11. Git 내부 구조 및 원리

`.git` 디렉토리 구조

`.git/`은 Git 저장소의 **모든 내부 정보**를 저장하는 핵심 디렉토리다.
이 안에 Git의 **스냅샷, 참조, 로그, 설정, 상태**가 모두 존재한다.

🗂️ `.git` 주요 구조

```
.git/
├── HEAD
├── config
├── description
├── index
├── hooks/
├── info/
├── logs/
│   ├── HEAD
│   └── refs/
├── objects/
│   ├── info/
│   └── pack/
├── refs/
│   ├── heads/
│   ├── tags/
│   └── remotes/
└── packed-refs
```

📌 주요 파일 및 디렉토리 설명

1. `HEAD`

- 현재 체크아웃된 브랜치를 가리키는 **심볼릭 참조**
- 내용 예시:

```
ref: refs/heads/main
```

---

2. `config`

- Git 저장소의 **로컬 설정** 파일
- 사용자 정보, remote, 푸시 방식 등이 저장됨

---

3. `description`

- 기본 Git에서는 거의 사용되지 않음 (bare repo에서 웹뷰용 설명용도)

---

4. `index`

- **스테이징 영역(Stage)**
- 어떤 파일이 커밋에 포함될지 추적함
- `git add` 시 이 파일에 기록됨

---

5. `hooks/`

- 커밋/푸시 등 이벤트에 반응하는 **스크립트 자동 실행** 디렉토리
- 예: `pre-commit`, `pre-push`, `post-merge` 등
- 실전에서는 **CI 도구나 코드 포맷터와 연계**

---

6. `info/`

- `.gitignore`보다 **로컬 전용** 무시 규칙을 작성할 수 있는 `exclude` 포함

---

7. `logs/`

- 모든 **브랜치 이동, 커밋, 리베이스 이력** 기록
- `git reflog`는 여기 있는 로그를 기반으로 동작함

---

8. `objects/`

- Git의 진짜 저장소
- 모든 커밋, 트리, 블롭(파일), 태그가 **SHA-1 해시로 저장**
- 예:

```
objects/ab/cd123... ← ab는 디렉토리, cd123은 파일 앞부분
```

- 구조:
  - `blob`: 파일 내용
  - `tree`: 폴더와 파일 구조
  - `commit`: 메타데이터와 트리 포인터
  - `tag`: 태그 객체

---

9. `refs/`

- 브랜치, 태그, 원격 브랜치의 **참조 정보**
- 내부 디렉토리:
  - `refs/heads/` : 로컬 브랜치
  - `refs/tags/` : 태그
  - `refs/remotes/` : 원격 브랜치

---

10. `packed-refs`

- 태그나 참조가 많아지면 `refs/` 안에 저장하지 않고
  하나의 **압축 파일**로 관리하는 인덱스 파일

🧠 요약 표

경로	역할
`HEAD`	현재 브랜치 참조
`index`	스테이징 영역
`config`	로컬 설정
`hooks/`	자동 실행 스크립트
`logs/`	브랜치/커밋 기록 추적
`objects/`	모든 Git 데이터의 실제 저장소
`refs/`	브랜치/태그/원격 참조
`packed-refs`	많은 참조를 압축 저장
`info/exclude`	로컬 전용 ignore 설정

🔍 예시: 커밋 한 번 할 때 내부에서 일어나는 일

1. `git add file.txt`
   - `index` 파일에 `file.txt`의 hash가 추가됨
2. `git commit -m "add file"`
   - `blob` 객체 생성 → `tree` 객체 생성 → `commit` 객체 생성
   - 이들이 `objects/`에 저장됨
   - `refs/heads/main`이 새 커밋을 가리킴
   - `logs/HEAD`에 커밋 로그 추가됨

✅ 추가 고급 항목

항목	설명
`alternates`	다른 저장소의 객체 공유 시 사용 (대용량 모듈)
`info/grafts`	커밋 히스토리를 수동으로 연결
`shallow`	얕은 클론 시 사용하는 깊이 정보
`worktrees/`	Git Worktree 사용 시 생기는 외부 작업 공간 추적

🔚 마무리

`.git` 디렉토리는 Git이 파일 시스템처럼 동작할 수 있도록
모든 버전 정보를 객체 기반으로 기록하고 추적하는 핵심 구조이다.
이걸 이해하면 Git이 왜 "스냅샷" 기반이고, 왜 "되돌리기", "브랜치 전환",
"reflog 복구"가 자유로운지 이해할 수 있다.

Blob, Tree, Commit 객체

✅ 1. Git 저장소의 핵심 개념

Git은 **변경점(diff) 기반 저장소가 아님** →
매번 **전체 스냅샷을 찍는 시스템**.

그러나 동일한 내용(해시)이 이미 저장되어 있으면
중복 저장 없이 동일 객체를 재사용함.

모든 데이터는 `.git/objects/`에 **압축된 형태(zlib)**로 저장됨.
Git이 사용하는 4종류 객체:

종류	역할
Blob	실제 **파일 내용** 저장
Tree	디렉토리 구조와 Blob/Tree 참조 저장
Commit	하나의 **스냅샷 + 메타데이터** 저장
Tag	태그 객체 (선택적)

🗂️ 2. Blob 객체

정의

> **파일의 내용 자체** (내용만 저장, 파일명은 저장 X)

특징

- 내용이 동일한 파일은 동일한 blob 해시 사용
- 실제 파일명이 없는 단일 파일 덩어리

예시

```
git hash-object file.txt
# 출력 → SHA-1 해시
```

Git 내부에서:

```
.git/objects/ab/cd1234... → blob 파일
```

Blob 구조

```
blob 23\0Hello Git world!
```

🌳 3. Tree 객체

정의

> **디렉토리(폴더)**의 상태 및 구성 관리
> 어떤 파일명과 어떤 Blob/Tree를 포함하고 있는지 저장

역할

- **파일명과 권한 정보**, 참조하는 blob/tree SHA-1 포함
- 디렉토리 = Tree 객체

예시 구조 (가상적):

```
tree 56cd12...
├── 100644 file1.txt → blob abcd12...
├── 100644 file2.txt → blob bcde23...
└── 040000 src/ → tree 9f8a23...
```

Git 확인 예시

```
git ls-tree HEAD
```

출력:

```
100644 blob a1b2c3d4 file.txt
040000 tree e5f6g7h8 src
```

📝 4. Commit 객체

정의

> Git이 **스냅샷(버전)과 메타데이터**를 저장하는 객체

역할

- **루트 Tree 객체**를 가리킴 → 전체 프로젝트 상태 정의
- **부모 Commit SHA** 저장 (없으면 최초 커밋)
- **작성자(author), 커밋터(committer), 시간, 메시지** 포함

구조 예시

```
commit 254a...
tree 56cd12...      # 루트 트리 참조
parent a1b2c3...    # 부모 커밋 (없으면 최초 커밋)
author John Doe <john@example.com> 1700000000 +0900
committer John Doe <john@example.com> 1700000001 +0900

Initial commit
```

🔗 5. Blob → Tree → Commit 관계

전체 관계도

```
[Commit]
  ↓
[Tree] (루트 디렉토리)
  ↓
├── [Tree] (src/)
│     ↓
│     ├── [Blob] file1.c
│     └── [Blob] file2.h
├── [Blob] README.md
└── [Blob] LICENSE
```

핵심 개념

- Commit = "이 시점의 프로젝트 상태는 이 Tree임!"
- Tree = "이 폴더에는 이런 파일/디렉토리가 있음"
- Blob = "이 파일의 내용은 이것!"

→ Git은 파일명을 Blob에 저장하지 않음! Tree가 파일명과 blob를 매핑함

🧠 6. 왜 이런 구조를 쓸까?

이유	설명
중복 방지	동일한 파일 내용(blob)은 여러 커밋에서 재사용 가능
빠른 비교	commit → tree → blob 따라가면서 빠른 비교 가능
스냅샷	모든 commit은 완전한 프로젝트 상태(Tree) 가짐
무결성	모든 참조가 SHA-1 해시 기반으로 검증 가능 (변조 불가)

✅ 7. Git plumbing 명령어로 확인하기

```
git cat-file -t <해시>    # 객체 타입(blob/tree/commit) 확인
git cat-file -p <해시>    # 객체 내용 출력
```

예:

```
git cat-file -p HEAD
git cat-file -p <tree 해시>
git cat-file -p <blob 해시>
```

🗂️ 8. 정리

객체	설명
Blob	파일의 내용 (파일명은 없음)
Tree	디렉토리 구조 및 Blob/Tree 참조 (파일명 저장)
Commit	버전(스냅샷) 정보 (루트 Tree + 메타데이터 + 부모 커밋)

→ Commit = 특정 시점의 전체 Tree(디렉토리) 구조 + 커밋 메타정보

🌳 한마디로:

👉 Git은
Commit → Tree → Blob
이라는 **Immutable Snapshot 체인**을 쌓아가며 동작하는 시스템이다.
→ 이 덕분에 브랜치, revert, rebase, cherry-pick 등이 "참조 변경"만으로 빠르게 가능.

HEAD 포인터의 의미

기본 개념

- **HEAD** = 현재 작업 중인 **브랜치의 최신 커밋을 가리키는 포인터**
- Git은 **브랜치 = 일종의 포인터(Reference)** → HEAD는 그 브랜치를 참조하는 특별한 포인터임

예를 들어:

```
HEAD → main → 커밋 C3
```

→ 현재 브랜치가 `main`이고, `main`이 가리키는 커밋이 C3 → HEAD는 `main`을 통해 C3 커밋을 참조

HEAD 구조적으로

- `.git/HEAD` 파일에 현재 HEAD가 어디를 가리키고 있는지 기록됨

```
ref: refs/heads/main
```

→ HEAD는 현재 `refs/heads/main` → 즉 main 브랜치를 가리킴

HEAD의 역할

✅ Git 명령어들이 **HEAD가 가리키는 커밋을 기준으로 동작**
✅ 새로운 커밋 생성 시 → HEAD → 브랜치 → 새 커밋으로 이동
✅ 체크아웃 시 → HEAD가 다른 브랜치로 이동

---

detached HEAD

- HEAD가 브랜치를 가리키는 대신 → 특정 커밋 자체를 직접 가리키는 상태

```
git checkout <커밋 해시>
```

→ detached HEAD 상태 → HEAD가 직접 해당 커밋을 참조

→ 이 상태에서 새 커밋을 하면 **브랜치가 아닌 HEAD가 직접 이동** → 브랜치로 연결되지 않음 → 주의 필요

HEAD 이동 예시

일반적인 commit

```
git commit -m "New commit"
```

→ HEAD → 브랜치 → 새 커밋으로 이동

checkout

```
git checkout develop
```

→ HEAD → refs/heads/develop → 해당 브랜치로 이동

detached HEAD

```
git checkout a1b2c3d
```

→ HEAD → 커밋 `a1b2c3d` 직접 참조 → detached HEAD 상태

정리

상태	HEAD가 가리키는 대상
정상 상태	브랜치 참조 (refs/heads/브랜치명)
detached HEAD	특정 커밋 (커밋 해시 직접 참조)

결론

- HEAD는 Git에서 **현재 작업 위치(작업 기준점)를 나타내는 핵심 포인터**
- HEAD → 브랜치 → 커밋 구조로 관리됨
- HEAD가 어떻게 움직이는지 이해하면 → **checkout, commit, reset, rebase** 동작을 정확히 이해할 수 있음
- **detached HEAD 상태에서 작업 시 주의** → 브랜치에 연결되지 않은 상태에서 커밋하면 해당 커밋을 잃을 수 있음

SHA-1 해시 구조

SHA-1 해시란?

- **SHA-1** = Secure Hash Algorithm 1
- **40자리 16진수 문자열**로 표현되는 **160-bit 해시값**
- 입력 데이터(임의의 길이)를 고정 길이의 해시값으로 변환

예시:

```
e69de29bb2d1d6434b8b29ae775ad8c2e48c5391
```

Git에서 SHA-1의 역할

- Git은 **모든 객체(커밋, 트리, 블롭, 태그)** 를 내부적으로 **SHA-1 해시 기반으로 저장**
- Git은 **내용 기반 주소 지정(content-addressed storage)** 모델 → 내용이 같으면 해시도 동일

핵심 아이디어:

✅ Git은 **파일명이나 경로가 아니라 파일 내용 자체의 해시**로 관리
✅ 커밋도 내용 기반 해시 → Git은 전체 프로젝트의 스냅샷을 해시로 보장

SHA-1 사용 위치

객체	설명	해시로 구분됨
blob	파일 내용	내용 해시
tree	디렉토리 구조	트리 구조 + 이름 + 모드 해시
commit	커밋 (트리+메타데이터+부모 커밋 정보)	전체 스냅샷 해시
tag	태그 객체	태그 메타정보 포함 해시

SHA-1 해시 구성 원리 (커밋 예시)

커밋 해시 생성 원리

커밋 해시는 다음 내용의 해시:

```
commit <content length>\0
tree <트리 해시>
parent <부모 커밋 해시> (없을 수도 있음)
author <작성자 정보>
committer <커밋터 정보>
commit message
```

→ 이 전체 문자열을 SHA-1 해시로 변환 → 커밋 해시 생성됨

왜 강력한가?

- 커밋 해시는 **내용 + 트리 구조 + 부모 커밋까지 전부 포함** → 커밋 하나라도 변경 시 전체 해시가 달라짐
- Git은 해시를 통해 **데이터 무결성 보장** → 위변조 탐지 가능
- 동일한 내용은 동일한 해시 → 중복 저장 없음 → 저장 공간 최적화

SHA-1 해시 특성

✅ 고정 길이 (160-bit → 40자리 16진수 문자열)
✅ 동일한 입력 → 항상 동일한 해시 출력
✅ 작은 변경 → 완전히 다른 해시 출력 → **충돌 회피에 강함**
✅ Git은 이 해시를 **객체 고유 ID**로 사용

SHA-1 한계

- SHA-1은 원래 **암호학적 해시 함수**였지만 **충돌 공격 가능성**이 발견됨
- Git은 기본적으로 SHA-1 사용하지만 → **SHA-256으로 전환 준비 진행 중 (Git 2.29 이후 지원)**
- 실무에서는 SHA-1 해시의 충돌 확률이 매우 낮기 때문에 여전히 안전하게 사용됨 (Git 내부 설계상 안전성 보완됨)

Git CLI에서 SHA-1 확인 예시

```
git log --oneline
```

→ 커밋 해시(짧은 형식) 표시됨

```
git rev-parse HEAD
```

→ 현재 HEAD의 **전체 SHA-1 해시 출력**

```
git cat-file -p <해시>
```

→ 해당 해시 객체의 원본 내용 확인 가능

정리

개념	설명
SHA-1 해시	Git 내부 객체의 고유 식별자 (40자리 16진수 문자열)
대상 객체	blob(파일), tree(디렉토리), commit, tag
생성 원리	객체 내용 기반으로 SHA-1 해시 생성
장점	내용 기반 주소 지정, 중복 방지, 무결성 보장
한계	SHA-1 충돌 가능성 발견 → Git은 SHA-256 전환 중

결론

- Git은 SHA-1 해시를 기반으로 **모든 데이터를 안전하게 식별하고 관리**
- 해시 기반 저장 덕분에 Git은:
  - 빠르고 효율적이고
  - 중복을 방지하고
  - 데이터 무결성을 강하게 보장
- Git을 **파일 기반이 아니라 "스냅샷 + 해시 기반 구조"로 이해하는 것이 핵심**

staging area 동작 방식

Staging Area(스테이징 영역)란?

기본 개념

- Staging Area = **커밋 후보군** 을 모아두는 중간 단계 공간
- 흔히 **index** 라고도 불림 → 내부적으로 `.git/index` 파일로 관리됨
- Git은 3가지 레이어 구조로 동작함:

```
Working Directory → Staging Area → Repository (Commit)
```

영역	상태
Working Directory	실제 파일 수정
Staging Area	커밋 예정 파일 모음
Repository	이미 커밋된 상태

Staging Area의 역할

✅ **어떤 파일/변경만 커밋할지 선택** 가능
✅ 파일의 일부만 선택적으로 스테이징 가능 (hunk 단위 스테이징 가능)
✅ 커밋 전에 최종 상태를 검토할 수 있음
✅ 효율적인 커밋 단위 구성 가능 (작고 논리적인 커밋 작성 지원)

기본 명령 흐름

1️⃣ 수정 → Working Directory

파일 수정 → **Working Directory 변경됨**

```
git status
# → modified 상태 표시됨
```

2️⃣ add → Staging Area 이동

```
git add <file>
```

- 해당 파일이 Staging Area로 이동됨
- 이후 커밋하면 **Staging Area에 있던 내용만 커밋됨**

```
git status
# → Changes to be committed 영역에 표시됨
```

3️⃣ commit → Repository 반영

```
git commit -m "Message"
```

- Staging Area의 내용만 커밋됨 → Repository에 기록됨

Staging Area 없이 바로 커밋?

```
git commit -a -m "Message"
```

- `-a` 옵션 → **tracked된 파일만 자동 add 후 commit**
- 신규 파일(`git add` 하지 않은 파일)은 포함 안 됨

Staging Area의 실제 활용 예시

✅ 파일 10개 수정 → 5개만 우선 커밋하고 나머지는 나중에 커밋하고 싶을 때
✅ 한 파일 안에서도 일부만 커밋하고 싶은 경우:

```
git add -p <file>
```

→ Hunk 단위(변경 블록 단위)로 선택 가능 → 매우 유용한 고급 기능

내부 구조 (깊이 이해)

- Staging Area = `.git/index` 파일
- Working Directory의 현재 상태와 Staging Area의 상태가 다를 수 있음
- 커밋 시 → Staging Area의 상태를 스냅샷으로 Commit 객체로 만듦
- 즉 **Staging Area = 다음 Commit의 설계도**

핵심 명령 정리

명령어	동작
`git add <file>`	파일을 Staging Area에 추가
`git reset HEAD <file>`	Staging Area에서 파일 제거 (Working Directory는 그대로 유지)
git status	현재 Staging Area와 Working Directory 상태 확인
git diff	Working Directory ↔ Staging Area 차이 확인
git diff --cached	Staging Area ↔ Repository (HEAD) 차이 확인

정리

구성 요소	설명
Working Directory	실제 로컬 파일 수정 상태
Staging Area (Index)	커밋할 내용(변경 사항)을 미리 준비해두는 공간
Repository (HEAD)	이미 커밋되어 버전 관리 중인 내용

장점

- ✅ 어떤 변경을 커밋할지 세밀하게 제어 가능
- ✅ 커밋 단위를 논리적으로 구성 가능
- ✅ 변경 검토 + 테스트 후 커밋 가능

결론

- Git은 단순히 "파일 → 커밋" 구조가 아님 → **Staging Area(중간 준비 단계)가 핵심 역할**
- add → Staging Area → commit → Repository 흐름을 정확히 이해해야 **Git을 효과적으로 쓸 수 있음**
- Staging Area의 존재 덕분에 **Git 커밋 단위가 매우 유연하고 강력함**