Git초 local

작성자 : 이준호
작성일 : 2022. 02. 09
수정일 : 2023. 01. 08
버전: 0.5

TOTORo

totOroprog@gmail.com
https://totOrokr.github.io
https://github.com/TOTORoKR

목차

Local Repo 명령어

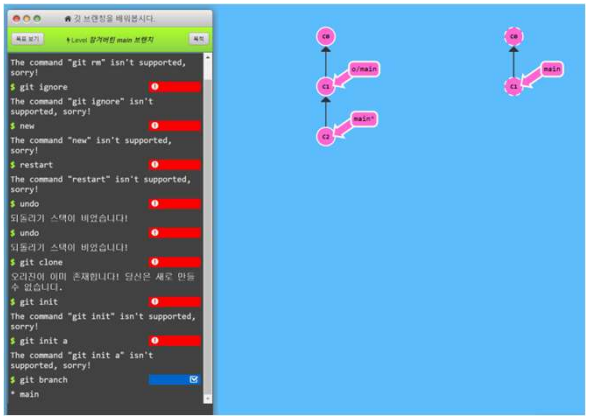
목차

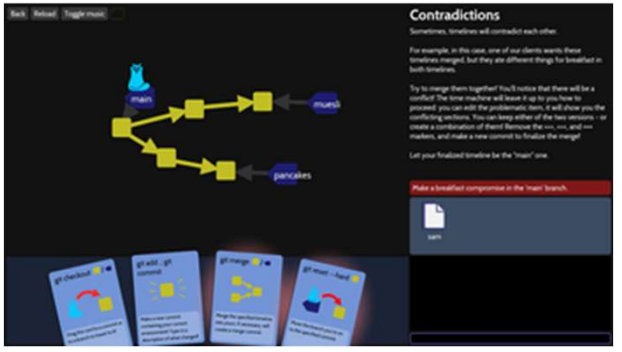
Remote Repo 명령어

목차

기타 명령어

- git tag
- git config
- git log
- git diff
- patch
- git stash
- git reflog





Git 연습 사이트

- https://learngitbranching.js.org/?locale=ko: 트레이닝 사이트.
- https://blinry.itch.io/oh-my-git: 게임

목차

Advanced 명령어

Local Repo 명령어

TOTORo

totOroprog@gmail.com
https://totOrokr.github.io
https://github.com/TOTORoKR

Background

■**prefix '$': shell 명령어**

■**중괄호 '{ }': 필수 옵션**

■**대괄호 '[ ]': 선택 옵션**

■**바 '|': or**

■**" ": 띄어쓰기가 필요한 경우 쌍따옴표 사용**

■**더블 대시 '--': 옵션 입력이 끝났음을 의미**

■**Work tree: 작업 디렉토리. 실제 파일 내용 - 실시간으로 수정**

■**Staging area: Commit을 통해 Repository에 반영할 내용 - 일종의 Cache**

- Index: 현재 Commit 내용에 staging area 내용을 더한 내용

■**Repository: Commit된 내용**

- Git directory: git에 저장된 내용

git init

■새로운 repository 생성

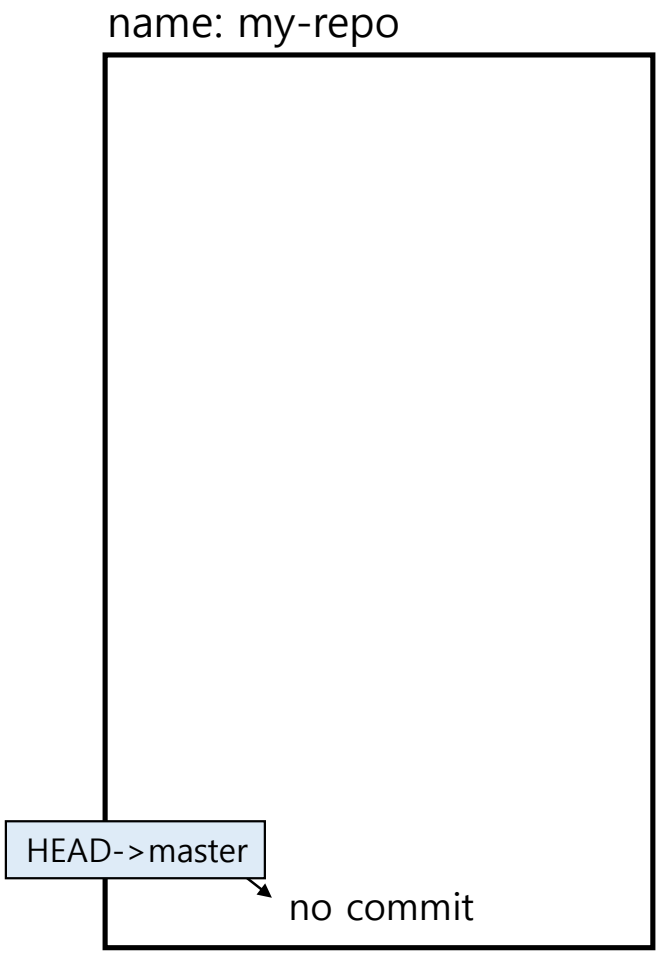
- $ git init
  - git repository의 정보를 담은 .git 디렉토리 생성

■파일 .gitignore

- 해당 파일에 적힌 파일은 추적하지 않음 (commit에 반영되지 않음)

■예제

- $ mkdir my-repo
- $ cd my-repo
- $ git init
- (master) $



git log & config - early

■사용자 이름 및 이메일 설정

- $ git config user.name <NAME>
- $ git config user.email <이메일>
- -- global 옵션을 주면 ~/.gitconfig에 설정되어 전역으로 사용 가능

■commit log 확인

- $ git log
  - HEAD가 가장 최신 commit으로 간주
- $ git log --all
  - 전체 로그 확인 (local, remote 모든 branch 확인)
- $ git log --graph --abbrev-commit --oneline
  - graph 형식으로 보이면서
  - 40자 HASH 값->구별할 수 있는 일부만 보여주고
  - 핵심만 1줄로 요약

■log 명령어 alias

- $ git config --global **alias.lg** "log --graph --abbrev-commit --oneline"
  - $ git lg: 위의 명령이 실행됨

git add & status & commit - early

■파일 생성 - Work-tree에 반영

- ➢ 예: README.md 파일 생성
  - $ git status

```
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        README.md

nothing added to commit but untracked files present
(use "git add" to track)
```

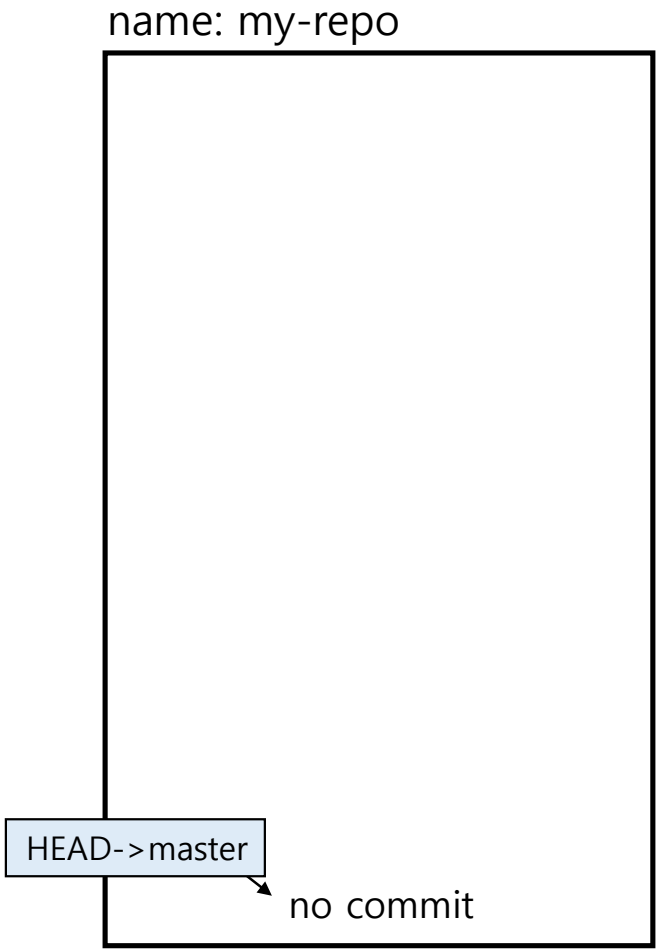
■Staging area에 추가 = Index에 반영

- ➢ $ git add <PATH>
  - $ git status

```
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   README.md
```

■Commit 작성 - Repository에 반영

- ➢ $ git commit -m <커밋 메시지>

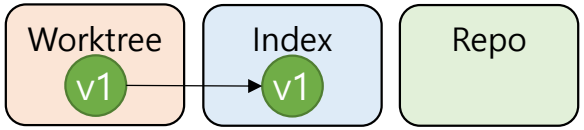


git add & status & commit - early

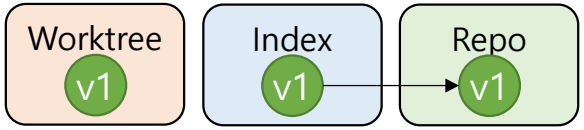
현 상태

v1 : file v1

■git add

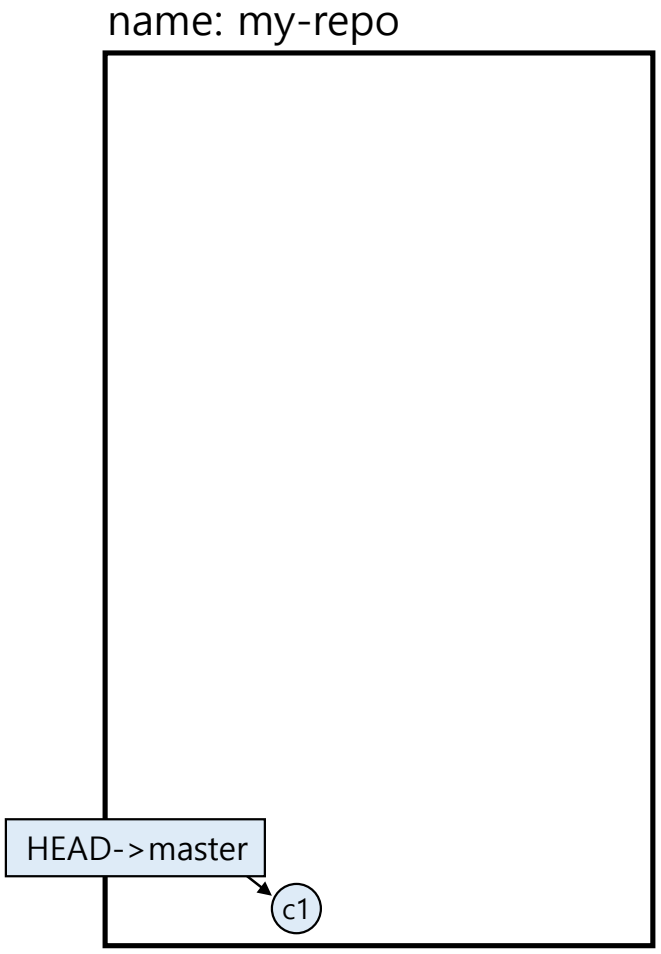


■git commit



■예제

- (master) $ echo "# Read Me" > README.md
- (master) $ git add README.md
- (master) $ git commit -m "Repository init"



※ 여기서 c1은 commit id이며 실제로는 18e4aa0dc08f15bdd055a1d6ccb1986031fc0f87와 같은 HASH 값으로 표현됨

git branch

■새로운 branch 생성

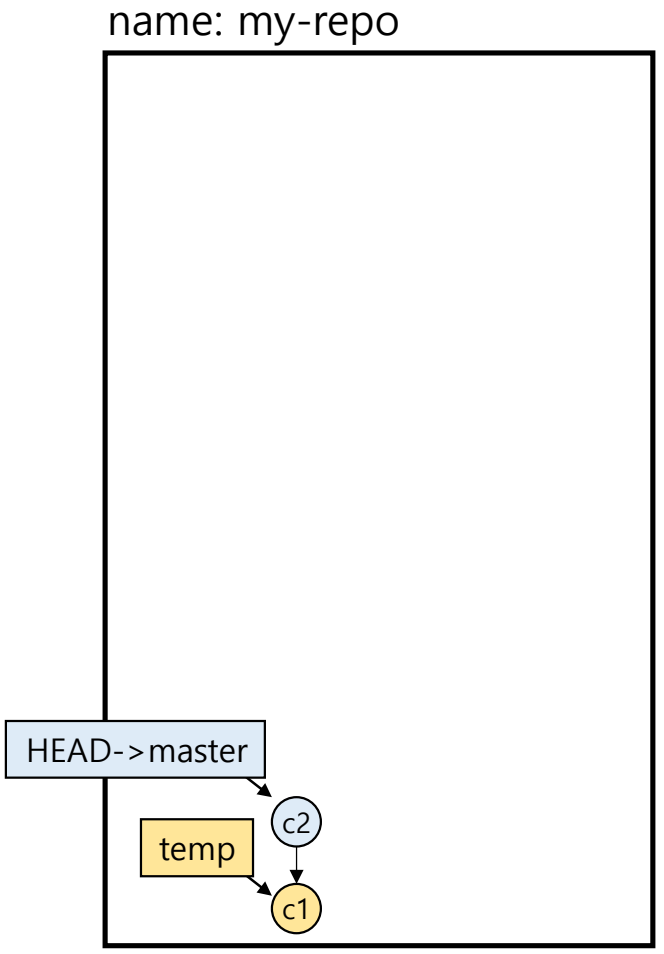
- $ git branch <이름>

■branch 정보 확인

- $ git branch
  - 로컬 branch 정보. -l 옵션이 생략되어있음
- $ git branch -v
  - 로컬 branch 정보 + 마지막 커밋
- $ git branch -r
  - 리모트 branch 정보
- $ git branch -a
  - 모든 branch 정보

■예제

- (master) $ git branch temp
- (master) $ echo "Apache License Version 2.0, January 2004" > LICENSE
- (master) $ git add LICENSE
- (master) $ git commit -m "License"



git branch - 2

■branch 이름 변경

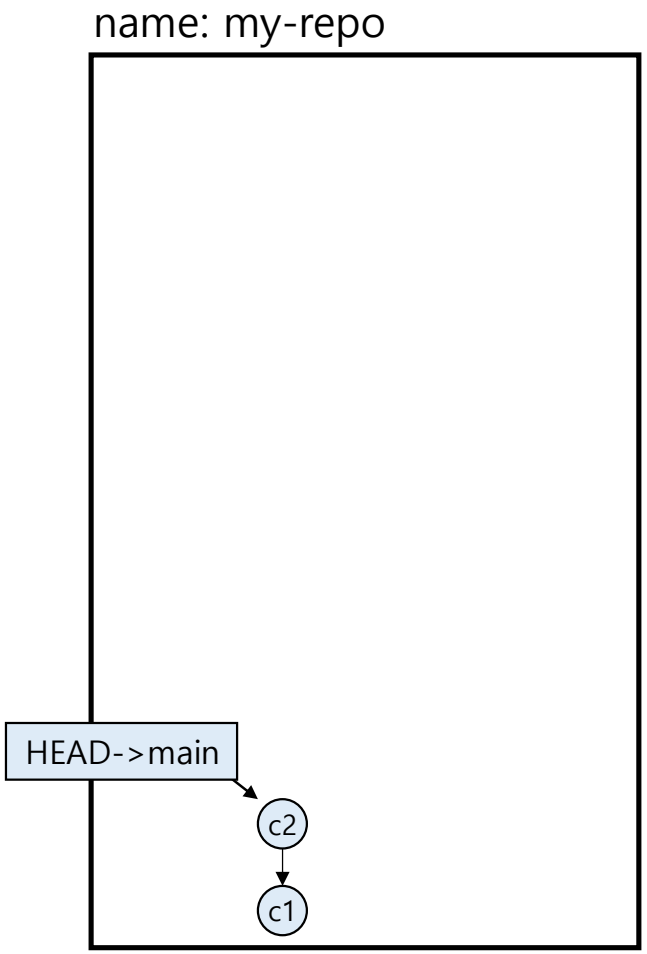
- $ git branch -m <SRC> <DST>:
  - branch 명을 <SRC>에서 <DST>로 변경

■branch 삭제

- $ git branch -d <브랜치>
  - <브랜치> branch를 제거

■예제

- (master) $ git branch -d temp
- (master) $ git branch -m master main
- (main) $



git checkout

■HEAD

- 어떤 commit 위에서 작업 중인지 가리키는 포인터

■Working branch

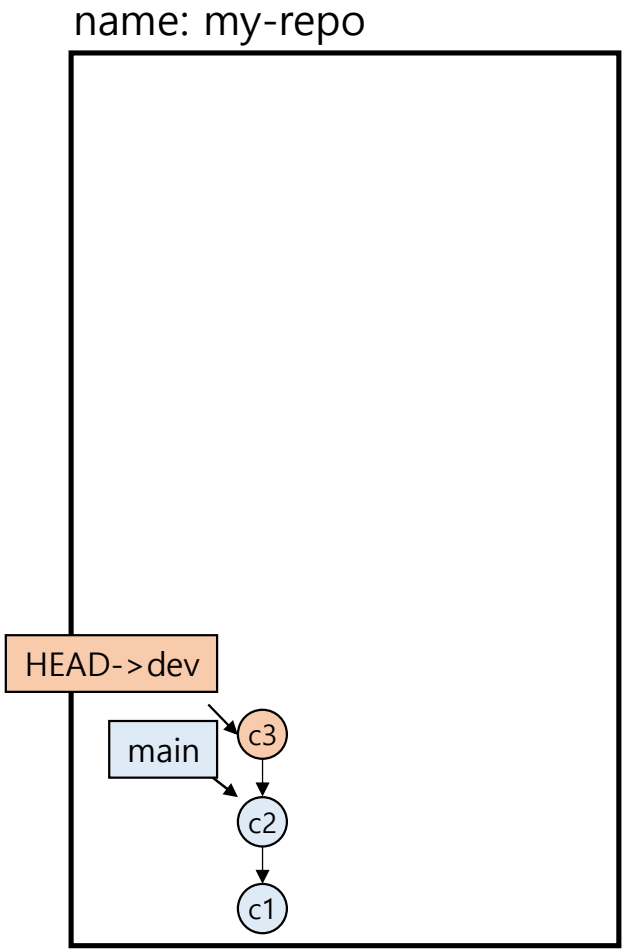
- HEAD가 가리키는 branch

■원하는 branch에서 작업하기

- $ git checkout <브랜치>
  - HEAD를 <브랜치>으로 변경
- $ git checkout -b <브랜치>
  - <브랜치> branch를 생성하고 HEAD를 <브랜치>으로 변경

■예제

- (main) $ git checkout -b dev
- <무언가 파일을 생성 및 수정 작업 & git add> **※ 추후 수정 및 add 과정 생략**
- (dev) $ git commit



git checkout - 2

■원하는 commit에서 작업하기

- $ git checkout <커밋>
  - ※ 40자 HASH 전부가 아닌, 겹치지만 않으면 앞에서부터 7~8자리만 작성해도 됨

■예제

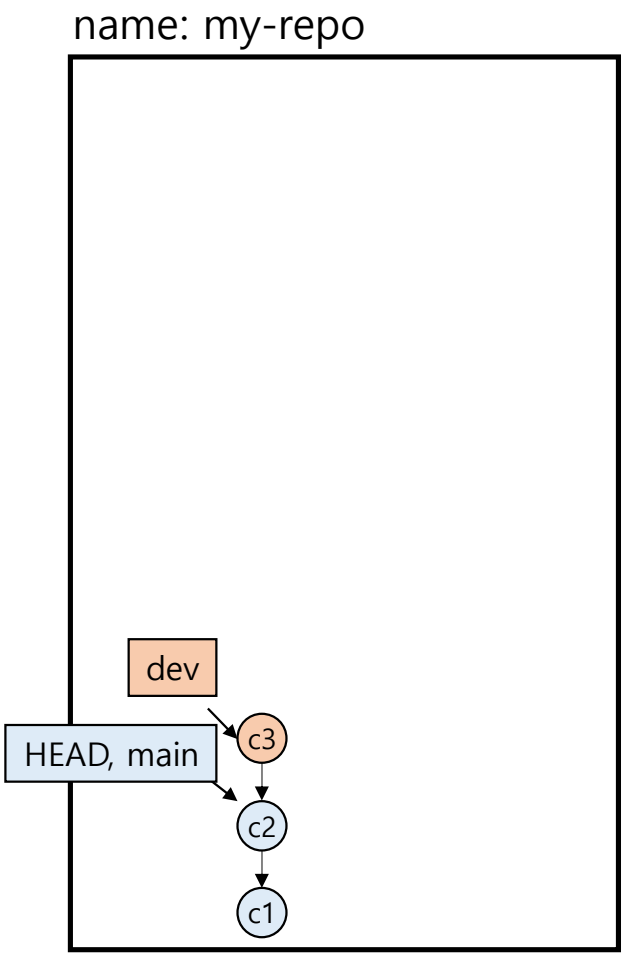
- (dev) $ git checkout c2
- ((c2)) $

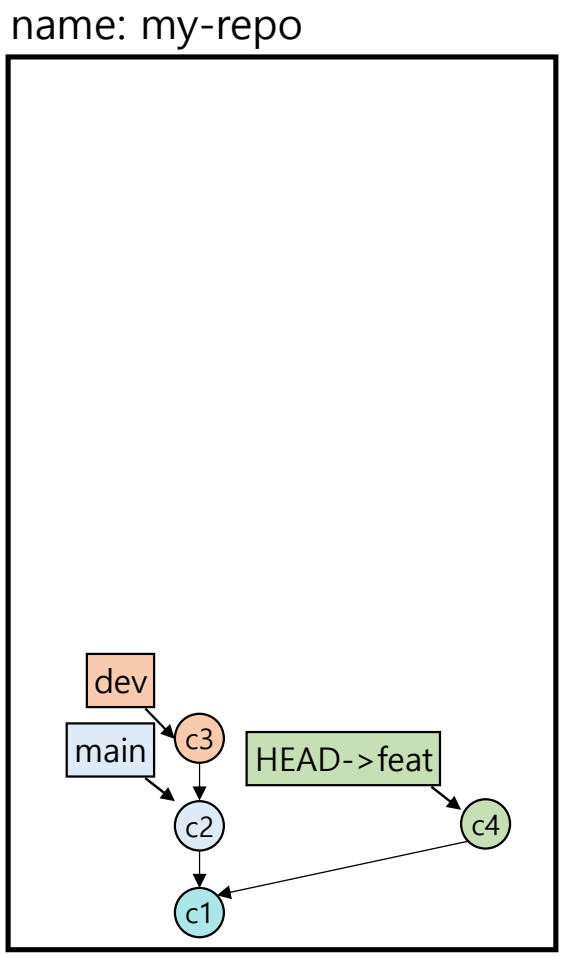
■상대참조

- ^: 개수에 따른 상위 commit
  - HEAD^^: HEAD에서 2개 상위 commit
- ~<NUM>: 숫자에 따른 상위 commit
  - main~5: main branch에서 5개 상위 commit

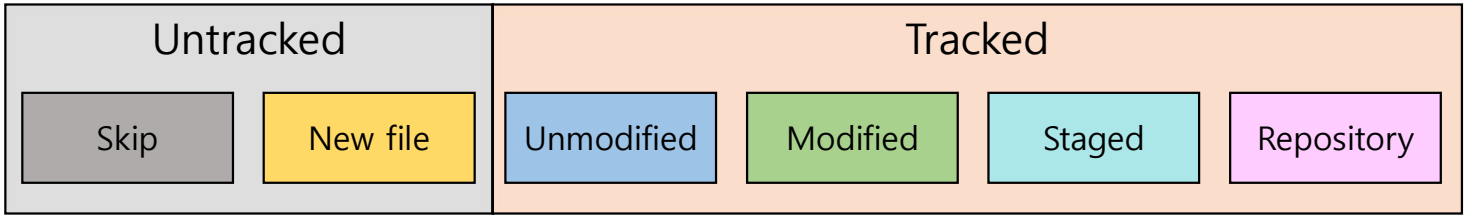
■예제

- ((c2)) $ git checkout HEAD^
- ((c1)) $ git checkout -b feat
- (feat) $ git commit



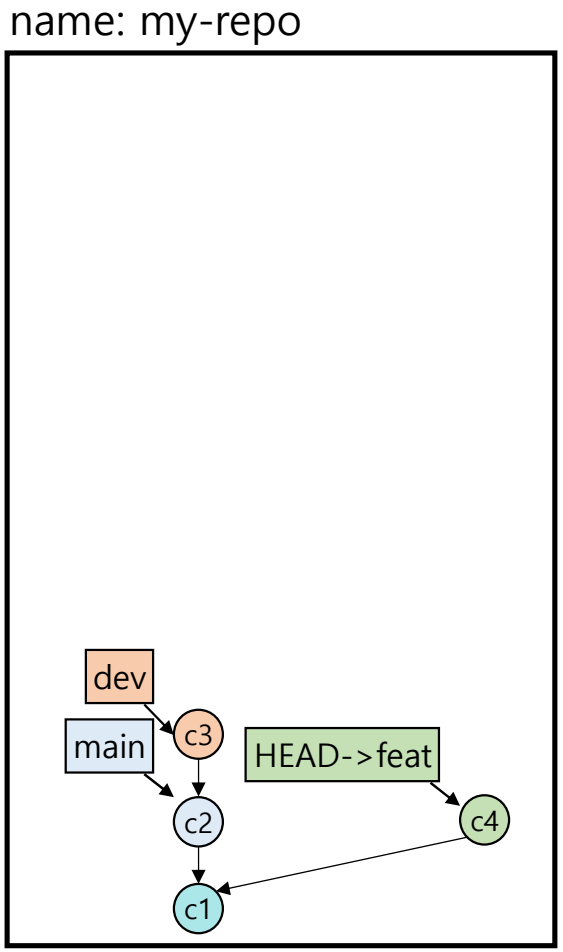


git status

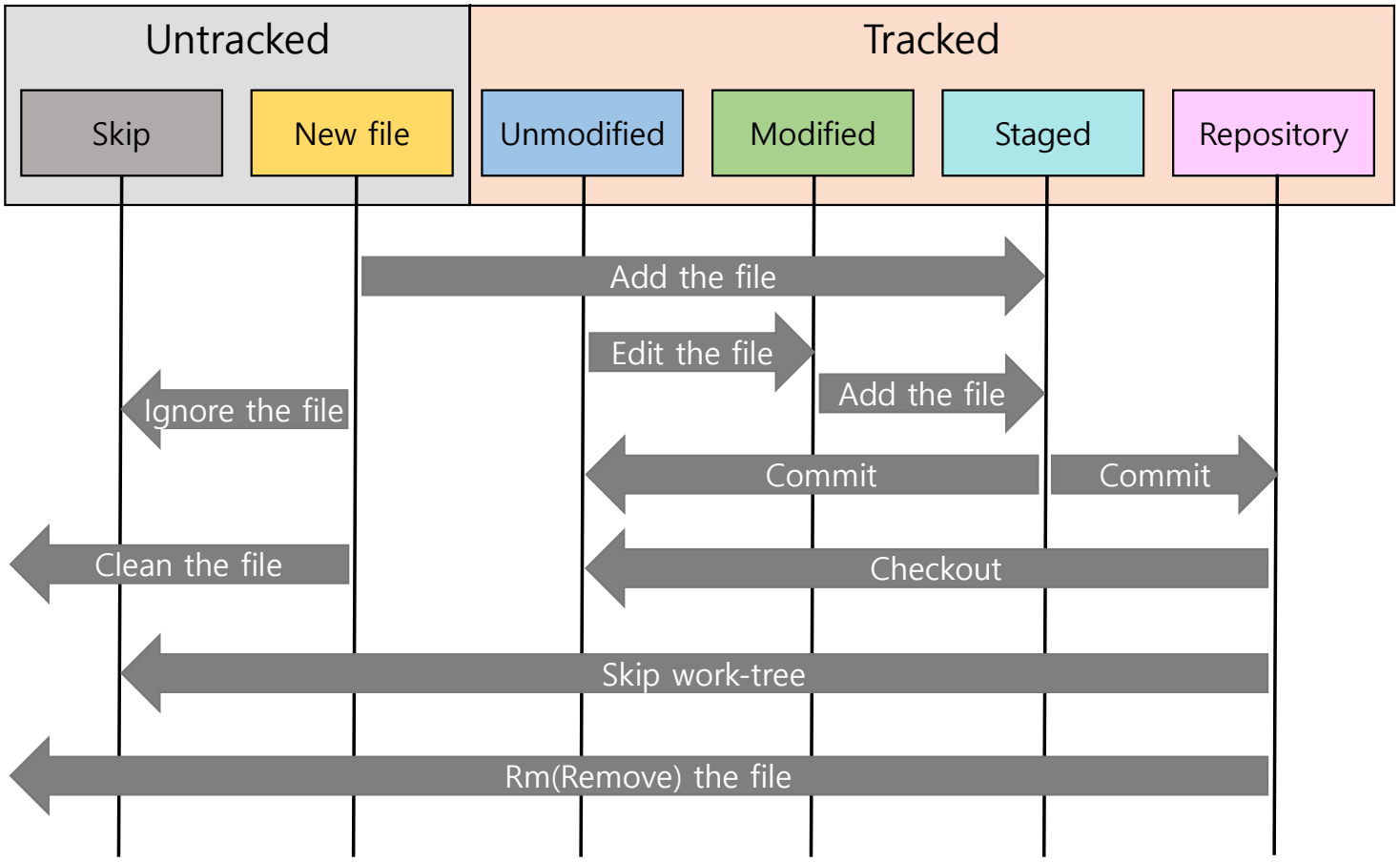


■파일 상태

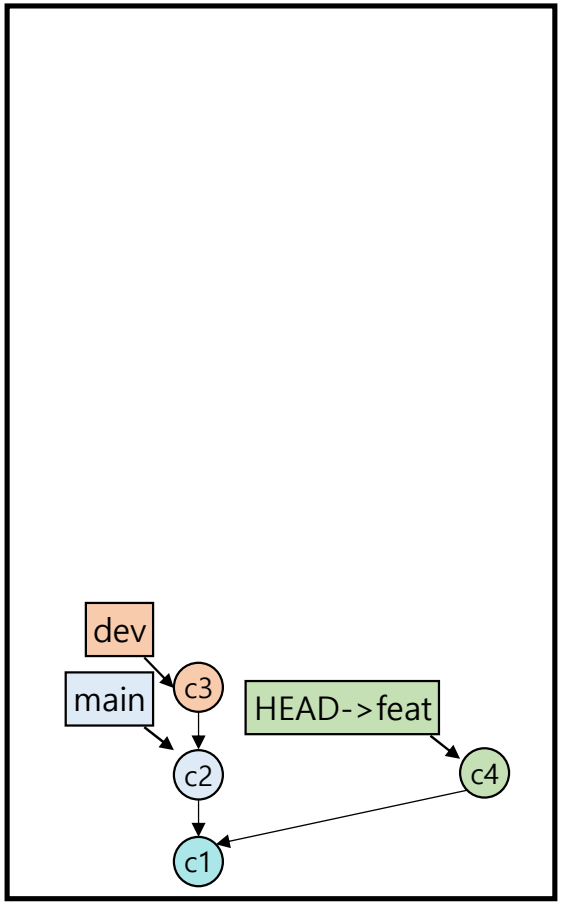
- **Untracked:** git이 관리하지 않는 파일
- **Tracked:** git이 관리하는 파일
- **Skip:** working tree에서 더 이상 추적하지 않는 파일
- **New file:** 새로 생성한 파일
- **Unmodified:** 현재 commit에 적용된 파일
- **Modified:** 수정된 파일
- **Staged:** staging area에 올라간 상태
- **Repository:** git directory에 올라간 상태



git status - 2



name: my-repo



git add

■파일을 Stage에 올리기

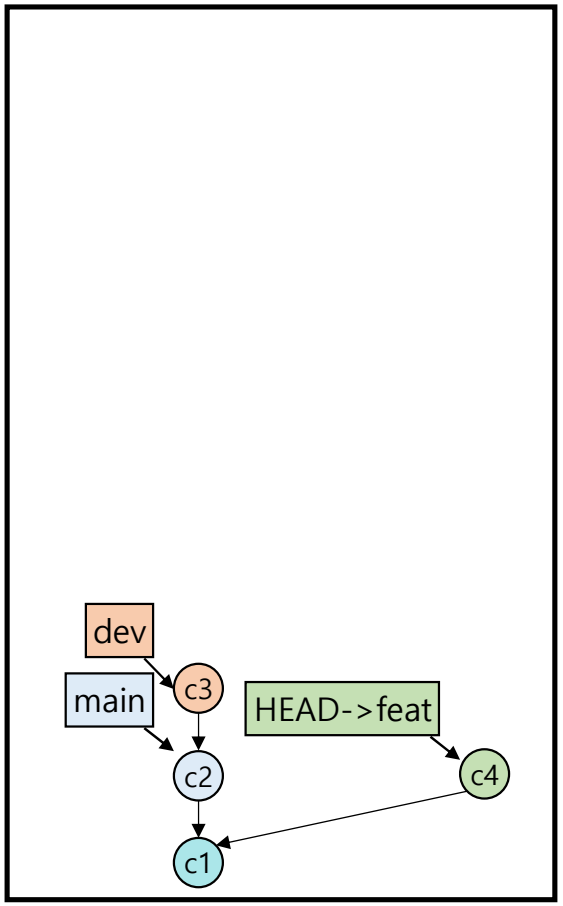
- ➢ $ git add <파일>
  - 현재 반영된 <파일>을 stage에 올림.
  - Staging area에 올린 파일을 추가 수정하면, 해당 부분은 un-staged 상태임
- ➢ $ git add <디렉토리>
  - <디렉토리> 하위의 파일을 모두 staging area 올림
  - ex) $ git add . : 현재 디렉토리 하위의 파일 대상
- ➢ $ git add -A
  - Working tree에 있는 모든 파일을 staging area에 올림
- ➢ $ **git add -p**
  - **(※추천)** tracked 파일 중에 변경 부분을 따라가면서, stage하고 싶은 hunk를 확인하고 반영. Stage 하고싶은 hunk만! **(untracked 파일은 무시. 이게 꿀)**
  - 해당 명령을 치고 응답형 명령어로 '?' 를 치면 설명을 볼 수 있음.

■예제

- ➢ (feat) $ git add -p

```
tot0ro@DESKTOP-JUNHO:~/my-repo$  (base) git add -p
diff --git a/temp1.txt b/temp1.txt
index efb645a..db7370f 100644
--- a/temp1.txt
+++ b/temp1.txt
@@ -1,2 +1,2 @@
 Temp1 textfile
-This is commited on c4
+This is commited on c5
(1/1) Stage this hunk [y,n,q,a,d,e,?]?
```

name: my-repo



git commit

■Staged 파일을 Git directory에 올리기

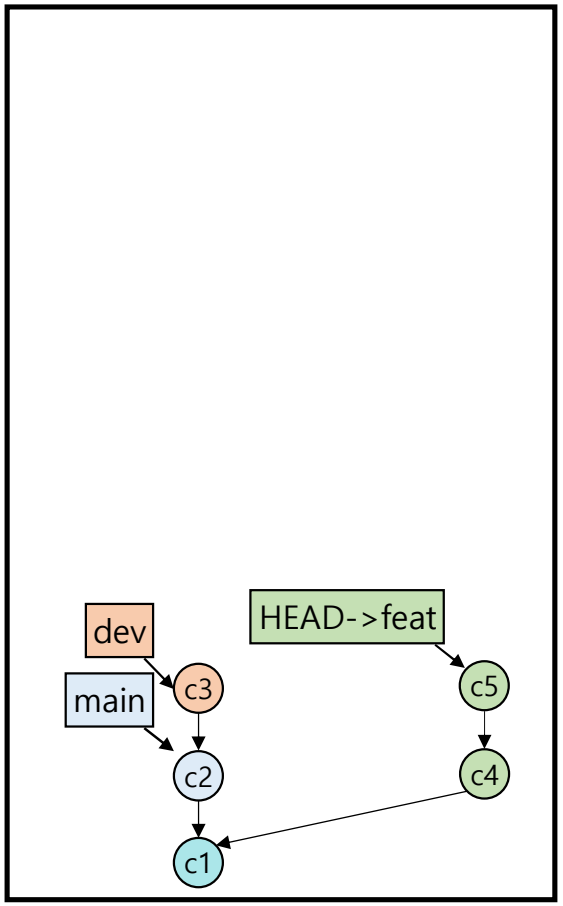
- ➢ $ git commit -m <메시지>
  - 메시지 한 줄로 간략하게 commit 생성
- ➢ $ git commit
  - 에디터로 메시지 작성. commit에 반영되는 파일을 확인할 수 있음
- ➢ $ **git commit -v**
  - **(※추천)** 변경사항까지 확인 가능.
- ➢ 에디터 변경
  - $ git config --global core.editor vim

■예제

- ➢ (feat) $ git commit -v

```
Modify temp1

# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
#
# On branch base
# Changes to be committed:
#       modified:   temp1.txt
#
# ------------------------ >8 ------------------------
# Do not modify or remove the line above.
# Everything below it will be ignored.
diff --git a/temp1.txt b/temp1.txt
index efb645a..db7370f 100644
--- a/temp1.txt
+++ b/temp1.txt
@@ -1,2 +1,2 @@
 Temp1 textfile
-This is commited on c4
+This is commited on c5
```

name: my-repo



git commit - 2

■Commit 메시지 작성 요령

■Commit 메시지 7가지 규칙

1. 제목과 본문을 빈 행으로 구분합니다.
2. 제목을 50글자 이내로 제한합니다.
3. 제목의 첫 글자는 대문자로 작성합니다.
4. 제목의 끝에는 마침표를 넣지 않습니다.
5. 제목은 명령문으로! 과거형을 사용하지 않습니다.
6. 본문의 각 행은 72글자 내로 제한합니다.
7. 어떻게 보다는 **무엇**과 **왜**를 설명합니다.

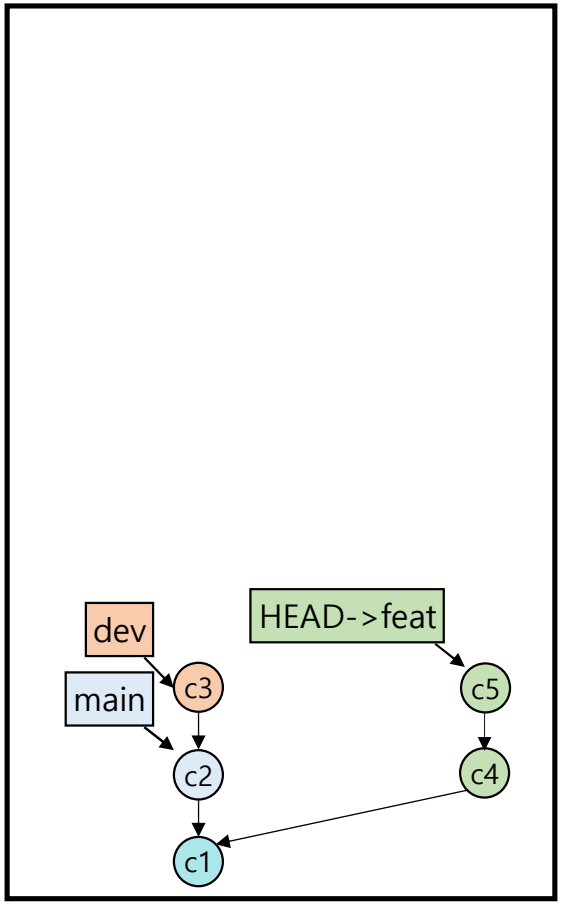
<type> 종류
feat : 새로운 기능
fix : 오류 및 이슈 해결
build : 빌드 관련 파일 수정
chore : 그 외 자잘한 수정(기타 변경)
ci : CI 관련 설정 수정
docs : 문서 수정
style : 코드 스타일 혹은 포맷 등
refactor : 코드 리팩토링
test : 테스트 코드 수정

■Commit 메시지 구조

<type>(<scope>): <subject>	--헤더
<BLANK LINE>	--빈 줄
<body>	--본문

- <type>은 해당 commit의 성격을 나타내며 아래 중 하나여야 합니다.
- <body>는 본문으로 헤더에서 생략한 상세한 내용을 작성.
  헤더로 충분한 표현이 가능하다면 생략. (Blank Line으로 구분한 다수 문단 사용 가능)

name: my-repo



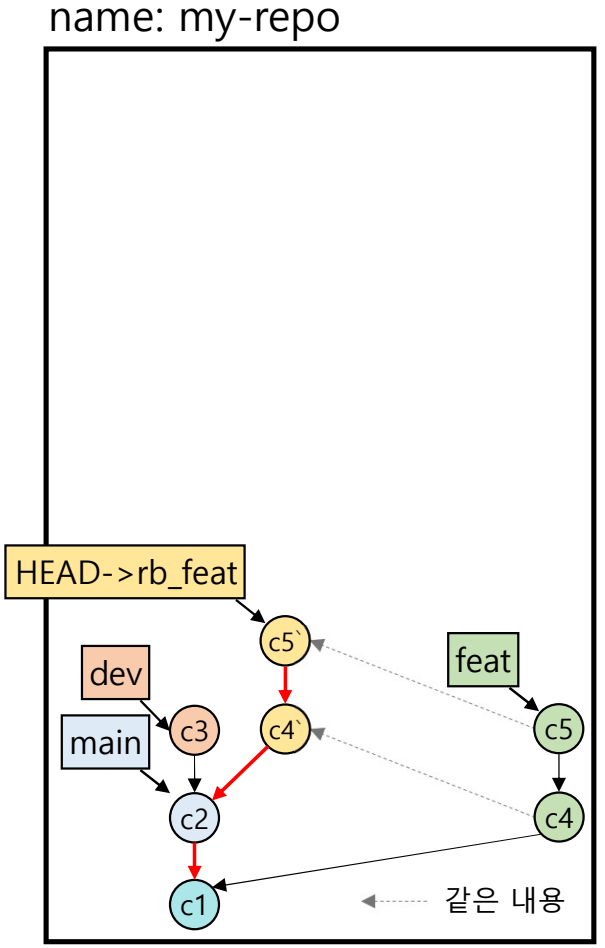
git rebase

■현재 branch의 베이스 commit을 변경

- ➢ $ git rebase <베이스>
  - 베이스로 할 commit을 <베이스>로 변경
- ➢ commit log가 선형으로 남음
- ➢ 두 commit이 서로 다른 기능인 경우에 매우 유용
  - Server와 Client를 각각 개발 후, main에 rebase

■ 예제

- ➢ (feat) $ git checkout -b rb_feat
- ➢ (rb_feat) $ git rebase main
- ➢ (rb_feat) $



git merge

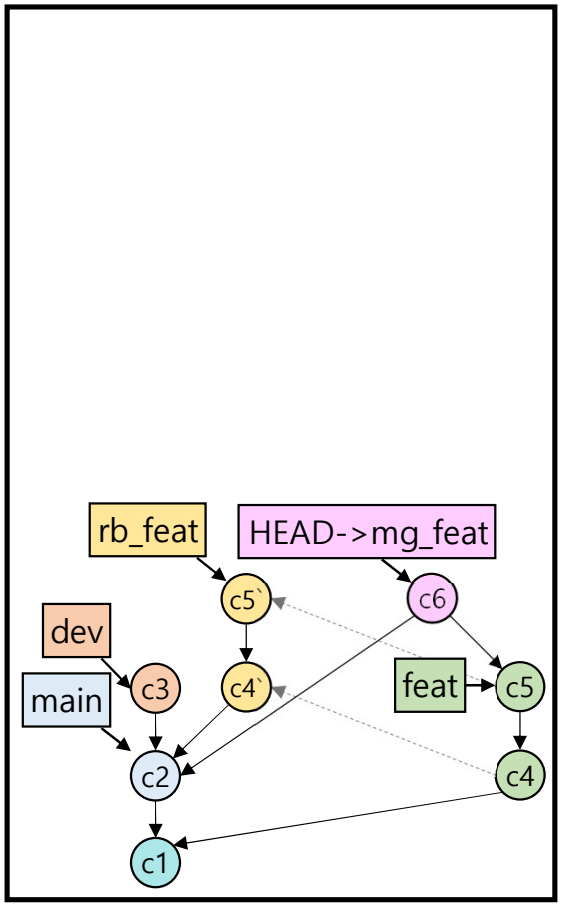
■현재 branch를 다른 branch와 병합 ➔ 3-way merge

- ➢ $ git merge <브랜치>
  - 현재 branch와 <브랜치>의 내용을 합친 새로운 commit 생성 ➔ Merge Commit

■ 예제

- ➢ (rb_feat) $ git checkout feat
- ➢ (feat) $ git checkout -b mg_feat
- ➢ (mg_feat) $ git merge main
- ➢ (mg_feat) $

name: my-repo



git merge - 2

■Fast-Forward: 두 branch가 조상-자손 관계

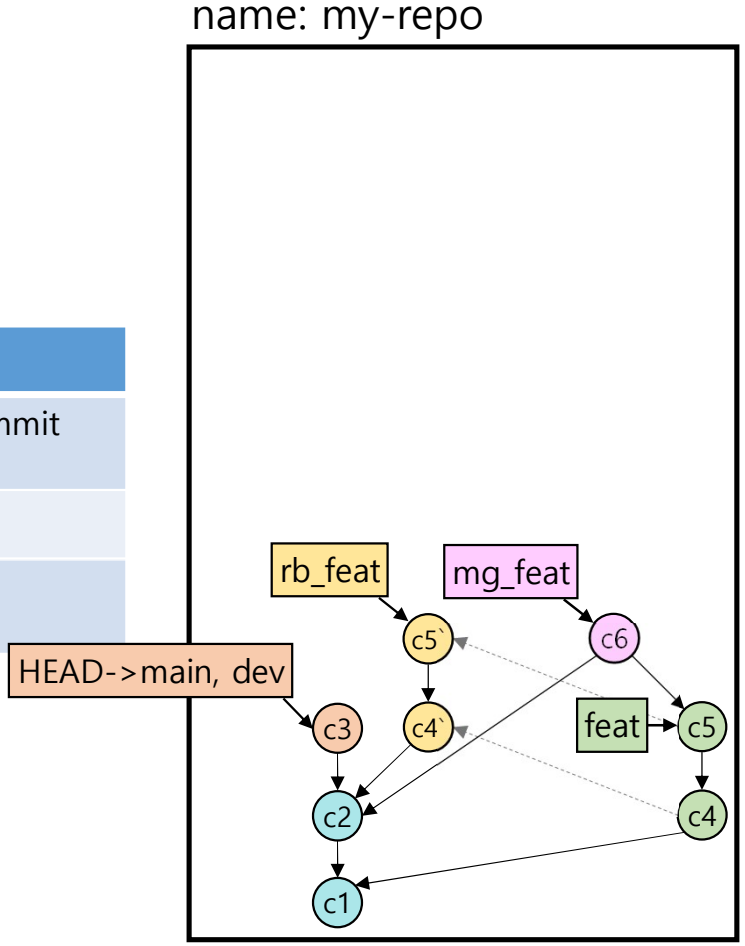
■과거 commit에 머무른 branch를 앞으로 이동 ➔ Fast-Forward

- $ git merge <브랜치> (--ff 생략)
- 현재 branch가 <브랜치>의 조상인 경우에 발동

옵션	설명
git merge [--ff] <브랜치>	<브랜치>가 자손인 경우 Fast-Forward. Without merge commit <브랜치>가 자손이 아닌 경우 병합. With merge commit
git merge --no-ff <브랜치>	Fast-Forward와 병합 모두 with merge commit
git merge --ff-only <브랜치>	병합 관계 시 동작하지 않음. Fast-Forward 경우만 동작. Without merge commit

■예제

- (mg_feat) $ git checkout main
- (main) $ git merge dev
- (main) $



git merge - 3

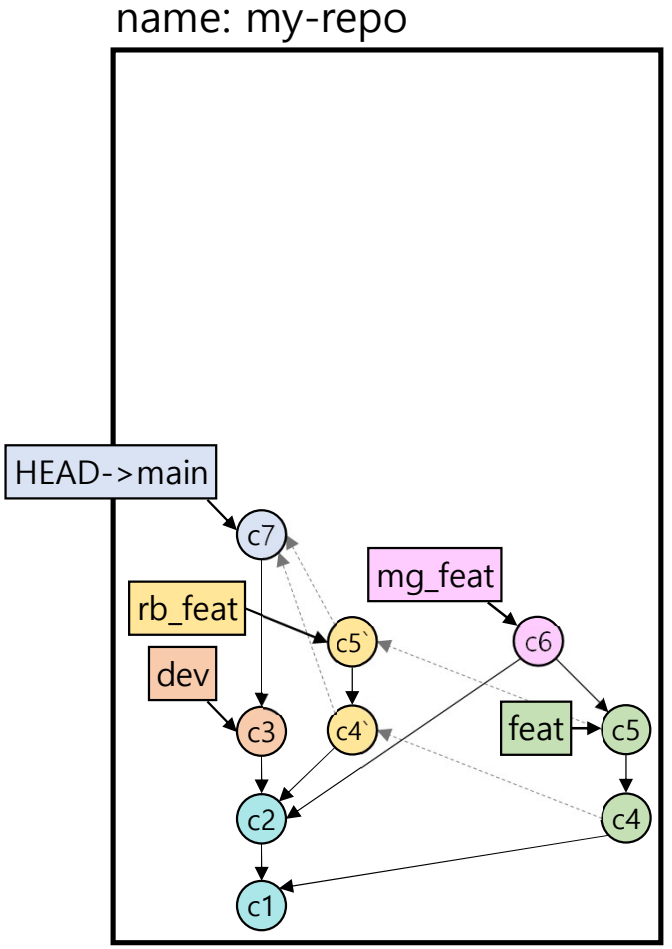
■Squash: 여러 commit을 하나로 병합

■다른 브랜치의 commit 내용을 압축

- $ git merge --squash <브랜치>
  - <브랜치>와 차이나는 commit을 하나로 병합
- rebase와 유사

■예제

- (main) $ git merge --squash rb_feat
- (main) $



git merge & rebase - pros. cons.

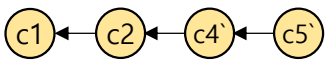
■Rebase 및 squash의 장단점

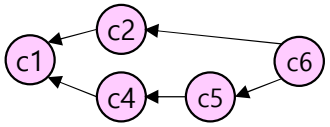
- ➢ Commit History를 변경
  - 예시에서 원본 *c4*, *c5*와 동일한 내용의 commit을 *c4`*, *c5`*로 새로 생성하게 됨
  - **Pros)** History를 알아보기 쉽게 변경 가능
  - **Cons)** History를 변경한다는 것 자체가 문제가 될 수 있음
- ➢ Remote Repository에 push한 경우에는 rebase를 사용하지 않아야 함
  - 위와 같은 이유로, 기존의 history를 가진 동료가 있는데, 내가 history를 변경한 경우, 양쪽의 history를 모두 갖게 될 수 있음 ➔ 혼란 유발, 나쁜 history 유발
- ➢ Local Repository에서 지저분한 commit을 압축하고, 가독성을 높일 때 사용

■Merge의 장단점

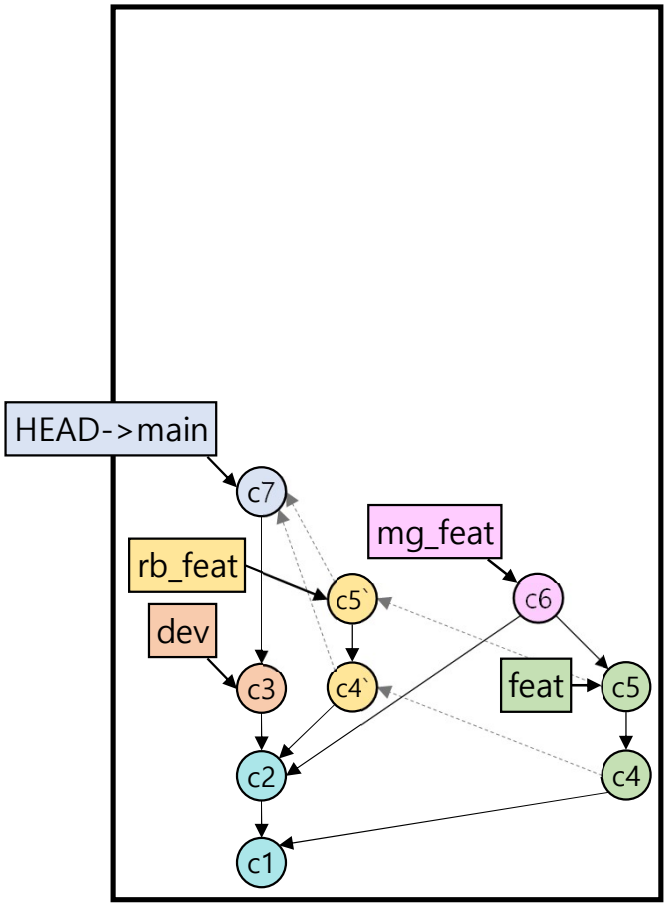
- ➢ Commit History를 정확하게 남길 수 있음
  - **Pros)** 수정 사항 기록을 정확히 추적가능
  - **Cons)** History가 매우 지저분해짐 ➔ 가독성이 떨어지고, commit 추적이 힘들어짐
- ➢ Remote Repository에 업로드된 commit을 병합하거나, 현재 commit history를 남기고 싶을 때 사용

※ 예시: *rb_feat*와 *mg_feat*은 완전히 동일한 내용임 (*c5`* = *c6* = *c2* + *c5*)

mg_feat history: 

rb_feat history: 

name: my-repo



git cherry-pick

■기본적으로 버전 관리에 사용하는 명령은 아님

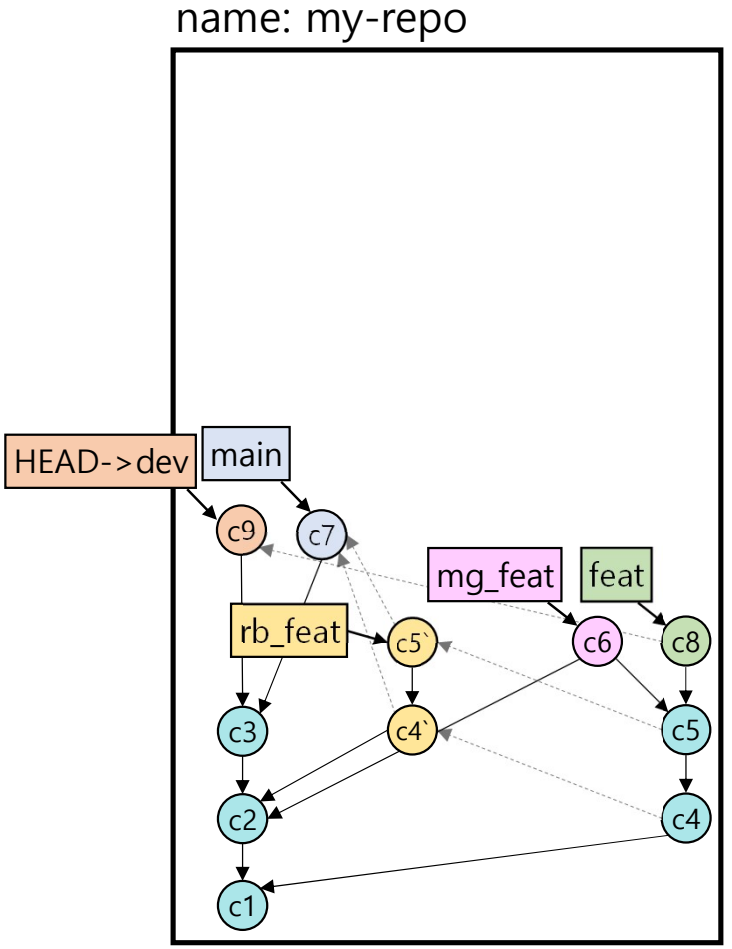
- 일부 commit만을 사용해야하는 경우 사용
- **(※ 주의)** 아래와 같은 특수 상황 이외에는 사용하지 말 것

■특정 commit만 병합

- $ git cherry-pick <커밋>
  - 특정 commit을 현재 branch와 병합
  - $ git cherry-pick <커밋1> <커밋2> <커밋3>: 여러 커밋 동시에 병합
  - $ git cherry-pick <커밋-처음>..<커밋-마지막>:
    <커밋-처음>부터 <커밋-마지막>까지 한번에 옮길 수 있음

■예제

- (main) $ git checkout feat
- (feat) $ 파일 생성 및 add/commit
- (feat) $ git checkout dev
- (dev) $ git cherry-pick c8
- 이를 통해, c4, c5에서 수정/생성된 파일은 제외하고,
  c8 수정/생성 사항만 가져올 수 있음



Merge, Rebase, Cherry-pick - Conflict

■Merge, Rebase, Cherry-pick 사용 시 Commit 간의 충돌

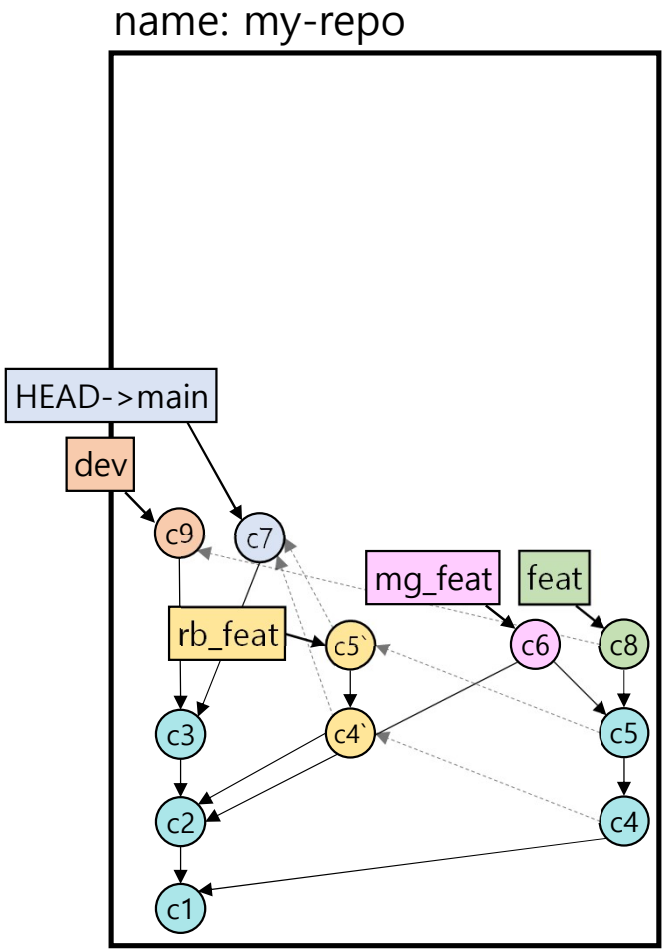
- 양쪽 commit에서 수정 라인이 겹치는 내용은 충돌할 수 있음
- working tree에서 modified/staged 상태의 파일에서 변경된 commit이 존재하면 merging 실패 ➔ commit 해야함

```
tot0ro@DESKTOP-JUNHO:~/my-repo$  (main) git merge dev
error: Your local changes to the following files would be overwritten by merge:
        README.md
Please commit your changes or stash them before you merge.
Aborting
```

- 양쪽 history에 변경된 부분이 겹치면 conflict 발생
  (아래 예시 명령 $ git st → $ git status)

```
tot0ro@DESKTOP-JUNHO:~/my-repo$  (main) git merge dev
Auto-merging README.md
CONFLICT (content): Merge conflict in README.md
Automatic merge failed; fix conflicts and then commit the result.
tot0ro@DESKTOP-JUNHO:~/my-repo$  (main|MERGING) git st
On branch main
You have unmerged paths.
  (fix conflicts and run "git commit")
  (use "git merge --abort" to abort the merge)

Changes to be committed:
        new file:   temp2.txt

Unmerged paths:
  (use "git add <file>..." to mark resolution)
        both modified:   README.md
```

Conflict된 파일



Merge, Rebase, Cherry-pick - Conflict - 2

■예제

- *main* branch의 *README.md* 파일과, *dev* branch의 *README.md* 파일을 수정 및 commit
- (main) $ git merge dev

■충돌 해결

```
<<<<<< HEAD
현재 commit 내용
=======
merge 대상 branch 혹은 commit 내용
>>>>>> 대상 branch 혹은 commit
```

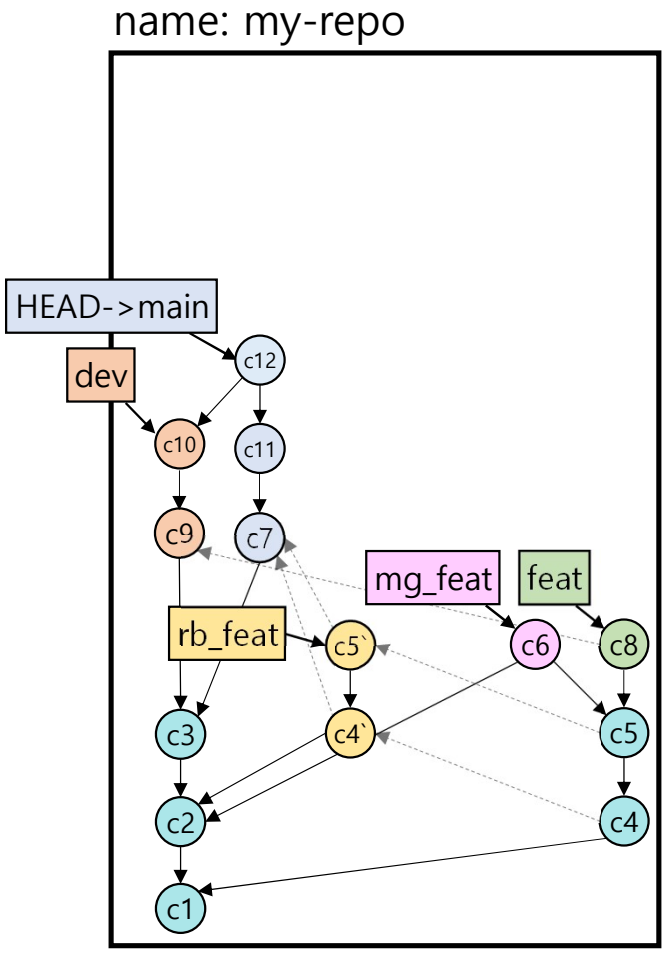
README.md

```
# Read Me

<<<<<<< HEAD
## Do it yourself
=======
## You can do it
>>>>>>> dev
```

- 지우고 싶은 내용, 남길 내용 자유롭게 수정 및 초록색 라인 삭제 후
- $ git add -p
- $ git <명령어> --continue

■명령 수행 중단

- $ git <명령어> --abort: <명령어>를 수행하기 전으로 돌아감

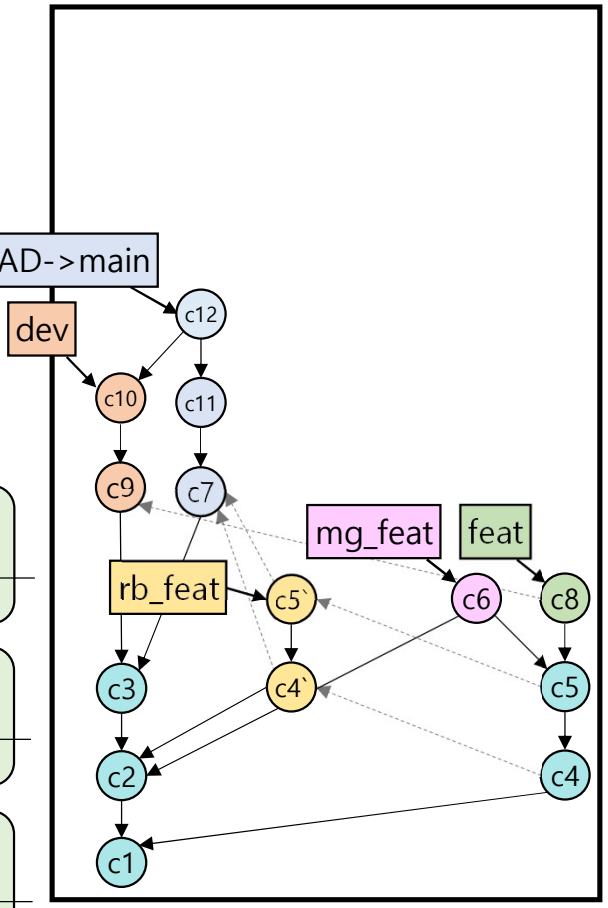


git reset - Workdir

■조상 Commit으로 되돌아가기

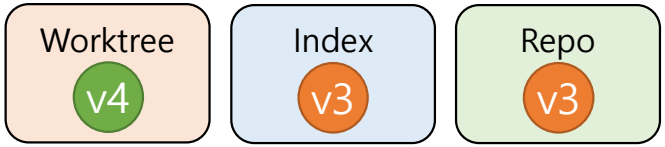
- $ git reset (--mixed 생략)
- Rebase와 마찬가지로 History를 변경하기에, Remote branch에 올라간 commit에 사용하지 말아야함

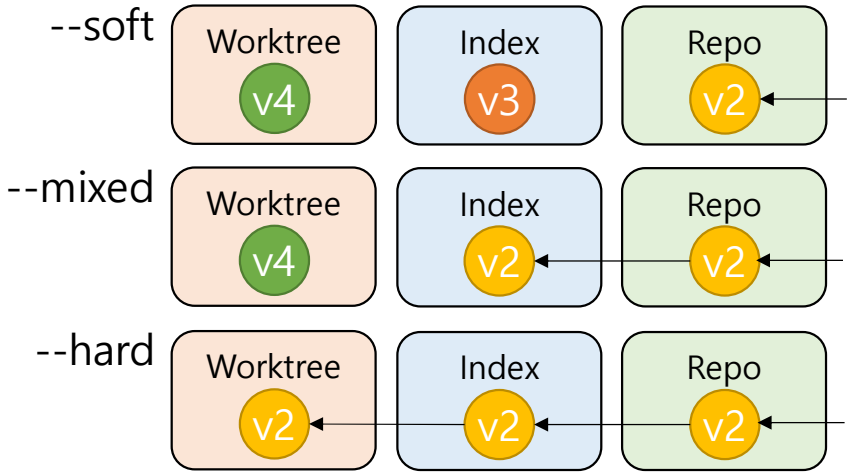
옵션	설명
git reset --soft <커밋>	현재 branch의 HEAD를 <커밋>으로 이동
git reset --mixed <커밋>	--soft 옵션에 더하여 Staging area를 비움
git reset --hard <커밋>	--mixed 옵션에 더하여 Working directory의 수정사항을 지움



■예시

- 가정 1: Repo에 v3를 commit하고 v4를 수정 중
- 가정 2: $ git reset <옵션> v2를 수행





git reset - 2 - Workdir

■예제 --hard

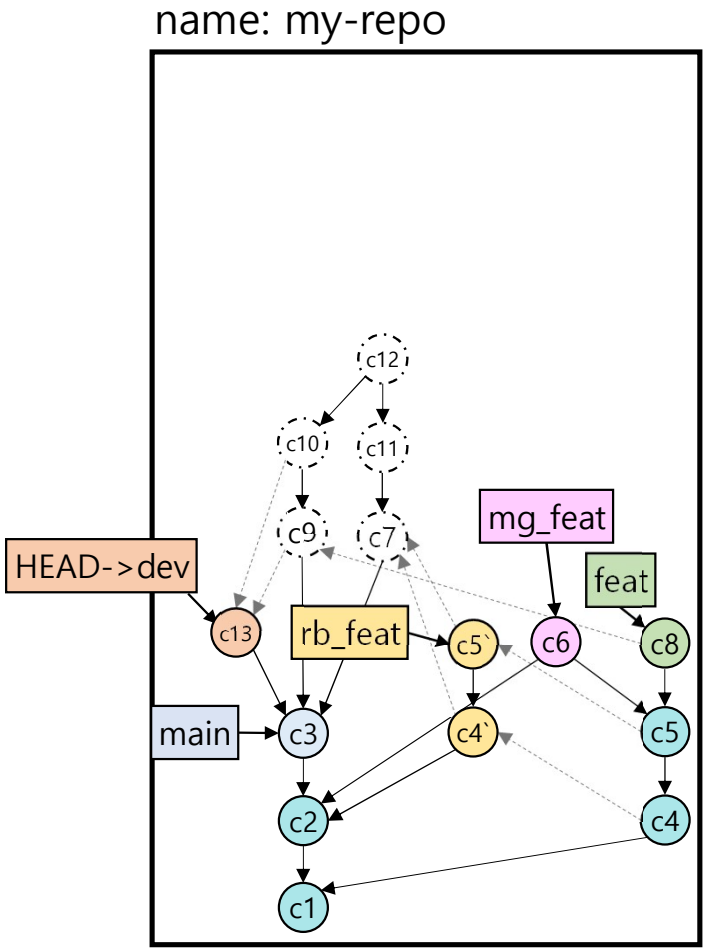
- (main) $ git reset --hard c3
  - 혹은 $ git reset --hard HEAD~3

■예제 --soft

- (main) $ git checkout dev
- (dev) $ git reset --soft main
  - 혹은 $ git reset --soft c3
  - 혹은 $ git reset HEAD~2 && git add -A
- (dev) $ git commit -m "reset and re-commit"
- 이런 식으로 최근 commit을 압축하는 데에 사용할 수 있음

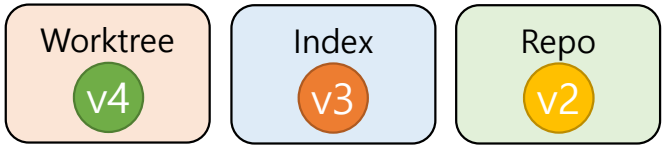
※ 점선 commit은 local repo에 남아 있지만,
가리킬 수 있는 branch가 없기 때문에 상실된 것으로 취급.

- UUID를 강제로 알아내서 복구할 수는 있음
- 추후 reflog 챕터에서 복구하는 방법을 알아볼 예정



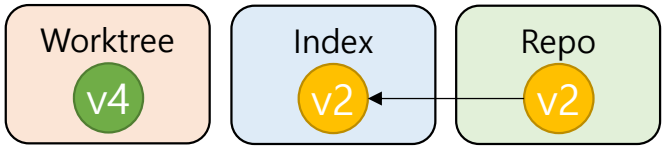
git reset - 3 - files

현 상태

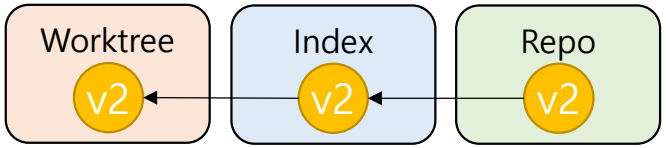


■Staging Area에서 수정된 내용을 내리고 싶을 때

- $ git reset <PATH>
  - PATH 경로의 파일을 staging area에서 내림 (add와 반대 동작)

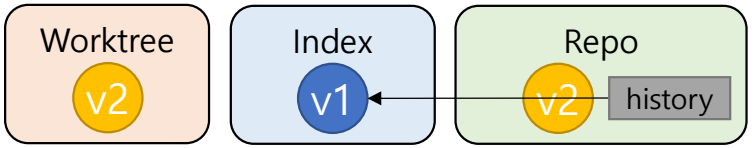


- $ git checkout <PATH>
  - 위 reset 동작에 더하여 working tree까지 바꿈

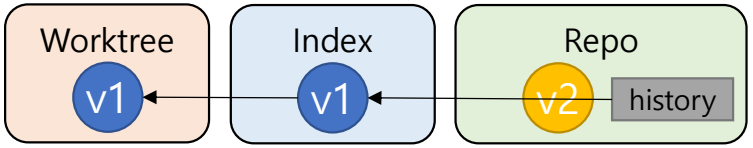


■특정 commit 버전으로 변경

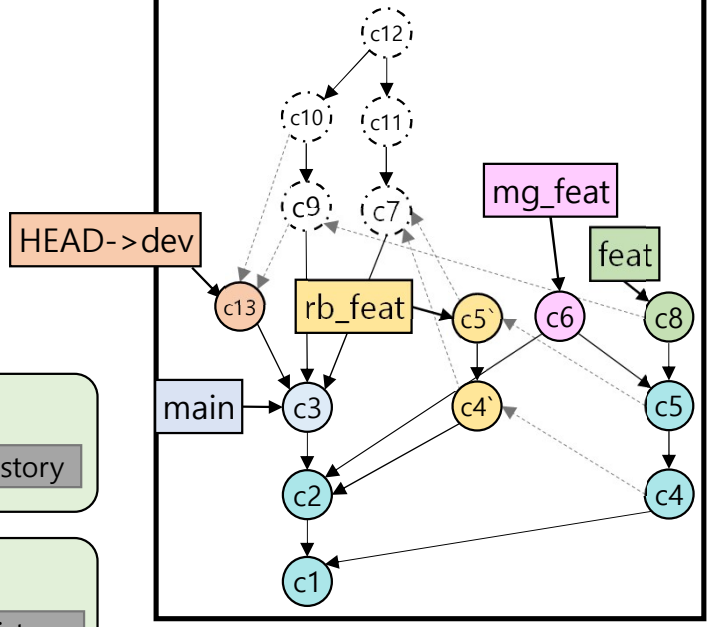
- $ git reset <커밋> -- <PATH>



- $ git checkout <커밋> -- <PATH>



name: my-repo



git reset - 4 - files

■Staging Area에서 수정된 내용을 내리고 싶을 때 - 2

- ➢ $ git reset -p
  - git add -p 처럼 hunk 단위로 un-staged 시킬 수 있음

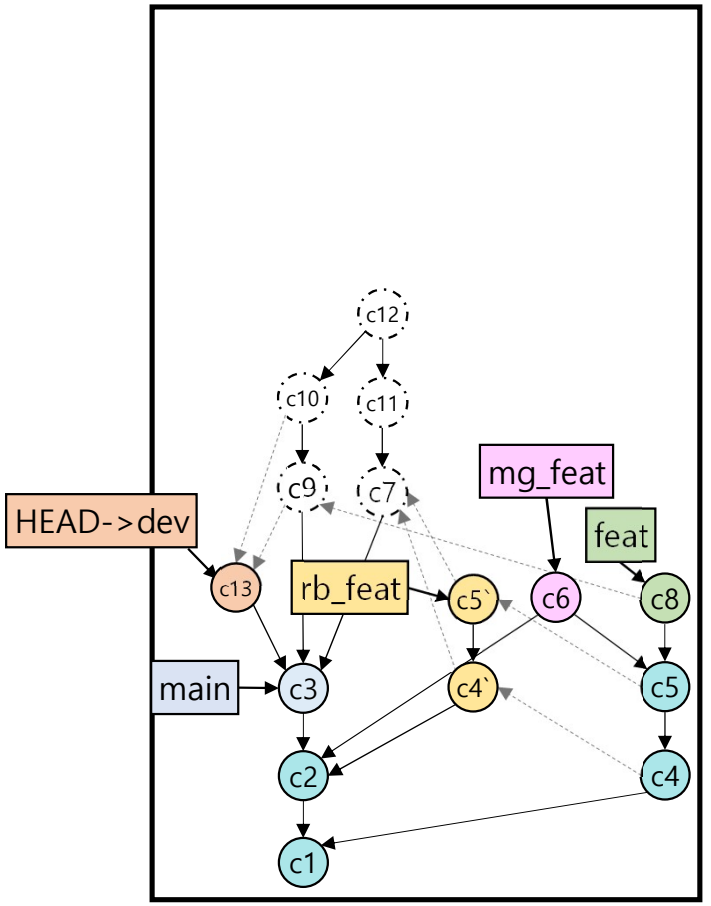
```
tot0ro@DESKTOP-JUNHO:~/my-repo$  (dev) git reset -p
diff --git a/README.md b/README.md
index 6996019..1194175 100644
--- a/README.md
+++ b/README.md
@@ -1,3 +1,2 @@
 # Read Me

-## You can do it
(1/1) Unstage this hunk [y,n,q,a,d,e,?]?
```

- ➢ $ git checkout -p: 마찬가지로 가능

■Git 2.23 버전 이후부터 명령어가 변경됨

- ➢ git reset <PATH> → git restore --staged <PATH>
- ➢ git checkout <PATH> → git restore <PATH>

name: my-repo



git revert

■커밋 되돌리기

- ➢ $ git revert <커밋>
  - 해당 <커밋>의 변경 사항 취소하는 commit 생성
- ➢ Conflict 가능성 존재

■예제

- ➢ (dev) $ git revert c2

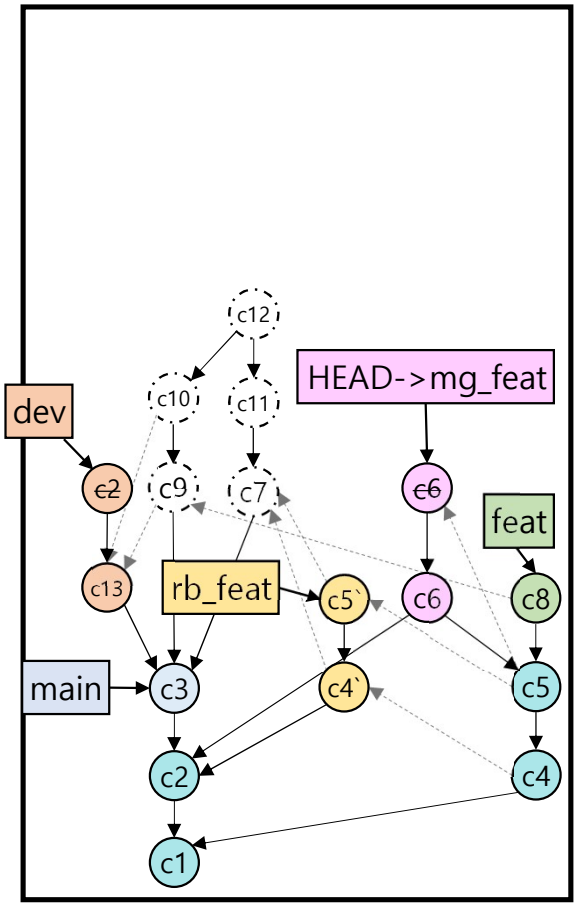
■병합 되돌리기

- ➢ $ git revert -m <부모 번호> <병합 커밋>
  - <병합 커밋>을 <부모 번호> 쪽으로 취소
- ➢ $ git show <커밋>
  - <커밋>의 내용을 볼 수 있음
  - 부모 순서대로 순번이 매겨짐

■예제

- ➢ (dev) $ git checkout mg_feat
- ➢ (mg_feat) $ git revert -m 1 HEAD → 내용은 c5`
- ➢ c6은 c5에서 c2를 merge했던 것 이기 때문에 일반적으로 c2가 revert됨

name: my-repo



git rebase -i

■커밋 interactive rebase

- $ git rebase -i <커밋>
  - HEAD부터 <커밋> **'전'**까지의 commit 변경

■예시

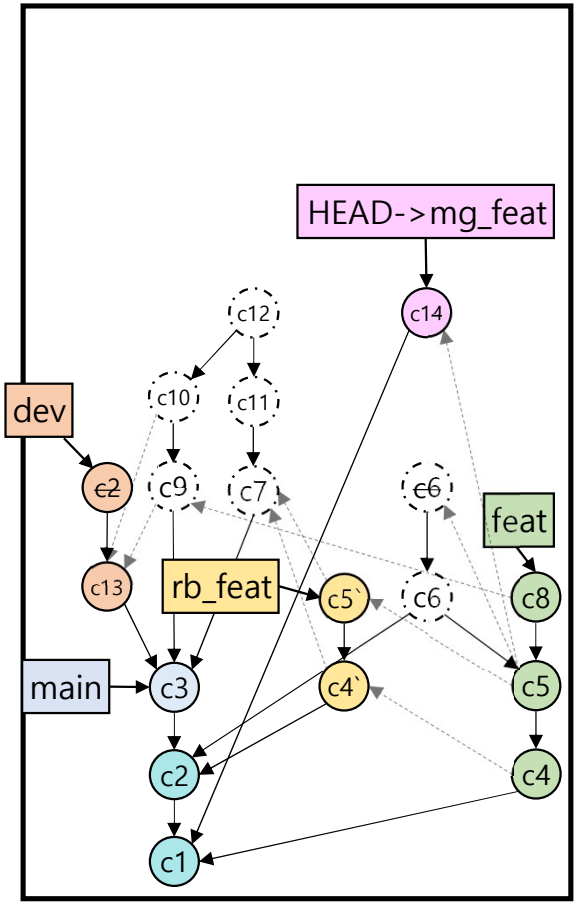
- (mg_feat) $ git rebase -i HEAD^^^^

```
pick 412a649 c2
drop 5d297fd c4
squash a813f39 c5
fixup 29d1298 Revert "Merge commit 'c2+c5' into mg_feat"

# Rebase 0056e73..29d1298 onto 29d1298 (4 commands)
#
# Commands:
# p, pick <commit> = use commit
# r, reword <commit> = use commit, but edit the commit message
# e, edit <commit> = use commit, but stop for amending
# s, squash <commit> = use commit, but meld into previous commit
# f, fixup <commit> = like "squash", but discard this commit's log message
# x, exec <command> = run command (the rest of the line) using shell
# b, break = stop here (continue rebase later with 'git rebase --continue')
# d, drop <commit> = remove commit
# l, label <label> = label current HEAD with a name
# t, reset <label> = reset HEAD to a label
# m, merge [-C <commit> | -c <commit>] <label> [# <oneline>]
# .       create a merge commit using the original merge commit's
# .       message (or the oneline, if no original merge commit was
# .       specified). Use -c <commit> to reword the commit message.
#
# These lines can be re-ordered; they are executed from top to bottom.
```

- pick 부분을 주석에 써 있는 Commands 중 하나로 변경하면 해당 방식으로 동작
- commit 순서 변경 가능

- **edit:** commit을 수정
- **pick:** commit 그대로 사용
- **drop:** commit 사용 안함
- **squash:** commit을 상위 commit에 병합
- **fixup:** squash와 같지만, commit messege는 drop

name: my-repo



Rebase 수행 결과

■결과

```
tot0ro@DESKTOP-JUNHO:~/my-repo$  (main) git lg --all
* 571b465 - (36 minutes ago) c14 - JHLEE (mg_feat)
| * 4958d62 - (30 minutes ago) c5 - JHLEE (rb_feat)
| * 7fd2e5c - (30 minutes ago) c4 - JHLEE
| | * ea5b272 - (30 minutes ago) c8 - JHLEE (feat)
| | * 31f5f80 - (30 minutes ago) c5 - JHLEE
| | * f1e509f - (30 minutes ago) c4 - JHLEE
| |/
|/|
| | * cd308da - (32 minutes ago) Revert "c2" - JHLEE (dev)
| | * f79357c - (32 minutes ago) c13 - JHLEE
| | * 227aa52 - (35 minutes ago) c3 - JHLEE (HEAD -> main)
| |/
| * 412a649 - (36 minutes ago) c2 - JHLEE
|/
* 0056e73 - (36 minutes ago) c1 - JHLEE
```

- 점선 commit은 실질적으로 참조할 수 없음. 때문에 제거된 것과 동일.
  - 하지만 지워진 것은 아니므로, commit HASH 값을 알아낼 수 있다면 복구 가능
  - ex) $ git checkout c12