- `init`: 필요한 작업 디렉토리를 생성하고 초기화할 때
- `branch`: 브랜치 목록을 보거나 생성하거나 지울 때
- `add`: 디렉토리의 변경된 파일들을 스테이지(index)에 추가할 때
- `status`: 작업 트리의 상태를 출력할 때
- `commit`: 스테이지 파일들을 커밋으로 기록할 때
- `log`: 커밋 로그를 출력할 때
- `switch`: 스테이지 파일들을 커밋으로 기록할 때

구현 우선순위

1. init
2. branch
3. add
4. commit
5. log
6. status
7. switch

구현 모듈

init

```
```text
.git/
  ├── objects/
  │     ├── <blob>들 (파일 내용)
  │     ├── <tree>들 (디렉토리 구조)
  │     └── <commit> (스냅샷의 포인터)
  ├── refs/heads/<branch이름> → "<현재 포인팅하고있는 커밋 해시>"
  └── HEAD → "refs/heads/master" (현재 보고있는 branch 이름)
```
```

git init하게 되면 내부 동작
1. 작업 폴더에 .git 디렉토리 생성 하고 하위 폴더에 아래와 같은 폴더들을 저장한다
`.git/objects/` - 오브젝트들이 저장될 폴더
`.git/refs/heads/` - 브랜치들이 저장될 폴더
`.git/HEAD` - 현재 내가 바라보는 브랜치의 이름만 저장하는 파일
다음 내용 저장 ${ref: refs/heads/main}

2. 기본 브랜치인 .git/refs/heads/ 내부에 main 브랜치 만들기
내용은 해당 브랜치가 가리키고 있는 커밋 객체 해시값
3.

요약
1. 디렉토리 생성
2. main 브랜치

branch

구현할 기능

1. branch 파일을 생성한다.
   - 해당 파일내에 커밋 SHA 번호를 저장한다.
2. 브랜치 목록 출력


```
.git/
  ├── refs/
  │     └── heads/
  │           ├── master  → abc123...
  │           └── feature → abc123...  ← 새로 만든 브랜치
  └── HEAD → refs/heads/master
```

add

add 기능 구현

1. 해당 파일을 읽는다
2. 내용 -> SHA 해시 값을 추출해서 파일 이름으로 지정
3. 압축하기 - 내용은 `blob ${해당파일 내용의 바이트수}\0${실제 파일내용}` 형식으로 압축한다 (ex. blob 12\0hello world\n)
4. 해당 파일을 /git/objects/<ID 앞2개>/ID 로 생성
5. .git/index에 맵핑 정보를 저장(원래 파일명 - 해시ID)

```
blob 12\0hello world\n

blob <size>\0<내용>
```

index 파일 내용

```
<파일모드>  <해시값>     <충돌여부>  <파일이름>
100644     b6d81b36...  0          src/app.js
100644     e69de29b...  0          README.md
```

위 파일에는 스테이징 정보가 저장되어있다 . 따라서 commit 하면서 파일을 읽을 때 index 파일도 같이 지워야함

깃 내부 데이터 상태 정리
modified - 내 작업 디렉토리에서 파일을 수정한 상태를 의미
staged - 스테이징 영역에 추가되어 변경된 파일을 커밋할 준비가 된 상태
tracking - 이전 커밋이나 add로 Git에 등록하여 Git이 관리하고 있는 파일

'git add'된 파일들이 잠시 머무는 임시구역(=다음 커밋 대기열) 스테이징 영역

.git/index

1. `hello.txt` 파일의 내용을 읽는다.
2. 파일 내용을 압축해서 `blob`으로 저장한다.
   - 파일 내용을 `SHA-1 해시`로 계산하여 ID를 생성한다.
   - 이를 `blob`타입으로 `.git/objects/` 디렉토리에 저장한다.
3. `.git/index` 파일을 갱신한다. (스테이징 처리)
   - blob 객체의 해시와 이 해시가 어떤 파일과 맵핑되는지를 `.git/index`에 저장한다.
   - `.git/index` 에 저장된 객체상태 == 스테이징 상태

commit

Tree 객체 구현

(하위 디렉토리부터 먼저 생성한다)
1. index의 파일모드를 읽고 디렉토리만 찾아낸다.
2. 해당 디렉토리들만 Tree 객체로 만든다
3. 폴더 구조와 그 안의 파일과 하위 폴더의 연결 정보를 저장 한다 .
   파일의 경우 blob ID를 바로 저장하고, 디렉토리의 경우 디렉토리 ID를 저장
   -> 파일 이름을 통해서 하위 정보를 파악하여 저장
4. 내용으로 해시값을 생성 한다. 이를 파일 이름으로 지정한다.
3. zlib 압축 후 .git/objects에 저장한다

commit 객체 구현

1. 다음 정보를 저장한다.
   - 갱신을 완료 한 이후에 root Tree의 해시 ID
   - 이전 커밋 객체 해시 ID
   - 기본 정보 (author, committer)
   - 커밋 메시지
2. zlib으로 압축한 이후 .git/objects에 저장한다.
3. git/refs/heads/브랜치명/ 여기에 커밋 해쉬값 갱신

```
commit 1(현재 커밋 포인트)
└─> tree T1 (root)
      ├── README.md → blob1
      └── src → tree T2
            ├── app.js → blob2
            └── util → tree T3
                  └── logger.js → blob3
```

Tree 객체는 어떻게 표현할까?

각 디렉토리마다 객체로 만들며,
(권한, 타입, 해시, 이름 형식으로 저장한다)
(ex. 100644 blob A README.md
100644 blob B hello.txt
040000 tree I src)

객 체 파일명은 `타입+길이+널문자+내용`을 기준 값을 삼아 해쉬 하고 해당 해시명을 파일명 그대로 .git/objects에 저장

main.js 파일이라고하면
(ex. 100644 main.js\0{main.js의Blob해싱값} 형태로 해싱)

커밋 객체 내용

tree <트리 객체 해시 ID>
parent <커밋 객체 해시 ID>
author '이름' <you@example.com> <timestamp>
committer '이름' <you@example.com> <timestamp>

(커밋 메시지)

트리 객체 내용

// 포맷 : <mode> <filename>\0 <SHA-1 해시값>

100644 app.js\0 b2c3d4...
100644 util.js\0 c3d4e5...

스테이징 변경사항을 반영하여 객체 수정하기

1. 변경한 파일

1. 어떤 파일을 변경했는지 어떻게 추적?
-> index에 저장되어있는 맵핑 정보 사용