Git/Linux

- Git 로컬 작업 폴더 원격 저장소에 연결하기 (HTTPS 방식 기준)
- GitHub README 마크다운 문법 정리
- nano 에디터
- 왜 C6이 아니라 C7이 생성됐을까 ?
- [docker] 파일 찾기 sduo find, public IPv4확인은 ns lookup,
- git pull --allow-unrelated-histories
- kill, pkill, killall
- /etc/aliases, /etc/mail/virtusertable
- Git 브랜치 병합 전략 비교 : merge, rebase, cherry-pick
- Git 내부 구조 (Basic)
- commit 생성시 .git 내부 변화
- Git 명령어에 따른 .git 내부 구조 변화 (1부)
- Git 명령어에 따른 .git 내부 구조 변화 (2부)
- 도식화
- 마인드맵
- 환경설정파일 주석

Git ?? ?? ?? ?? ???? ???? (HTTPS ?? ??)

---

1. ?? Git ??? ??? ? ?? ??

1.1 Git ???

```
git init
```

- 현재 폴더를 Git 저장소로 초기화 (`.git` 폴더 생성)

1.2 ?? ??? ??

```
git remote add origin https://github.com/사용자명/저장소명.git
```

- GitHub에서 생성한 저장소 주소를 `origin`이라는 이름으로 등록

---

2. GitHub Personal Access Token(PAT) ??

> GitHub는 2021년부터 비밀번호 인증을 중단하고 토큰**(PAT)** 인증 방식만 허용함

2.1 ?? ?? ??

1. GitHub 로그인
2. 우측 상단 → `Settings > Developer settings > Personal access tokens`
3. `Tokens (classic)` → `Generate new token`

4. `repo`, `workflow` 등의 권한 선택 후 생성

2.2 ?? ?? ??

?? 1: Git ?? ? ?? ??

```
git push origin main
```

- GitHub 사용자명 입력 후
  비밀번호 자리에 토큰 붙여넣기

?? 2: ?? ??

```
git config --global credential.helper store
```

- 한 번 입력하면 이후 저장됨 (로컬 평문 저장 주의)

---

3. Git ?? ??? ??

3.1 ?? ?? ??? ?? ???? (??)

```
git pull origin main --allow-unrelated-histories
```

- 로컬과 원격 저장소에 공통 커밋이 없을 때 충돌 방지용
- 초기 1회만 사용

3.2 ?? ?? ????

```
git add .
```

- 현재 폴더의 모든 변경 파일을 스테이징 영역에 추가

3.3 ??

```
git commit -m "커밋 메시지"
```

- 변경 내용을 로컬 저장소에 기록

3.4 ?? ???? ??

```
git push -u origin main
```

- `-u` 옵션은 이후 `git push`만으로도 기본 원격 브랜치 지정됨

?? ???

```
git init
git remote add origin https://github.com/username/repo.git
git pull origin main --allow-unrelated-histories
git add .
git commit -m "초기 커밋"
git push -u origin main
```

GitHub README ???? ?? ??

> GitHub README 마크다운 문법 정리

- .md 확장자는 마크다운 문서
- GitHub는 README.md를 자동으로 프로젝트 홈에서 보여준다 .
- 코드 블럭에 언어명을 지정하면 문법 강조가 적용된다 . (예 : javascript, python, bash)

? 1. ?? (Headings)

```
# H1 제목
## H2 제목
### H3 제목
#### H4 제목
##### H5 제목
###### H6 제목
```

?? 2. ?? (Emphasis)

```
*기울임*  또는  _기울임_
**굵게**  또는  __굵게__
***굵고 기울임***  또는  ___굵고 기울임___
```

? 3. ?? (List)

● 순서 없는 목록

```
- 항목 1
- 항목 2
  - 하위 항목
```

● 순서 있는 목록

```
1. 첫 번째 항목
2. 두 번째 항목
    1. 하위 항목
```

? 4. ?? (Code)

● 인라인 코드

```
`코드`
예: `console.log("Hello")`
```

● 코드 블럭

<pre> <code> ```javascript function hello() { console.log("Hello, world!"); } ``` </code> </pre>

? 5. ?? & ???

- 링크

```
[링크텍스트](https://example.com)
```

- 이미지

```
![설명](이미지URL)
```

- 예

```
![로고](https://github.githubassets.com/images/modules/logos_page/GitHub-Mark.png)
```

? 6. ??? (Blockquote)

```
> 인용문입니다.
>> 중첩 인용문도 가능합니다.
```

? 7. ??? (???)

```
---
***
___
```

? 8. ???? (To-do List)

```
- [ ] 할 일
- [x] 완료된 일
```

? 9. ?? ?? (Details Toggle)

```
<details>
<summary>여기를 클릭해서 열기</summary>

숨겨진 내용을 여기에 작성합니다.

</details>
```

nano ???

```
nano docker-compose.yml
```

내용 수정 후

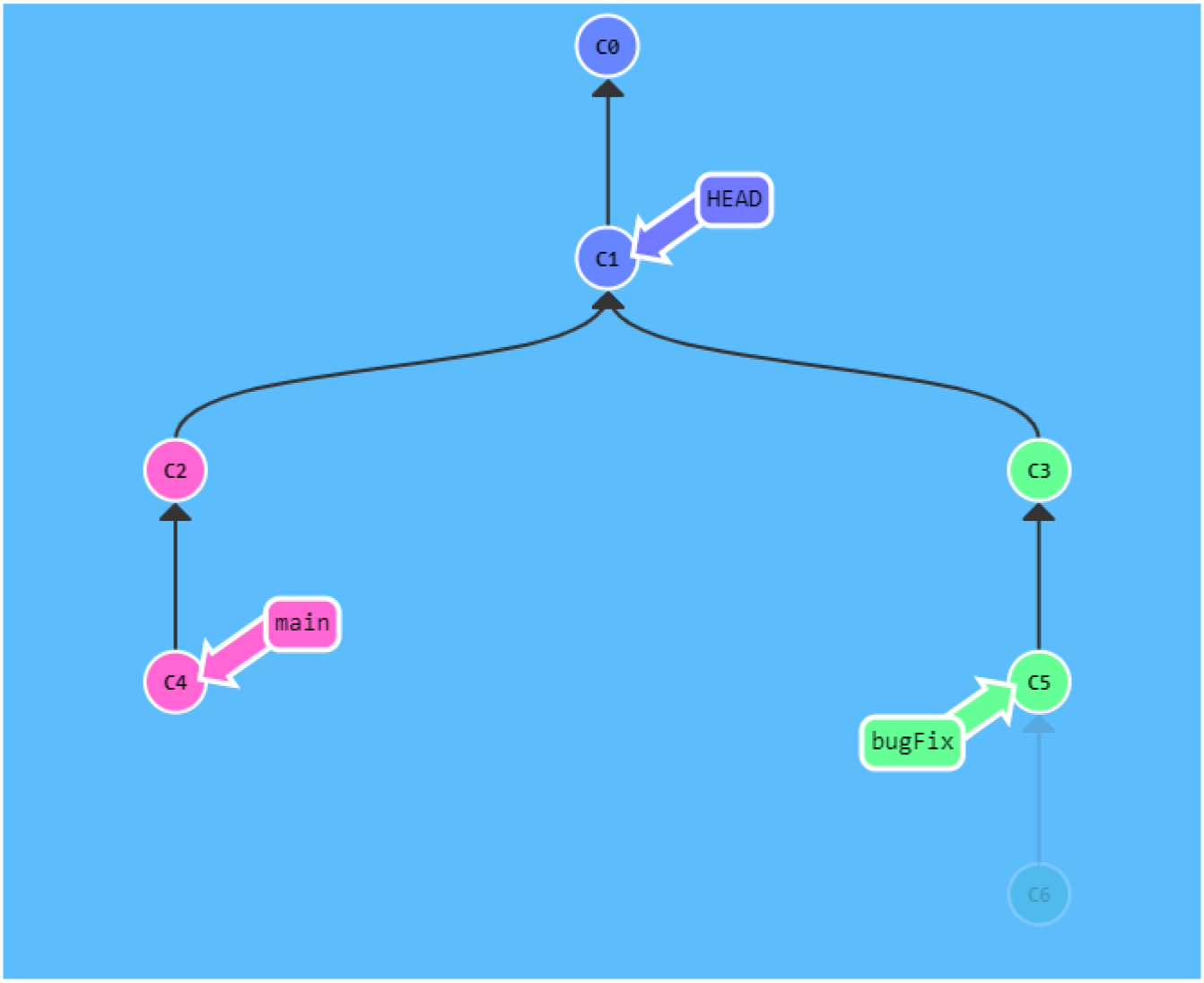
- Ctrl + O 저장
- Enter
- Ctrl + X로 나가기
- Ctrl + K 여러 번 눌러서 한 줄씩 삭제
- 마우스 or Shift + ↓ 로 전체 선택 후 삭제
- 방향키 ↓↓ 끝까지
- Ctrl + K로 한 번에 삭제

? C6? ??? C7? ??????

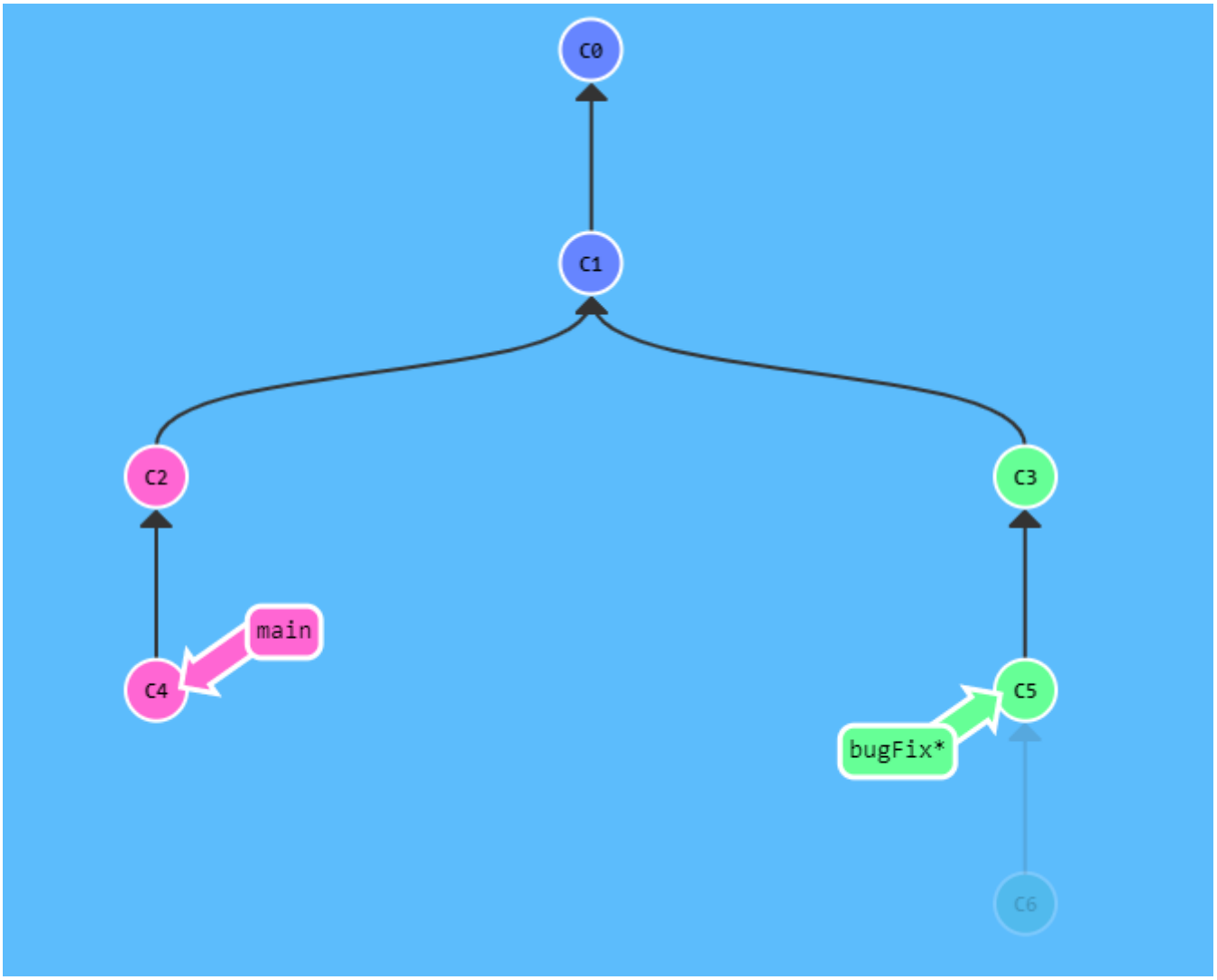
- Git 시각화 툴에서 `bugFix` 브랜치에서 `git commit`을 실행했더니 , 새로운 커밋이 `C6`이 아니라 `C7`로 만들어짐 .
- `C6`은 존재하지만 흐릿하게 표시되고 , 어떤 브랜치에서도 이어지지 않음 .

```
$ git checkout HEAD^
$ git checkout bugFix
$ git commit
```

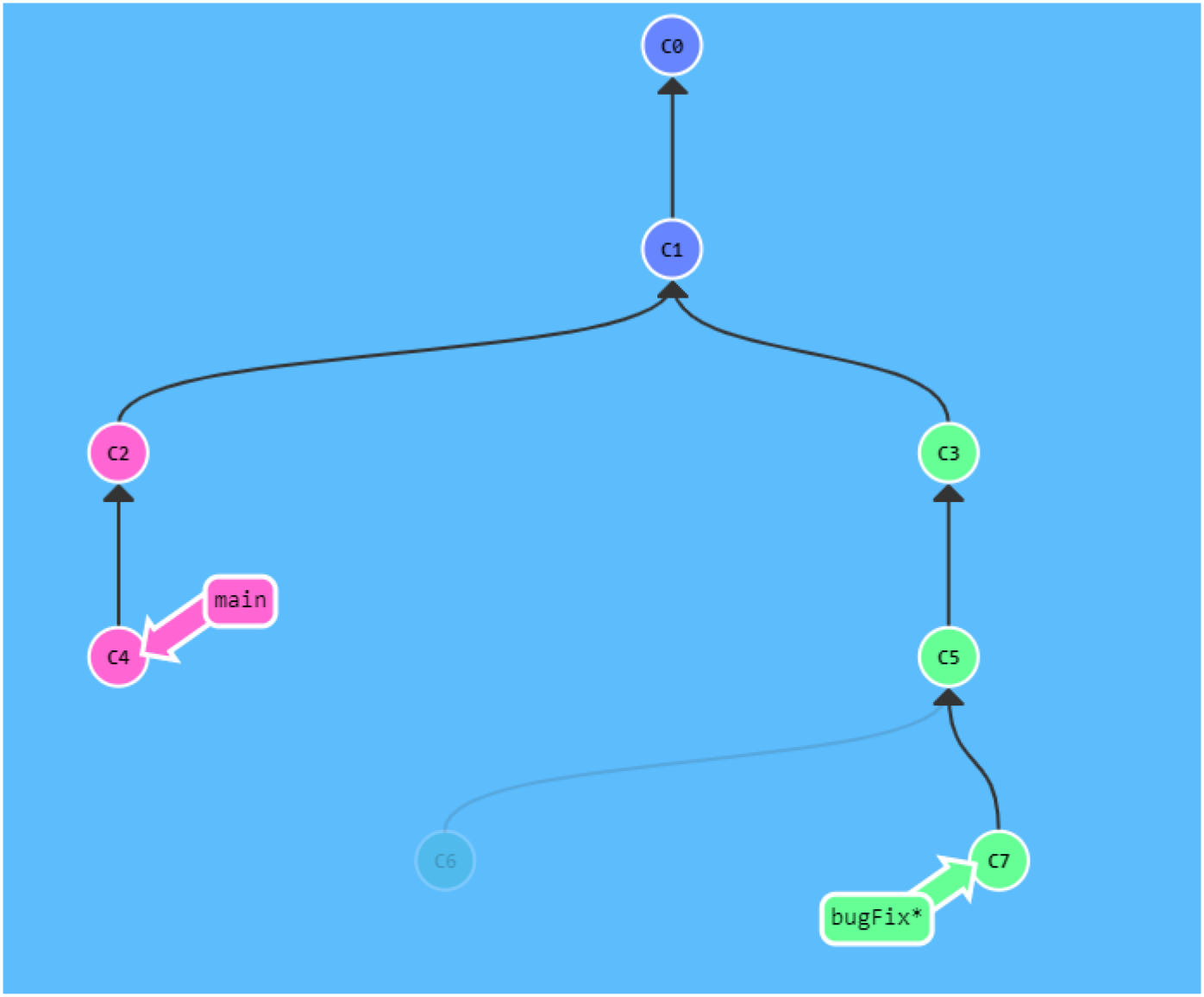
git checkout HEAD^ 수행 직후



git checkout bugFix □□ □□



git commit □□ □□



?? ?? ??

- **C5** 이후에 커밋이 생성됨 .
- C6은 있지만 밝은 색으로 표시 되어 있고 , 커밋 라인이 연결되지 않음 .
- 현재 **HEAD**는 `bugFix` 브랜치에 있고, 별표(*)로 표시되어 있음.
- 새 커밋은 **C7**으로 기록됨 .

? ? C6? ??? C7???

1. **HEAD**가 어디에 있는지가 중요

- Git에서 커밋을 하면 항상 `HEAD`가 가리키는 브랜치 기준으로 커밋이 생긴다 .

```
git commit
```

- 이 명령은 현재 `HEAD`가 가리키는 브랜치 위에 커밋을 생성한다 .
- 현재 `HEAD → bugFix`였기 때문에 , 커밋은 **C5 → C7**으로 이어졌음 .

2. C6? `HEAD`? ???? ??? ?? ??

- C6는 `git checkout`이나 `rebase` 도중 임시로 생겼다가 현재 브랜치 기준에서는 더 이상 이어지지 않은 과거 커밋 이다 .
- 브랜치나 다른 커밋에서 참조하지 않는 커밋은 **garbage collection** 대상 이 된다 .
- 즉 , C6은 예전에 생성됐지만 예전에 생성됐지만 현재 브랜치나 HEAD가 가리키지 않는 고아 커밋이다 .
- C7이 bugFix 브랜치에서 새롭게 커밋한 결과물이다 .

??? ?? ??

```
C4 (main)
 ↑
C2 ←──────┐
          │
C1        │
 ↑        ↓
C0       C3 ← C5 ← C7 (HEAD → bugFix)
            ↖
             C6 (고아 커밋)
```

?? ???

```
# 현재 브랜치 및 HEAD 확인
git branch
git status

# 모든 브랜치와 커밋 로그 확인
git log --graph --oneline --all
```

[docker] ?? ?? sduo find, public IPv4??? ns lookup,

docker-compose.yml 파일 있는 곳 찾기

```
sudo find / -name docker-compose.yml 2>/dev/null
```

EC2 퍼블릭 IP 확인하는 명령어

```
nslookup dainwiki.com
```

git pull --allow-unrelated-histories

1. git pull --allow-unrelated-histories ??

`--allow-unrelated-histories`는 Git에서 서로 연관되지 않은 두 히스토리를 강제로 병합할 때 사용하는 옵션이다 .

보통 서로 다른 프로젝트를 하나로 합치거나 , 원격 저장소와 로컬 저장소가 서로 공통 조상을 갖지 않을 때 사용한다 .

2. ?? ?? ??

```
fatal: refusing to merge unrelated histories
```

? ?? ?????

- 로컬에서 `git init`으로 새 저장소 생성 후 커밋
- 원격 저장소 (GitHub)에 다른 초기 커밋이 존재 (예 : README.md)
- 이후 `git pull origin main` 시 위 오류 발생

3. ?? ??

```
git pull origin main --allow-unrelated-histories
```

- 두 히스토리 병합 강제 수행
- 이후 충돌 (conflict)이 발생할 수 있으니 수동으로 해결 필요

4. ?? ?

```
# 로컬 초기화
git init
git remote add origin https://github.com/user/repo.git

# 강제 병합
git pull origin main --allow-unrelated-histories
```

---

5. ????

- 한 번만 사용하면 충분 : 병합이 끝나고 연결되면 , 이후부터는 일반 `git pull`로도 작동합니다 .
- 병합 충돌 발생 가능성이 크므로 사용 후 커밋 기록 확인 필수

---

6. ?? ??

옵션	설명
`--rebase`	병합 대신 리베이스로 히스토리 깔끔하게 유지
`--no-commit`	병합하되 커밋은 직접 하고 싶을 때
`--strategy=ours` / `theirs`	충돌 시 우선할 히스토리 선택

---

kill, pkill, killall

kill, pkill, 그리고 killall은 유닉스와 유닉스 계열 시스템 (예: Linux)에서 프로세스를 종료하기 위한 명령어들이다.

이 명령어들은 각각 다르게 작동하며, 다음과 같은 특징이 있다.

kill

- kill 명령어는 주어진 프로세스 ID (PID)를 사용하여 특정 프로세스를 종료한다.
- 기본적으로 kill 명령어는 TERM 시그널 (시그널 15)을 보낸다.
- 다른 시그널도 보낼 수 있는데, 예를 들어, kill -9 [PID] 명령어는 KILL 시그널 (시그널 9)을 보내 프로세스를 강제로 종료한다.

```
kill 12345
kill -9 12345
```

pkill

- pkill 명령어는 프로세스 이름을 기준으로 프로세스들을 찾아 종료한다.
- 프로세스 이름 또는 다른 특성을 사용하여 여러 프로세스를 한번에 종료할 수 있다.

```
pkill firefox
pkill -9 firefox
```

killall

- killall 명령어도 프로세스 이름을 기준으로 프로세스를 찾아 종료하지만, 주로 같은 이름을 가진 모든 프로세스를 종료하는데 사용된다.

```
killall firefox
killall -9 firefox
```

??

1. kill : pid
2. pkill : name
3. killall : name, pid

/etc/aliases, /etc/mail/virtusertable

모두 메일 서버 설정 파일로 , 메일 주소 또는 계정으로부터 메일을 다른 메일 주소 또는 계정으로 전달하기 위한 규칙을 정의한다 .

?? ??

- /etc/aliases: 주로 로컬 시스템 계정에 대한 메일 전달을 처리하는데 사용된다 .
- /etc/mail/virtusertable: 여러 도메인을 호스팅하는 환경에서 메일 전달을 처리

?? ???

- /etc/aliases: 주로 로컬 도메인 또는 특정 계정에 대한 메일 전달을 설정
- /etc/mail/virtusertable: 특정 도메인으로부터의 메일 전달을 설정할 수 있으며 , 도메인 간 메일 전달도 설정 가능

?? ?? ? ??

- /etc/aliases 파일과 관련 명령이 있는 경우 (newaliases, sendmail -bi)
- /etc/mail/virtusertable 파일과 관련 명령이 있는 경우 (makemap hash)

/etc/aliases

- 시스템에 존재하는 실제 사용자 계정 이름을 기준으로 메일을 전달
- 예 : root로 온 메일을 admin에게 전달
- name: account1, account2,...: 특정 이름 (또는 로컬 계정)으로 수신된 메일을 목록에 나열된 계정으로 전달
- name::include:filename: 파일에 명시된 계정 목록으로 메일을 전달
- 적용 명령 : 설정을 적용하기 위해 newaliases 또는 sendmail -bi를 실행할 수 있다 .

```
webmaster: user1, user2, user3, externaluser@example.com
admin::include:/etc/mail/admingroup
```

/etc/mail/virtusertable

- 도메인 기반 이메일 주소를 실제 사용자 계정이나 외부 주소로 매핑
- 예 : info@example.com → realuser
- 한 메일 서버에서 여러 도메인을 호스팅하고 있을 때 특정 도메인으로의 메일을 다른 계정 또는 도메인으로 전달하는 데 사용됨

- user@domain1.com user2: 도메인 1에 있는 사용자로부터 오는 메일을 도메인 2의 다른 사용자에게 전달
- @domain.com user@otherdomain.com: 도메인의 모든 메일을 다른 도메인의 특정 사용자에게 전달
- 설정을 적용하기 위해 makemap hash /etc/mail/virtusertable < /etc/mail/virtusertable를 실행

```
user1@domain1.com user2

user1@domain2.com user3

@domain3.com user4@domain4.com
```

Git ??? ?? ?? ??: merge, rebase, cherry-pick

1. ?? ?? ??

명령어	설명
`merge`	브랜치 간 변경사항을 병합, 병합 커밋 생성
`rebase`	커밋 기반을 다른 브랜치 위로 옮겨 이력 재정렬
`cherry-pick`	특정 커밋만 골라 현재 브랜치에 복사

2. ???? ?? ??

`git merge <브랜치명>`

- 역할 : 다른 브랜치의 변경 내용을 현재 브랜치에 병합
- 병합 커밋 을 만들어 두 브랜치의 내용을 하나로 합침

```
git checkout main         # 병합 대상 브랜치
git merge feature/login   # 병합할 소스 브랜치
```

- □□ 즉 : `main ← feature/login` 병합됨

`git rebase <브랜치명>`

- 역할 : 현재 브랜치 커밋을 `<브랜치명>` 위로 옮김
- 커밋이 새로 복사되어 **SHA** 변경
- 병합 커밋 없이 깔끔한 직선형 이력 생성

```
git checkout feature/login
git rebase main
```

- □□ 즉 : `feature/login`을 `main` 기준으로 다시 쌓음

`git cherry-pick <커밋해시>`

- 역할 : 특정 커밋 하나를 현재 브랜치로 가져옴
- 전체 브랜치를 병합하지 않고 선택적 반영

```
git checkout main
git cherry-pick 7a1f3b2
```

---

3. ?? ?? ??

`merge`

```
A---B---C---M       (main)
     \     /
      D---E         (feature)
```

`rebase`

```
A---B---C---D'---E' (feature rebased onto main)
```

`cherry-pick`

```
main:    A---B---C---E'
feature:           D---E
```

---

4. ??? ??

상황	추천 명령어
팀 프로젝트에서 병합 이력 유지	`merge`
개인 작업 커밋 정리	`rebase`

특정 버그 수정만 따로 반영	`cherry-pick`

5. ????

- `merge`: 병합 커밋으로 이력이 복잡해질 수 있음
- `rebase`: 협업 중인 브랜치에 사용 금지 (이력 충돌)
- `cherry-pick`: 동일 커밋이 중복될 수 있음

6. ?? ???: `revert`, `reset`, `stash`

명령어	역할	주 사용 목적
`revert`	특정 커밋을 취소하는 새 커밋 생성	실수한 커밋을 안전하게 되돌림
`reset`	커밋 이력 /스테이징 /작업 폴더 상태를 되돌림	브랜치나 파일을 과거 상태로 복구
`stash`	현재 작업을 임시로 저장	급히 브랜치를 바꿔야 할 때

```
# revert: 안전한 취소
git revert <커밋 해시>

# reset: 이력 되돌리기
git reset --soft HEAD~1   # 커밋만 되돌림
git reset --hard HEAD~1   # 작업 내용까지 삭제

# stash: 현재 작업 저장
git stash
git stash pop
```

Git ?? ?? (Basic)

1. Git? ?? ??? ?? 스냅샷 시스템

Git은 "변경점 (diff)"이 아니라 , 전체 프로젝트의 스냅샷(상태)을 저장 한다 .

> 커밋 하나 = 전체 폴더의 압축 스냅샷 (단 , 중복은 자동 제거됨)

2. Git? 3? ??

```
작업 디렉토리          →      스테이징 영역          →      로컬 저장소
(Working Directory)      (Staging/Index)          (Repository/.git)
    코드 작성                  git add                  git commit
```

영역	설명
작업 디렉토리	실제로 코드 작성 /수정하는 공간
스테이징 영역 (Index)	커밋할 파일을 담아두는 임시 공간 (`git add`)
로컬 저장소	커밋이 저장되는 `.git` 디렉토리

3. ?? Git ?? ??

Git은 4가지 핵심 객체로 이루어져 있습니다 :

객체	설명	예시
Blob	파일 내용 저장 (binary large object)	`hello.java` 내용
Tree	디렉토리 구조와 파일 리스트	폴더 구조
Commit	커밋 메타데이터 + 부모 커밋 + tree	"커밋 메시지 "
Tag	특정 커밋에 이름 부여	v1.0.0

커밋은 tree를 가리키고 , tree는 blob들을 가리킵니다 .

```
[commit]
   |
   v
[tree]
 ├── [blob] hello.txt
 └── [blob] app.js
```

---

4. HEAD??

```
HEAD → 현재 체크아웃한 브랜치 or 커밋
```

- 보통은 `HEAD → main`
- `git checkout` 하면 HEAD가 다른 커밋을 가리킴
- `git reset`은 **HEAD**의 위치를 옮기는 것

---

5. ?????

브랜치는 단지 커밋을 가리키는 포인터 이다 .

```
A---B---C  ← main (브랜치)
        ↑
      HEAD
```

- 브랜치를 옮긴다 → 포인터 이동
- 커밋한다 → 현재 브랜치가 가리키는 곳에서 새로운 커밋 추가

---

6. Reflog??

```
git reflog
```

→ HEAD와 브랜치가 가리키던 이전 위치들의 로그

- `reset`이나 `checkout` 후에도 추적 가능
- 내부적으로 Git은 HEAD의 이동을 계속 추적함

7. .git ???? ?? ??

```
.git/
├── objects/        ← 커밋, 트리, 블롭, 태그 객체 저장소
├── refs/           ← 브랜치, 태그 등의 포인터
├── HEAD            ← 현재 가리키는 브랜치
├── index           ← 스테이징 정보
├── logs/           ← reflog 정보 저장
└── config          ← Git 설정
```

? ??

개념	설명
Commit	전체 스냅샷 + 부모 커밋 정보
Blob	파일 내용 저장
Tree	폴더 구조
Branch	커밋을 가리키는 포인터
HEAD	현재 브랜치 or 커밋
Staging Area	다음 커밋을 준비하는 공간
Reflog	HEAD의 이동 이력 추적 가능 (복구에 유용)

commit ??? .git ?? ??

????

1. `hello.txt` 파일 생성
2. `git add hello.txt`
3. `git commit -m "Add hello"`

---

1. ?? ???? (Working Directory)

```
echo "Hello Git" > hello.txt
```

- 파일 생성됨
- 아직 Git은 이 파일을 추적하지 않음

---

2. `git add hello.txt`

```
git add hello.txt
```

?? ??:

구성요소	변화 내용
`.git/index`	스테이징 영역에 파일 등록됨 (경로 , 해시 , 권한 등 저장)
`.git/objects/`	`hello.txt` 의 내용이 `blob` 객체로 저장됨 (SHA-1 해시 기반)

◫◫ 예시 구조 :

```
.git/
└── objects/
```

```
    └── 3a/
        └── 3efbed4aef... ← hello.txt의 blob 객체
```

---

3. `git commit -m "Add hello"`

?? ??:

.git/objects/

객체	내용
`tree` 객체	커밋 당시의 디렉토리 구조 (hello.txt 포함)
`commit` 객체	커밋 메시지 , 트리 포인터 , 작성자 등 메타데이터 포함

```
.git/objects/
├── 3a/...(blob)
├── 47/... (tree)
└── a1/... (commit)
```

.git/refs/heads/main

- 브랜치 포인터가 새 커밋 해시로 갱신됨

```
refs/heads/main → a1b2c3d4e5...
```

.git/HEAD

- 여전히 `ref: refs/heads/main` 상태 → main 브랜치를 따라감

.git/logs/HEAD & logs/refs/heads/main

- 커밋 전후 `HEAD` 포인터 변경 이력 추가됨 (reflog)

```
0000000 → a1b2c3d4 Commit: Add hello
```

?? ?? (?? 1? ?)

```
□□Working Directory        ← hello.txt 생성
    ↓ git add
□□.git/index               ← 스테이징 정보 저장
    ↓ git commit
□□.git/objects/
    ├── blob (파일 내용)
    ├── tree (디렉토리 구조)
    └── commit (메타데이터)

□□.git/refs/heads/main     ← 브랜치 → 커밋 연결
    ↑
□□.git/HEAD                ← 현재 브랜치 추적
□□.git/logs/               ← HEAD/브랜치 이력 추적
```

?? ?? (?? ?)

```
cat .git/HEAD
# ref: refs/heads/main

cat .git/refs/heads/main
# a1b2c3d4e5...

git ls-tree HEAD
# 100644 blob 3a3efbed...□hello.txt

git cat-file -p <커밋해시>
# tree 47fa...
# author ...
# message: Add hello
```

??

단계	설명
`git add`	blob 객체 생성 + index 등록
`git commit`	tree + commit 객체 생성 , 브랜치 포인터 이동

`.git` □□	objects, refs, HEAD, logs □□ □□

Git ???? ?? .git ?? ?? ??(1?)

1. `git merge` ? ?? ??

1.1 ?? ???

```
git checkout main
git merge feature
```

1.2 `.git` ?? ??

구성 요소	변화 내용
`.git/objects/`	병합 결과로 새로운 **merge commit** 객체 생성
`.git/refs/heads/main`	브랜치 포인터가 merge 커밋으로 이동
`.git/HEAD`	여전히 `ref: refs/heads/main` 유지
`.git/MERGE_HEAD`	병합 대상 커밋 (`feature`) 해시 저장 (병합 중일 때 생성됨)
`.git/logs/HEAD`	HEAD의 이동 이력 반영됨 (reflog)

1.3 ?? ??

```
A---B---C ← main
     \
      D---E ← feature

▶ git merge feature

결과:

A---B---C-------M (merge commit) ← main, HEAD
```

```
    \       /
     D---E
```

---

2. `git rebase` ? ?? ??

2.1 ?? ???

```
git checkout feature
git rebase main
```

2.2 `.git` ?? ??

구성 요소	변화 내용
`.git/objects/`	`feature`의 커밋들을 기반 변경 후 새 커밋으로 재작성
`.git/refs/heads/feature`	브랜치가 새 커밋들로 덮어씀
`.git/REBASE_HEAD`	리베이스 기준 커밋을 기록
`.git/HEAD`	여전히 `ref: refs/heads/feature`
`.git/logs/HEAD`	HEAD의 커밋 재작성 이력 반영됨

2.3 ?? ??

```
main:    A---B---C
feature:          D---E

▶ git rebase main

결과:
main:    A---B---C
feature:          D'---E' ← HEAD
```

> 원래 커밋 D, E는 삭제되지 않고 `.git/objects`에 보존됨

---

3. `git reset` ? ?? ??

3.1 ?? ???

```
git reset --hard HEAD~1
```

3.2 `.git` ?? ??

구성 요소	변화 내용
`.git/refs/heads/main`	브랜치 포인터가 이전 커밋으로 이동
`.git/HEAD`	여전히 `ref: refs/heads/main`
`.git/index`	되돌린 커밋 시점으로 스테이징 영역 초기화
작업 디렉토리	`--hard`인 경우 파일도 초기화
`.git/ORIG_HEAD`	이전 HEAD 커밋 해시를 저장
`.git/logs/HEAD`	이동 전후 이력 기록됨

3.3 ?? ??

```
A---B---C ← main, HEAD
        ↑
     ORIG_HEAD

▶ git reset --hard B

결과:

A---B ← main, HEAD
```

---

4. ?? HEAD ?? ??

파일 이름	생성 시점	설명
`.git/MERGE_HEAD`	`git merge` 수행 중	병합 대상 커밋 해시 저장
`.git/REBASE_HEAD`	`git rebase` 수행 중	기준 커밋 해시 저장
`.git/ORIG_HEAD`	`git reset`, `merge` 등	리셋 전 커밋 백업
`.git/FETCH_HEAD`	`git fetch` 이후	원격에서 받아온 커밋들 정보
`.git/CHERRY_PICK_HEAD`	`git cherry-pick` 중 충돌 시	적용 대상 커밋 기록

5. ?? ???

명령어	커밋 이동	브랜치 이동	**HEAD** 변경	디렉토리 영향	특수 파일 생성
`merge`	□	□	□	충돌 시 변동 있음	`MERGE_HEAD`
`rebase`	□ (rewrite)	□	□	충돌 시 변동 있음	`REBASE_HEAD`
`reset --hard`	□ (이전 커밋으로)	□	□	✔ 완전 초기화	`ORIG_HEAD`

Git ???? ?? .git ?? ?? ?? (2?)

4. `git stash` ? ?? ??

4.1 ?? ???

```
git stash
```

4.2 `.git` ?? ??

구성 요소	변화 내용
`.git/objects/`	현재 작업 상태 (스냅샷)가 `stash`용 커밋으로 저장됨 (2~3개의 커밋 객체)
`.git/logs/refs/stash`	새로운 stash가 추가된 로그 생성
`.git/refs/stash`	가장 최신 stash를 가리키는 포인터 (없으면 새로 생성됨)

```
.git/
├── objects/         ← stash 내용 포함 커밋 객체 추가
├── refs/stash       ← 최신 stash 참조
└── logs/refs/stash ← stash 이력 저장
```

4.3 ?? ??

```
main:    A---B---C
                 ↑
           HEAD / index


▶ git stash


결과:
- 워킹 디렉토리와 인덱스 초기화됨
- .git/refs/stash → stash 커밋 가리킴
```

5. `git cherry-pick` ? ?? ??

5.1 ?? ???

```
git cherry-pick a1b2c3d
```

5.2 `.git` ?? ??

구성 요소	변화 내용
`.git/objects/`	cherry-pick된 커밋 내용 기반으로 새 커밋 객체 생성
`.git/CHERRY_PICK_HEAD`	대상 커밋의 해시 저장 (충돌 발생 시 사용됨)
`.git/HEAD`	현재 브랜치에 새 커밋 추가
`.git/index`	병합 결과 반영

5.3 ?? ??

```
main:    A---B---C ← HEAD
feature:         D---E ← cherry-pick 대상

▶ git cherry-pick E

결과:
main:    A---B---C---E' ← HEAD
```

> E와 동일한 내용이지만 새 커밋 E′가 만들어짐

6. `git revert` ? ?? ??

6.1 ?? ???

```
git revert C
```

6.2 `.git` ?? ??

구성 요소	변화 내용
`.git/objects/`	**revert**용 새 커밋 객체 생성
`.git/index`	C의 반대 연산 내용 반영
`.git/HEAD`	새 커밋으로 이동

> `revert`는 원본 커밋을 삭제하지 않고 , "반대로 되돌리는" 커밋 을 추가한다 .

6.3 ?? ??

```
A---B---C ← HEAD

▶ git revert C

결과:

A---B---C---C' ← HEAD (C를 무효화하는 커밋)
```

---

7. ?? ?? ?? (2?)

파일 이름	생성 명령어	설명
`.git/CHERRY_PICK_HEAD`	`git cherry-pick`	적용 중인 커밋 해시 저장 (충돌 시 사용)
`.git/REVERT_HEAD`	`git revert` (중단 시)	revert 진행 중인 커밋 정보
`.git/refs/stash`	`git stash`	가장 최근 stash 참조
`.git/logs/refs/stash`	`git stash`	stash 추가 이력 저장

---

8. ?? ??? (2?)

명령어	새 커밋 생성	기존 커밋 삭제	**HEAD** 이동	특수 파일	비고
`stash`	O (임시 커밋)	□	□	`refs/stash`	작업 보관
`cherry-pick`	O	□	O	`CHERRY_PICK_HEAD`	선택 커밋 복사
`revert`	O	□	O	`REVERT_HEAD`	기존 커밋 반전

???

---

1. ?? ?? ??

```
.git/
├── HEAD                ← 현재 가리키는 브랜치 (예: ref: refs/heads/main)
├── config              ← 로컬 Git 설정
├── description         ← 주로 bare repo에서 사용
├── index               ← 스테이징 영역 정보 (트래킹 중인 파일 목록)
├── objects/            ← Git 객체 저장소 (blob, tree, commit, tag)
│   ├── xx/xxxx...      ← SHA-1 해시로 저장된 개별 객체
│   └── pack/           ← 압축된 객체 저장소 (.pack, .idx)
├── refs/               ← 브랜치, 태그 등 참조(포인터)
│   ├── heads/          ← 브랜치 참조 (예: main, feature)
│   ├── remotes/        ← 원격 브랜치 정보 (예: origin/main)
│   └── tags/           ← 태그 정보
├── logs/               ← HEAD 및 refs의 이동 이력 (reflog 저장소)
│   ├── HEAD
│   └── refs/
├── info/               ← exclude 설정 등 부가 정보
└── hooks/              ← 커밋, 푸시 등 Git 이벤트용 스크립트
```

---

2. ? ?? ?? ??

? `.git/HEAD`

- 현재 Git이 바라보는 위치
- 대부분 `ref: refs/heads/main` 형태
- **HEAD** → 브랜치 → 커밋

? `.git/objects/`

- Git의 핵심 : 모든 파일 /폴더 /커밋을 **SHA-1** 해시값 으로 저장

- 하위 폴더 + 해시 기반 객체들

```
.git/objects/
├── a7/...
├── d1/...
└── pack/    ← packfile 압축 저장소
```

? `.git/refs/`

- 브랜치 /태그가 어떤 커밋을 가리키는지 저장

```
.git/refs/heads/main → a1b2c3d4e5 (커밋 해시)
```

? `.git/logs/`

- `git reflog` 명령에서 쓰이는 내부 이력
- `HEAD`, `branches`, `remotes` 이동 내역 저장

? `.git/index`

- 스테이징 영역 (index)의 내부 상태를 저장하는 바이너리 파일
- `git add` → index에 등록 → `git commit` 때 커밋됨

? `.git/hooks/`

- 커밋 /푸시 /머지 전후에 실행되는 자동화 스크립트

```
pre-commit, post-commit, pre-push 등
```

---

3. ?? ??

```
□□Working Directory
   ↓ git add
□□.git/index (Staging Area)
   ↓ git commit
□□.git/objects/ (커밋, 트리, 블롭 저장)
   ↓ 브랜치 포인터 이동
```

```
□□.git/refs/heads/main
    ↑ HEAD → 현재 브랜치 참조
```

??

구성요소	설명
`HEAD`	현재 브랜치 또는 커밋을 가리킴
`refs`	브랜치/태그 → 커밋 연결
`objects`	모든 데이터 (SHA-1 기반) 저장소
`index`	스테이징 상태 추적
`logs`	HEAD 및 브랜치 이동 이력 (reflog)
`hooks`	Git 이벤트 발생 시 실행되는 스크립트

????

```
Git 내부 구조
├── 1. Working Directory
│   └── 실제 작업 공간 (코드 작성, 수정)
│
├── 2. Index (Staging Area) ← .git/index
│   └── git add로 관리됨
│
├── 3. Local Repository (.git/)
│   ├── 3.1 objects/
│   │   ├── blob: 파일 내용
│   │   ├── tree: 디렉토리 구조
│   │   └── commit: 스냅샷 + 부모 참조 + 메타
│   │
│   ├── 3.2 refs/
│   │   ├── heads/: 브랜치 포인터
│   │   ├── remotes/: 원격 브랜치
│   │   └── tags/: 태그 포인터
│   │
│   ├── 3.3 HEAD
│   │   ├── 현재 브랜치 참조 (예: ref: refs/heads/main)
│   │   └── detached 상태 가능
│   │
│   ├── 3.4 logs/
│   │   └── HEAD 및 refs의 이동 이력 (reflog)
│   │
│   ├── 3.5 config
│   │   └── 사용자 설정 정보
│   │
│   ├── 3.6 hooks/
│   │   └── pre-commit, post-commit 등 자동화 스크립트
│   │
│   └── 3.7 특수 HEAD 파일
│       ├── MERGE_HEAD        ← merge 중 생성
│       ├── REBASE_HEAD       ← rebase 중 생성
│       ├── ORIG_HEAD         ← reset 전 위치 백업
```

```
│       ├── CHERRY_PICK_HEAD  ← cherry-pick 중 충돌 시
│       ├── REVERT_HEAD       ← revert 충돌 시
│       └── FETCH_HEAD        ← fetch 결과
│
├── 4. 작업자 변경 사항
│   ├── git add
│   │   └── 변경 → index 반영
│   ├── git commit
│   │   ├── index → commit 객체 생성
│   │   └── 현재 브랜치 → 새 커밋 가리킴
│   ├── git reset
│   │   ├── HEAD/브랜치 포인터 이동
│   │   └── --hard 시 index + 작업 디렉토리 초기화
│   ├── git merge
│   │   ├── merge commit 생성
│   │   └── MERGE_HEAD 참조
│   ├── git rebase
│   │   ├── 커밋 재작성 (rewrite)
│   │   └── REBASE_HEAD 참조
│   ├── git cherry-pick
│   │   ├── 커밋 복사
│   │   └── CHERRY_PICK_HEAD 참조
│   ├── git revert
│   │   ├── 커밋 변경 취소
│   │   └── REVERT_HEAD 참조
│   └── git stash
│       ├── 작업 디렉토리 변경 임시저장
│       ├── refs/stash 참조
│       └── logs/refs/stash 로그 기록
│
└── 5. 조회 및 복구
    ├── git reflog  → HEAD, 브랜치 이동 기록 확인
    ├── git log     → 커밋 히스토리
    ├── git cat-file -p <해시> → 객체 내용 확인
    └── git fsck    → 손상된 객체 검사
```

??

키워드	의미
blob	파일 내용
tree	폴더 구조
commit	전체 스냅샷
HEAD	현재 위치 추적 포인터
refs	브랜치/태그가 가리키는 대상
logs	HEAD/브랜치 이동 기록 (reflog)
index	커밋 준비 상태 저장
stash	작업 보관소 (임시 커밋)

?????? ??

설정파일 (.env, .ini, conf, 등)에서 주석을 쓸 때 #, ; 씀

- .env → 되도록 #만 쓰자 , ;도 되는 경우 있음
- .conf, Dockerfile, sh → 무조건 #
- .ini → 되도록 ; 쓰자 , #도 되는 경우 있음