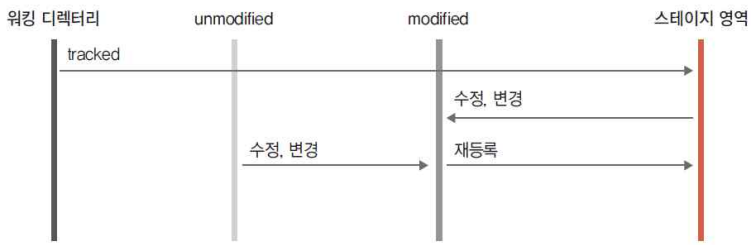
➢ **파일의 modified 상태와 unmodified 상태**

modified 상태

- 스테이지에 등록된 파일은 깃이 추적 관리함
- 깃이 실제로 기록한 파일이며, 사실상 버전을 의미함
- 파일 수정은 개발 과정에서 뗄 수 없는 작업임
- 개발 작업에서 수많은 코드가 변경되고, 깃을 사용하면 이 변경 내역은 영구적으로 기록됨
- 깃은 tracked 상태인 파일만 수정 여부를 관리할 수 있음
- tracked 상태인 파일이 수정되면 스테이지는 파일 상태를 modified 상태로 변경함

3. 스테이지

▼ **그림 3-17 파일 수정 여부 확인**



3. 스테이지

➢ **파일의 modified 상태와 unmodified 상태**

- 수정된 파일은 스테이지에서 잠시 **제외**됨
- 깃은 수정 여부만 체크해 주기 때문에 modified 상태로 변경된 파일은 스테이지로 **재등록**해야 함
- 수정된 파일을 스테이지 영역으로 다시 적용하려면 git add 명령어로 재등록함

3. 스테이지

➢ **파일의 modified 상태와 unmodified 상태**

unmodified 상태

- unmodified 상태는 tracked 상태이면서 스테이지에서 한 번도 수정하지 않은 원본 상태를 의미함
- 수정하지 않은 파일들은 재등록하지 않아도 됨
- 스테이지에 등록한 후 어떤 수정도 하지 않았다면 unmodified 상태임
- 깃은 파일의 수정 여부를 체크하고, 스테이지 영역의 갱신 작업 여부를 작업자에게 알려 줌

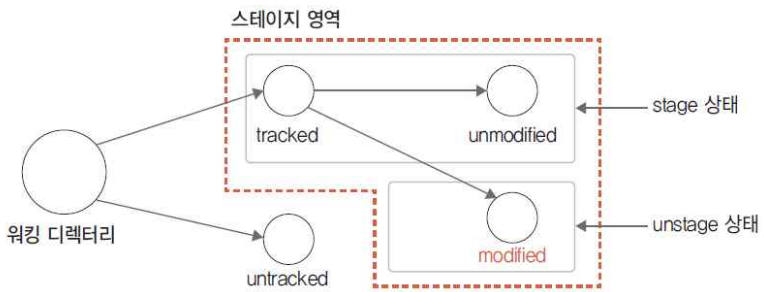
3. 스테이지

➢ **파일의 modified 상태와 unmodified 상태**

- 워킹 디렉터리에서 등록 명령(git add 명령어)을 실행하면 스테이지에 등록됨
- 이때 자동으로 tracked 상태가 됨
- 파일을 수정하지 않으면 계속 stage 상태에 머무름
- 파일이 수정되면 modified 상태가 되고, 스테이지에서 떨어져 나와 unstage 상태가 됨
- unstage 상태의 파일은 워킹 디렉터리에 잠시 담아 둠
- 이때 다시 등록 명령을 실행하면 stage 상태로 변경됨

3. 스테이지

▼ **그림 3-18 워킹 디렉터리와 스테이지 상태 구분**



3.4 스테이지

53

4. 파일의 상태 확인

➢ **파일의 상태 확인**

- 상태 개념은 깃의 분리된 저장 영역인 워킹 디렉터리와 스테이지, 추적 여부를 의미함
- 깃이 이렇게 다양한 저장 영역을 구분해서 가지고 있는 것은 파일들의 상태를 효율적으로 모니터링하기 위해서임

4. 파일의 상태 확인

➢ **status 명령어로 깃 상태 확인**

- 파일을 생성하고 수정한다는 것은 변화를 의미함
- 깃은 각 저장 영역에서 일어나는 다양한 변화를 감시함
- 변화를 감지하고 상태 메시지를 출력함
- status 명령어를 사용하면 깃의 상태 메시지를 확인할 수 있음
- status 명령어는 많이 사용하는 깃 명령어 중 하나임

4. 파일의 상태 확인

➢ **status 명령어로 깃 상태 확인**

- 터미널(깃 배시)에서 status 명령어를 입력함

```
$ git status -------- 상태 확인
On branch master
No commits yet -------- 커밋이 없다는 메시지
nothing to commit (create/copy files and use "git add" to track) -------- 변경된 내용이 없다는 메시지
```

- 우리는 처음 깃 저장소를 생성하고, 실습 이후에 아무 작업도 하지 않았음
- status 명령어를 실행하면 이처럼 변경된 내용과 커밋이 없다고 메시지를 출력할 것임
- 나중에 커밋 명령을 수행하면 status 명령어로 파일들의 변경된 상태를 확인할 수 있음

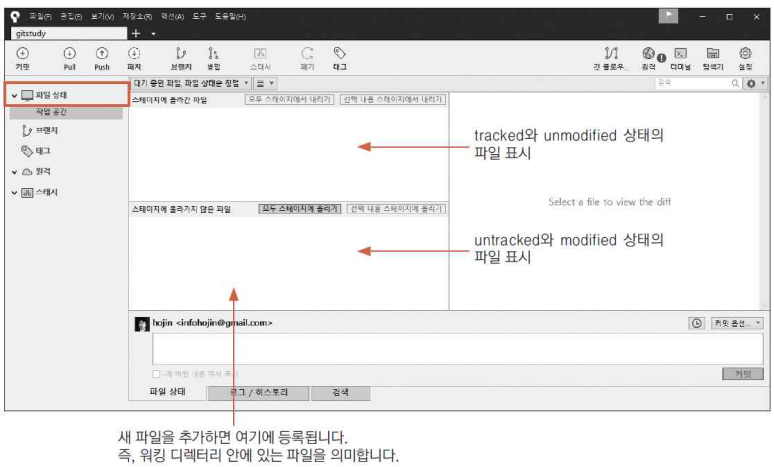
4. 파일의 상태 확인

➢ **소스트리에서 깃 상태 확인**

- 소스트리를 이용하면 콘솔에서 status 명령어를 입력하지 않고도 바로 **깃 상태**를 확인
  할 수 있음
- 소스트리를 실행한 후 왼쪽의 **파일 상태** 탭을 선택함

4. 파일의 상태 확인

▼ **그림 3-19 깃 상태 확인**



4. 파일의 상태 확인

➢ **소스트리에서 깃 상태 확인**

- 소스트리는 깃의 파일 상태를 두 영역으로 표시함
- 스테이지의 tracked와 unmodified 상태는 **스테이지에 올라간 파일**에 표시함
- untracked와 modified 상태는 **스테이지에 올라가지 않은 파일**에 표시함
- 새 파일을 추가하면 untracked 상태임
- **스테이지에 올라가지 않은 파일** 목록에 출력함
- 스테이지에 올라가지 않은 파일이란 워킹 디렉터리 안에 있는 파일을 의미함
- 소스트리는 직관적으로 깃에 추가된 파일을 확인할 수 있음

3.5 파일 관리 목록에서 제외: .gitignore 3.5 git

60

5. 파일 관리 목록에서 제외: .gitignore

➢ **파일 관리 목록에서 제외: .gitignore**

- 깃은 tracked 상태인 모든 것을 추적 관리함
- 파일뿐만 아니라 서브 폴더와 그 안의 파일들도 포함함
- 디렉터리 전체가 모두 관리 대상임
- 프로젝트를 하다 보면 모든 파일을 추적하지 않아야 하는 경우도 있음

5. 파일 관리 목록에서 제외: .gitignore

➢ **파일 관리 목록에서 제외: .gitignore**

- 실제 작업하다 보면, 워킹 디렉터리에 불필요한 파일이 생성되거나 보안에 민감한 파일들이 있을 수 있음
- 로컬 저장소를 혼자만 사용한다면 이러한 파일들은 신경 쓰지 않아도 됨
- 자신의 저장소를 다른 사람들과 공유한다면 이 파일들은 분리해서 관리해야 함
- 깃으로 관리하고 싶지 않은 파일과 폴더는 별도의 .gitignore 설정 파일 안에 나열해서 적어 줌

5. 파일 관리 목록에서 제외: .gitignore

➢ **.gitignore 파일**

- .gitignore는 git과 ignore(무시하다)의 합성어임
- 워킹 디렉터리 안에 .gitignore 파일을 생성함
- 이 파일은 앞에 점(.)이 있어 숨겨진 파일로 관리됨
- .gitignore 파일은 깃에서 관리하지 않는 파일들의 목록을 가지고 있음
- 깃은 이 파일에 작성된 목록들을 추적하지 않음
- 로컬 저장소를 서버로 전송하거나 다른 사람과 공유할 때도 이를 분리하여 처리함
- .gitignore 파일은 텍스트 에디터를 이용하여 간단하게 작성할 수 있음
- 특별한 도구 없이 파일 이름만 .gitignore로 만들면 됨
- 깃에서 제외할 파일 목록을 직접 적어 주거나 규칙을 사용하여 나열할 수 있음
- .gitignore 파일을 작성할 때는 저장소 폴더의 최상위 디렉터리에 두어야 함

5. 파일 관리 목록에서 제외: .gitignore

➢ **.gitignore 파일 표기법**

- 파일에서 #으로 시작하는 줄은 주석으로 처리함
- # 없이 완전한 파일 이름을 적어 주면 그 파일은 깃의 관리 대상에서 제외됨
- 이때 경로가 있다면 경로명도 같이 입력해야 함

```
# DB 접속 파일을 제외함
dbinfo.php
```

- 셸 글로빙(globbing):
  애스터리스크(*) 기호를 사용하여 패턴을 정의할 수 있음
  * 기호는 모든 문자열을 대체할 수 있음
- 글로빙 문자를 사용하여 패턴을 확장함

```
# 오브젝트 파일은 제외함
*.obj
```

5. 파일 관리 목록에서 제외: .gitignore

➢ **.gitignore 파일 표기법**

- ignore 패턴을 작성할 때 반드시 추적 관리를 제외하는 파일만 작성하는 것은 아님
- 제외하지 않는 파일과 필요한 파일은 파일 이름 앞에 느낌표(!)를 사용함
- 느낌표는 부정을 의미하는 not과 같음

```
# 환경 설정 파일은 제외하면 안 됨
!config.php
```

5. 파일 관리 목록에서 제외: .gitignore

➢ **.gitignore 파일 표기법**

- 운영 체제별로 디렉터리를 표현하는 방법이 다름
- 깃에서 디렉터리를 표현할 때는 리눅스와 같이 슬래시(/) 기호를 사용함

```
# 현재 디렉터리 안에 있는 파일 무시
/readme.txt

# /pub/ 디렉터리 안의 모든 것을 무시
/pub/

# doc 디렉터리 아래의 모든 .txt 파일 무시
doc/**/*.txt
```

- 깃은 glob 패턴을 지원하기 때문에 정규 표현식을 응용하여 작성 규칙을 넣을 수도 있음

3.6 깃 저장소 복제

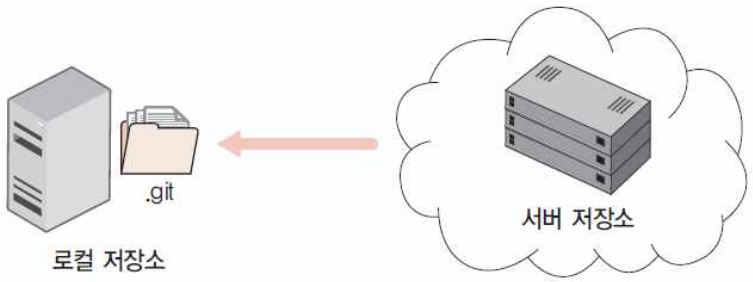
67

6. 깃 저장소 복제

➢ **깃 저장소 복제**

- 외부에 있는 기존 프로젝트(깃허브, 비트버킷)를 기반으로 저장소를 생성하고 싶다면 외부 저장소를 복제해서 생성할 수 있음
- 외부 저장소를 이용하여 로컬 저장소를 생성하는 것을 '깃 저장소 복제'라고 함

▼ **그림 3-20 외부의 깃 저장소 복제**



6. 깃 저장소 복제

➢ **공개 저장소**

- 깃은 다수의 사람과 코드를 공유하고 협업하여 개발하는 도구임
- 이미 깃을 기반으로 하는 공개 저장소가 여럿 있음
- 대표적으로 깃허브, 비트버킷 같은 깃 호스팅 사이트가 있음
- 깃 호스팅 서비스는 공개된 저장소와 비공개된 저장소를 모두 지원함
- 공개된 저장소는 누구나 복제하여 코드를 내려받을 수 있음
- 요즘은 오픈 소스가 활성화되어 저장소를 공개하고 있음
- 이미 수많은 오픈 소스를 깃으로 관리하고, 공개 저장소를 이용하여 배포함

6. 깃 저장소 복제

➢ **다운로드 vs 복제**

- 일반적으로 공개된 소스 코드를 얻으려면 웹 사이트에서 압축 파일을 내려받음
- 소스 코드를 내려받는다는 것은 해당 코드의 최종 복사본을 내 컴퓨터로 가져오는 것임
- 내려받기는 깃의 이력을 포함한 저장 영역까지 내려받는 것은 아님

▼ **그림 3-21 공개된 소스 코드의 최종 복사본 내려받기**

6. 깃 저장소 복제

➢ **다운로드 vs 복제**

- 이와 달리 깃을 이용하여 저장소를 복제하면, 최종 코드뿐만 아니라 중간에 커밋 같은 변화의 모든 이력도 같이 내려받을 수 있음
- 일부 코드를 변경하여 기여하는 것도 가능할 것임

6. 깃 저장소 복제

➢ **복제 명령어**

- 깃의 저장소를 복제하는 명령어는 clone임
- 복제하려면 공개된 저장소의 URL이 필요함
- 복제할 때 폴더 이름을 지정하지 않으면 공개 저장소에서 사용된 폴더와 동일한 이름으로

  새 폴더를 만듦
- 다른 이름으로 복제하길 원한다면 새 폴더 이름을 추가 인자로 적어 줌

```
$ git clone 원격저장소URL 새폴더이름
```

6. 깃 저장소 복제

➢ **복제 명령어**

- 다음은 필자가 운영하는 jinyphp 오픈 소스 사이트의 주소를 이용하여 저장소를 복제하는 예
- 새 폴더 이름을 적지 않아 공개 저장소에서 사용된 폴더와 동일한 이름으로 새 폴더를 만듦

예

```
$ git clone https://github.com/jinyphp/jiny -------- 저장소 복제
Cloning into 'jiny'...
remote: Enumerating objects: 975, done.
remote: Total 975 (delta 0), reused 0 (delta 0), pack-reused 975
Receiving objects: 100% (975/975), 4.98 MiB ¦ 3.67 MiB/s, done.
Resolving deltas: 100% (307/307), done.
```

6. 깃 저장소 복제

➢ **복제 명령어**

- git clone 명령어를 사용하면 깃은 자동으로 깃 서버에 접속함
- 저장소의 모든 소스 코드를 자동으로 내려받음
- 깃은 저장소 안에 있는 파일들과 .git 리포지터리를 기반으로 이력을 관리함
- 복제한 후에는 복제된 폴더 이름을 그대로 사용하지 않아도 됨
- 필요에 따라 깃의 폴더 이름을 변경 해도 괜찮음

3.7 정리

7. 정리

➢ **정리**

- 앞으로 깃을 학습하려면 이 장에서 배운 저장소 개념을 잘 알고 있어야 함
- 각 용어들을 구분하고 이해하는 것도 필요함(특히 워킹 디렉터리와 스테이지 영역)
- 각 장의 기능을 학습하다 용어 이해가 부족하다고 느낄 때 이 장을 다시 읽어 보는 것도 좋음