5. Rebase와 히스토리 조작

`git rebase`란 무엇인가?

✅ 1. 개념 정의

> `git rebase`는 **특정 브랜치의 커밋들을 다른 브랜치의 최신 커밋 이후에 다시 적용**하는 것.
> 즉, **브랜치의 시작점을 새롭게 바꾸는 것**이다.

```
기존 히스토리
A---B---C (main)
     \
      D---E (feature)

rebase 실행 후
A---B---C---D'---E' (feature)
```

→ `D`, `E`를 **C 이후에 다시 "복사"해서 새 커밋 D', E'로 붙이는 것**

✅ 2. 기본 사용법

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

> "feature 브랜치를 main 위에 다시 쌓아라"라는 의미

✅ 3. `merge` vs `rebase` 비교

항목	`git merge`	`git rebase`
히스토리	분기점과 병합 커밋 남음	선형 구조
커밋 수	병합 커밋 1개 추가	동일한 커밋 수 유지
사용 용도	협업, 충돌 추적에 유리	깔끔한 히스토리 유지
커밋 해시	그대로 유지	모두 바뀜 (새로 생성됨)
충돌 처리	병합 시	재적용 시 각각 발생

✅ 4. 실전 예시

```
# 브랜치 준비
git checkout main
echo "main" > a.txt
git add a.txt
```

```
git commit -m "main commit"

git checkout -b feature
echo "f1" > f.txt
git commit -am "feat: f1"
echo "f2" >> f.txt
git commit -am "feat: f2"

# main 브랜치에서 변경 발생
git checkout main
echo "hotfix" >> a.txt
git commit -am "fix: hotfix"

# feature 브랜치를 최신 main 위로 재배치
git checkout feature
git rebase main
```

→ `feature`의 커밋 2개가 **main의 최신 커밋 이후로 다시 쌓임**

✅ 5. 충돌 발생 시 대처

```
git rebase main
# 충돌 발생
# 파일 수정 후
git add <파일>
git rebase --continue
```

중단하고 되돌리기:

```
git rebase --abort
```

✅ 6. 주의사항: 공유된 브랜치에서는 rebase 금지!

> ❌ **이미 push한 브랜치를 rebase한 후 다시 push하면 충돌 위험이 매우 큼!**

이유:

- `rebase`는 **커밋 해시를 변경**함
- 다른 사람이 같은 브랜치를 pull 했을 경우, 서로 다른 히스토리가 생김

안전한 협업 전략

- `main`, `develop` 등 공유 브랜치에서는 `merge` 사용
- 개인 기능 브랜치에서만 `rebase`로 히스토리 정리

✅ 7. 인터랙티브 리베이스 (`-i`)

```
git rebase -i HEAD~3
```

> 최근 3개 커밋을 편집(수정/병합/삭제)할 수 있음

가능한 작업:

- `pick`: 커밋 유지
- `reword`: 메시지 수정
- `edit`: 커밋 내용 수정
- `squash`: 이전 커밋과 합침
- `drop`: 삭제

✅ 8. 마무리 요약

명령어	설명
`git rebase <branch>`	현재 브랜치를 `<branch>` 위에 재배치
`--continue`	충돌 해결 후 진행
`--abort`	리베이스 중단 및 복원
`-i`	커밋 목록 편집 (squash, reword 등)

🔐 결론: 언제 `rebase`를 쓰는가?

- **혼자 작업한 브랜치 정리**할 때
- **Pull 전에 선형 정리용으로** `git pull --rebase`
- **히스토리를 깔끔하게 정리하고 싶을 때**

> 단, **다른 사람과 공유된 커밋에는 절대 사용 금지!**

- ## vs `git merge`

✅ 1. 기본 개념 비교

항목	`git merge`	`git rebase`
정의	브랜치 변경 사항을 병합 (merge commit 포함)	브랜치 변경 사항을 기준 브랜치 위로 재적용
히스토리	분기점과 병합점이 **남는다**	선형(Linear) 히스토리로 정리됨
커밋 해시	**변하지 않음**	**다시 만들어짐 (해시 변경)**
병합 방식	**삼방 병합 (3-way merge)**	**기반 브랜치 위에 커밋들을 재적용**

항목	`git merge`	`git rebase`
충돌 처리	병합 시 한 번 처리	각 커밋마다 발생할 수 있음
사용 목적	**협업 브랜치 통합**, 병합 이력 추적용	**개인 브랜치 정리**, 깔끔한 커밋 그래프용

✅ 2. 히스토리 예시

📌 `git merge`

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

# 커밋 F는 merge 커밋 (D~E + C 병합)
```

→ **히스토리에 병합 흔적이 남음**

📌 `git rebase`

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

→ **커밋이 다시 만들어져 깔끔한 일자형 로그가 됨**
(`D'`, `E'`는 `D`, `E`의 복제본)

✅ 3. 명령어 비교

`git merge`

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

- 병합 커밋이 생성됨
- 커밋 히스토리 유지
- 협업 중에도 안전

`git rebase`

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

- `feature` 브랜치의 커밋들이 `main` 이후에 재배치
- 커밋 해시 변경
- **push 후에는 위험!**

✅ 4. 실전 상황별 선택 기준

상황	추천
협업 브랜치 병합 (PR)	✅ `merge`
본인의 기능 브랜치 정리	✅ `rebase`
커밋 히스토리 추적이 중요	✅ `merge`
히스토리를 선형으로 유지하고 싶을 때	✅ `rebase`
다른 사람이 같은 브랜치를 pull한 경우	❌ `rebase` (위험)
리뷰 전 깔끔하게 정리	✅ `rebase -i` + `squash`

✅ 5. 충돌 발생 시 차이

항목	`merge`	`rebase`
충돌 빈도	한 번	여러 번 (각 커밋마다)
처리 후	`git commit`	`git rebase --continue`
중단	`git merge --abort`	`git rebase --abort`

✅ 6. GitHub/GitLab에서의 선택

- 기본 PR 병합 전략: `merge commit` (기록 보존)
- 옵션:
  - **Squash and merge** → `rebase` + `squash` 효과
  - **Rebase and merge** → 선형 히스토리 유지, 단 push 주의 필요

✅ 7. 마무리 요약

항목	`merge`	`rebase`
히스토리	병합 지점 존재	선형 정리됨
커밋 해시	그대로	변경됨
협업 안정성	높음	위험할 수 있음
커밋 보기	복잡함	깔끔함
push 이후 사용	안전	위험 (force-push 필요)

🔐 결론

- **협업 브랜치**: `merge`가 안전하고 권장됨
- **혼자 작업한 기능 브랜치**: `rebase`로 깔끔하게 정리한 후 PR
- 리베이스 후에는 반드시:

```
git push --force-with-lease
```

`git rebase -i`로 커밋 정리 (squash, fixup 등)

✅ 1. 기본 개념

> `git rebase -i`는 선택한 커밋 범위를 **인터랙티브하게 조작**하여 커밋 히스토리를 깔끔하게 정리할 수 있는 명령어이다.

```
git rebase -i HEAD~n
```

- `HEAD~n`: 최근 n개의 커밋을 대상으로 함
- 실행하면 편집기에서 커밋 리스트가 열리고 각 커밋을 `pick`, `reword`, `squash` 등으로 조작할 수 있음

✅ 2. 주요 키워드 요약

키워드	설명
`pick`	커밋 유지 (기본값)
`reword`	커밋 메시지만 수정
`edit`	커밋 내용도 수정 (임시 정지 후 수동 커밋)
`squash`	이전 커밋과 **내용 + 메시지를 합침**
`fixup`	이전 커밋과 **내용은 합치되, 메시지는 버림**
`drop`	커밋 삭제

✅ 3. 예시

📌 현재 커밋 히스토리

```
git log --oneline
```

```
a3c1d9c (HEAD) fix: typo in login
b2e1a2b feat: add login validation
81d9fbb feat: add login form
```

이제 최근 3개 커밋을 `rebase -i`로 정리해보자.

📌 인터랙티브 리베이스 실행

```
git rebase -i HEAD~3
```

열리는 편집기 (예시)

```
pick 81d9fbb feat: add login form
pick b2e1a2b feat: add login validation
pick a3c1d9c fix: typo in login
```

🔁 커밋 정리: `squash`, `fixup`

```
pick 81d9fbb feat: add login form
squash b2e1a2b feat: add login validation
fixup a3c1d9c fix: typo in login
```

> ✅ 결과:

- **3개의 커밋이 하나로 합쳐지고**,
- `squash` 커밋은 메시지를 합쳐서 편집하게 되고,
- `fixup`은 메시지가 자동으로 제거됨

📌 메시지 편집 (squash 시)

```
# This is a combination of 3 commits.
# The first commit's message is:
feat: add login form

# The 2nd commit message:
feat: add login validation

# The 3rd commit message:
fix: typo in login
```

→ 여기에 원하는 커밋 메시지 하나만 남기고 저장하면 됨.

✅ 4. 커밋 순서 바꾸기

```
pick 81d9fbb feat: add login form
pick b2e1a2b feat: add login validation
```

→ 순서를 아래처럼 바꾸면, 커밋도 그 순서대로 적용됨

```
pick b2e1a2b feat: add login validation
pick 81d9fbb feat: add login form
```

> 📌 단, 충돌 가능성 있음

✅ 5. 커밋 내용 자체 수정 (`edit`)

```
pick 81d9fbb feat: add login form
edit b2e1a2b feat: add login validation
pick a3c1d9c fix: typo in login
```

실행 중 Git이 중단되며:

```
Stopped at b2e1a2b... feat: add login validation
You can amend the commit now, with

  git commit --amend

Once you're satisfied with your changes, run

  git rebase --continue
```

✅ 6. 리베이스 중단 / 롤백

- 중단:

  ```
  git rebase --abort
  ```

- 충돌 해결 후 계속:

  ```
  git rebase --continue
  ```

✅ 7. 주의사항

항목	설명
히스토리 변경	커밋 해시가 전부 바뀜
push 전에만 사용	이미 공유한 브랜치에는 절대 X
강제 push 필요	`git push --force-with-lease`

✅ 8. 실전 활용 예시

기능 개발 중 중간 커밋 정리:

```
feat: implement form
squash: add validation
fixup: fix typo
↓
최종 결과
feat: implement login form with validation
```

✅ 마무리 요약

목적	키워드	설명
메시지만 수정	`reword`	내용 유지, 메시지만 변경
커밋 병합	`squash`	메시지 + 내용 모두 합침
커밋 덧붙이기	`fixup`	메시지 제거, 내용만 합침
커밋 삭제	`drop`	해당 커밋 완전 제거
내용 수정	`edit`	수동으로 커밋 수정 가능

`git commit --amend`

✅ 1. 기본 개념

> `git commit --amend`는 **가장 최근 커밋을 수정**하는 명령이다.

이때,

- 커밋 메시지를 수정할 수 있고,
- 파일 내용을 추가하거나 제거해서 커밋 내용 자체를 바꿀 수도 있음.

✅ 2. 사용 상황

상황	예시
오타가 있는 커밋 메시지	`"fix: slep mode"` → `"fix: sleep mode"`
파일을 추가하지 않고 커밋해버림	새 파일을 stage한 뒤 `--amend`로 덧붙이기
`git rebase -i` 중 `edit` 커밋 수정	`git commit --amend` + `git rebase --continue`

✅ 3. 메시지만 수정하기

```
git commit --amend
```

- 텍스트 편집기가 열리며, 메시지를 수정할 수 있음
- 기존 커밋을 그대로 덮어쓴다.

> 📌 커밋 해시가 바뀌므로 **push한 커밋을 amend하면 강제 push 필요**

✅ 4. 새 파일 추가하고 amend

```
# 아까 빠뜨린 파일 추가
git add forgot.txt

# 기존 커밋에 포함시키기
git commit --amend
```

> 결과적으로 **기존 커밋이 수정되며**, 새 파일도 포함된 하나의 커밋으로 남음

✅ 5. 명령어 한 줄로 메시지 수정

```
git commit --amend -m "fix: corrected sleep mode bug"
```

> 편집기 안열고 바로 메시지 수정

✅ 6. 실습 예시

```
echo "a" > file.txt
git add .
git commit -m "fix: slep mode"

# 오타 발견!
git commit --amend -m "fix: sleep mode"
```

→ 커밋 해시는 바뀌고, 메시지가 새로 저장됨

✅ 7. push한 후에 amend 했다면?

```
git push --force-with-lease
```

> **중요:** 기존 커밋 해시와 달라졌기 때문에
> 원격 저장소에서 덮어쓰기(push -f)가 필요하다.

✅ 8. 주의사항

항목	설명
커밋 해시 변경	무조건 바뀜 (`--amend`는 rewrite 명령)
협업 브랜치 주의	이미 push한 경우 동료에게 혼란 줄 수 있음
로컬에서만 안전하게	PR 올리기 전 정리용으로는 매우 유용

✅ 9. `git commit --amend` vs `rebase -i`

항목	`--amend`	`rebase -i`
커밋 대상	최근 1개 커밋	여러 개 가능
수정 범위	메시지/내용	메시지, 순서, squash 등
실수 정정	빠르게 가능	더 복잡한 정리 가능
push 이후 사용	주의 필요 (강제 push)	동일

✅ 마무리 요약

기능	예시
메시지 수정	`git commit --amend`
파일 덧붙이기	`git add <file>` 후 `--amend`
메시지 직접 입력	`--amend -m "수정된 메시지"`
강제 push 필요	`git push --force-with-lease`

`git reset` (soft, mixed, hard)

✅ 1. 기본 개념

```
git reset [<mode>] <commit>
```

- `<mode>`는 `--soft`, `--mixed`(기본값), `--hard`
- `<commit>`은 되돌리고 싶은 커밋의 해시(또는 HEAD~1 등)

> 💡 `git reset`은 총 **세 가지 대상**에 영향을 줄 수 있다:

- **HEAD** (브랜치 포인터)
- **Index (Staging Area)** — `git add`된 상태
- **Working Directory** — 실제 파일

✅ 2. 옵션별 동작 비교

옵션	HEAD 이동	Index 변경	작업 디렉토리 변경	설명
`--soft`	✅ 이동	❌ 그대로	❌ 그대로	커밋만 되돌림 (add 상태 유지)
`--mixed` (기본)	✅ 이동	✅ 초기화됨	❌ 그대로	커밋 & add 상태 초기화
`--hard`	✅ 이동	✅ 초기화	✅ 초기화	커밋, add, 파일까지 완전 초기화

✅ 3. 각 옵션 설명 + 예제

🔸 `git reset --soft`

> 가장 최근 커밋을 없애고, 변경 내용을 staged 상태로 유지

```
git reset --soft HEAD~1
```

- `HEAD`는 한 커밋 이전으로 이동
- 커밋은 사라지지만, 변경 내용은 여전히 `git status`에 **staged** 상태로 남아있음

📌 **재커밋을 위한 커밋 수정, 병합 등에서 자주 사용**

🔸 `git reset --mixed` (기본값)

> 커밋도 없애고, staged 상태도 초기화. 파일은 그대로 유지

```
git reset HEAD~1
```

또는

```
git reset --mixed HEAD~1
```

- `HEAD` 한 칸 되돌림
- staging 영역 비움 (`git add` 초기화)
- 워킹 디렉토리에는 파일 그대로 있음

📌 **실수로 `git add`한 것 취소하고 싶을 때 유용**

🔸 `git reset --hard`

> ❗ 모든 것 제거. 커밋, staged, 작업 중 파일까지 완전히 이전 상태로 초기화

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

- 해당 커밋 이후의 **모든 변경 사항 사라짐**
- 복구 불가한 상황이 될 수 있음 (⚠️ 주의)

📌 **정말 필요할 때만 사용, 되도록 백업 후**

✅ 4. 시각적 예시

예) 현재 히스토리

```
A---B---C (HEAD)
          ↑ 현재 브랜치
```

```
git reset --soft B
```

결과:

- 브랜치: B로 이동
- C의 변경사항은 **staged 상태**로 남음

```
git reset --mixed B
```

- 브랜치: B로 이동
- C의 변경사항은 **unstaged 상태**로 남음

```
git reset --hard B
```

- 브랜치: B로 이동
- C의 변경사항은 **모두 사라짐**

✅ 5. 주의 사항

항목	주의
되돌릴 수 없음 (`--hard`)	실수하면 복구 불가
협업 브랜치에 사용 금지	push 후 reset은 히스토리 꼬임 발생
안전한 되돌리기	`git revert`가 협업 시 더 안전

✅ 6. 실무 활용 팁

상황	해결 방법
최근 커밋 취소하고 메시지만 바꾸고 싶을 때	`git reset --soft HEAD~1` + `git commit -m "..."`
add 했던 파일 모두 unstage 하고 싶을 때	`git reset` (`--mixed`)
개발 중 이상한 상태로 꼬였을 때 완전 초기화	`git reset --hard HEAD`
과거 특정 커밋으로 강제로 브랜치 돌리기	`git reset --hard <커밋 해시>`

✅ 7. 강제 push (주의!)

> `reset` 이후에는 **로컬 히스토리가 바뀌므로**, 이미 push된 경우:

```
git push --force-with-lease
```

- 되도록 팀원과 상의 후 진행
- `--force-with-lease`는 **상대방 변경을 덮지 않도록 보호**

✅ 마무리 요약

명령어	설명
`git reset --soft HEAD~1`	커밋만 취소, staged 유지
`git reset --mixed HEAD~1`	커밋 + staging 취소, 파일 유지
`git reset --hard HEAD~1`	완전 초기화 (파일 삭제 포함)

`git reflog`로 복구

✅ 1. 개념 요약

> `git reflog`는 **HEAD가 어디로 이동했는지를 시간순으로 기록**하는 로그다.

- `git log`는 **존재하는 커밋만 보여줌**
- `git reflog`는 **삭제되거나 HEAD가 이동했던 모든 기록**을 보여줌
- 이를 통해 **실수로 삭제한 브랜치, 커밋도 복구 가능**함

✅ 2. 사용법

```
git reflog
```

예시 출력:

```
bc3e5f7 HEAD@{0}: commit: fix: change password bug
a4df21c HEAD@{1}: commit (amend): fix typo
f145c3a HEAD@{2}: reset: moving to HEAD~1
bd234ab HEAD@{3}: commit: feat: login API 추가
```

- `HEAD@{n}` 형식: HEAD가 변경된 시점
- 해시 값: 해당 시점의 커밋 해시
- 메시지: 어떤 작업이었는지 보여줌 (commit, reset, checkout 등)

✅ 3. 실전 예시: 커밋 날렸을 때 복구

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

실수로 중요한 커밋 날렸다고 해보자.

🔄 복구 단계

```
git reflog
```

출력에서 날려버린 커밋을 찾고 예: `bd234ab HEAD@{3}`

```
git checkout bd234ab
```

→ 커밋 상태로 돌아옴 (Detached HEAD)

👉 브랜치로 살리기

```
git checkout -b recover-login-api
```

→ 날린 커밋 복구 완료

✅ 4. 실전 예시: 브랜치 날렸을 때 복구

```
git branch -D feature-ui
```

😱 브랜치 통째로 삭제했는데 복구하고 싶다?

```
git reflog
```

→ `feature-ui`가 HEAD였던 시점 찾기

```
git checkout -b feature-ui <해당 커밋 해시>
```

✅ 5. 실전 예시: `git reset --hard` 복구

```
git reset --hard HEAD~3
```

→ 마지막 3개 커밋까지 몽땅 날아감

복구:

```
git reflog
# 예: f37a456 HEAD@{4}
git reset --hard f37a456
```

✅ 6. `git log` vs `git reflog`

항목	`git log`	`git reflog`
보여주는 범위	현재 브랜치의 커밋	HEAD 이동 이력 전부
삭제된 커밋 포함 여부	❌ 없음	✅ 포함
reset, checkout 기록	❌ 없음	✅ 있음
브랜치 삭제 복구	❌ 불가	✅ 가능

✅ 7. reflog 보존 정책

- `.git/logs/` 폴더에 저장됨
- 보통 90일 또는 `gc.reflogExpire` 정책에 따라 보관됨
- 너무 오래되면 자동 삭제되니 주의

```
git config --get gc.reflogExpire
# 기본: 90 days
```

✅ 마무리 요약

기능	명령어
HEAD 이동 이력 보기	`git reflog`
과거 커밋 체크아웃	`git checkout HEAD@{n}`

기능	명령어
삭제된 브랜치 복구	`git checkout -b <이름> <커밋 해시>`
날린 커밋 복구	`git reset --hard <커밋 해시>`

🎯 `git reflog`는 **리셋/강제 푸시/브랜치 삭제 등으로 날아간 커밋을 되살리는 유일한 방법**이다.
→ 항상 **커밋 해시**를 잊지 말고, 실수했다면 `reflog` 먼저 봐야 한다.