# 목차

## 1.amend

## 2.cherry-pick

## 3.reset

## 4.reflog

## 5.revert

## 6.stash

## 7.tag

## 8.rebase

## =============

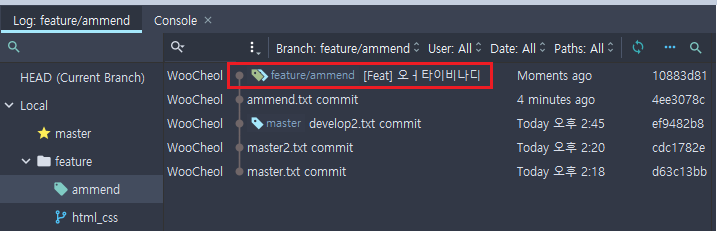
## 1.amend

현재 속해있는 브랜치 상에 있는 최종 커밋을 취소하고 새로운 내용 혹은 파일을 추가하거나 커밋 메시지를 수정할 수 있는명령어

### **간단 실습**

#### **[원격에 push 되지 않은 상태일때]**

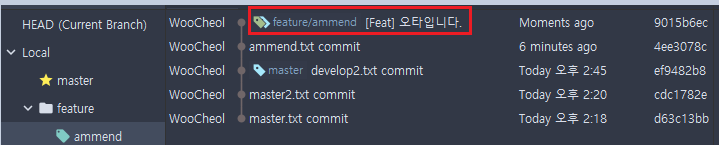
##### **[커밋 메시지를 잘못 입력 했거나 커밋 메시지 규약을 지키지 못했을때]**

****

amend명령어를 사용하여 가장 최근의 커밋 메시지 수정

****

정상적으로 수정확인

****

##### **[실수로 파일 혹은 코드를 빠뜨린 상태로 커밋했을 경우, 파일을 추가하며 기존 커밋 갱신하고싶을때]**

- 원하는 작업(파일 추가 혹은 수정 등)을 한뒤 git add 명령어를 입력한 다음,

- ‘git commit –amend –no-edit’을 한다.

- ‘--no-edit’를 붙이면 기존 커밋 메세지를 그대로 사용한다.

- 메시지까지 수정하려면 git commit –amend -m “메시지” 명령어를 사용한다.

- ‘&&’는 파이프라인으로 명령어를 연속적으로 실행하고 싶을때 사용

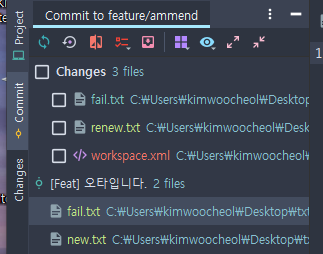
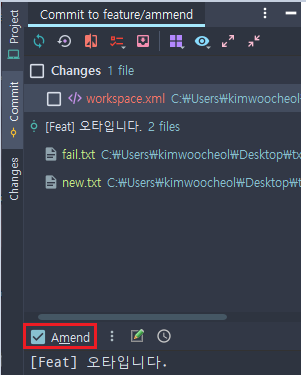


※ 인텔리 제이에서 하는 방법

- 왼쪽 메뉴의 commit -> Amend 체크박스를 클릭하면 가장 최근의 커밋 이력과 메시지가 뜬다.

- 커밋 메시지만을 수정하려면 메시지 수정후 commit

- 특정 파일들의 내용을 수정하거나 파일이 존재하면 add 후 commit

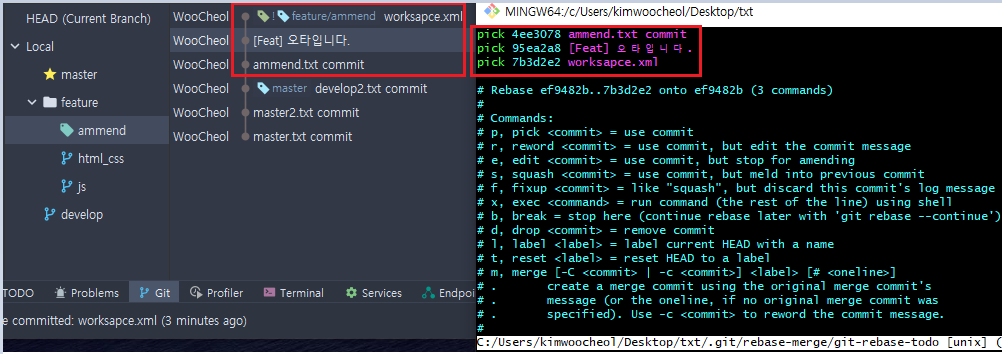
****

#### **[과거 커밋을 바꾸고 싶을때, git bash 방법]**

**git rebase -i HEAD~3** 명령어를 사용하면 현재 위치한 브랜치의 가장 최근 커밋 3개를 보여줌

(혹은 git reabse -i 3e5a9ac9처럼 커밋 해시값을 입력하면 해당 커밋 해시값 이후부터 현재(HEAD)까지 커밋 내역이 표시)

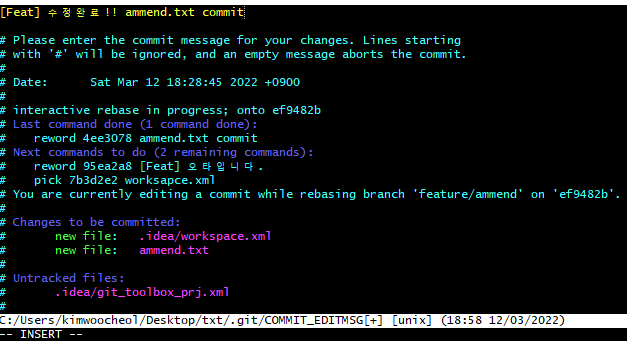
입력후 하단 오른쪽의 사진처럼 vi로 뜸(git bash에서는 맨 아래가 가장 최신)

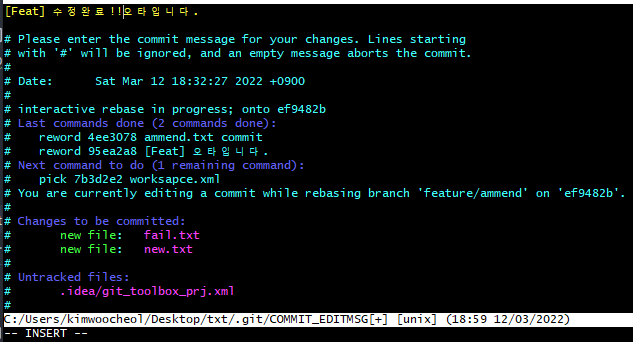
****

변경하고 싶은 커밋이력의 pick이란 단어를 reward로 변경후 esc -> :wq 명령어로 저장

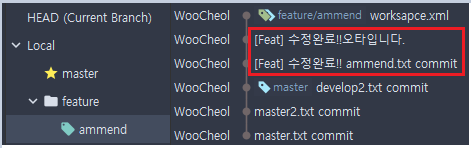
****

reward로 바꾼 커밋 이력을 수정할 수 있는 창이 순서대로 띄어진다. 원하는대로 커밋을 수정하고, esc -> :wq를 통해 저장

****

****

깃 히스토리를 보면 정상적으로 수정된걸 확인 가능

****

#### **주의점**

**수정하려는 커밋이 원격에 올라가지 않은 것에 해당해서만 사용하는궐 권장**한다. 바로 아래서 설명하겠지만 만약 원격에 올라가 있는 과거 commit까지 수정하려면 –-force 옵션을 통해 강제로 푸시해야 하는데 위험이 크다.(근데 혼자 쓰는 브랜치면 괜찮다)

※ 커밋메시지뿐만 아니라 해당 커밋의 파일의 내용을 수정하거나 파일 추가같은 작업까지 하려면 reward 대신 edit으로 바꿔주고 엔터를 친다.

****

그 후 수정모드로 들어가게 되는데 파일들의 내용수정, 파일 추가,삭제등 지지고 볶은다음

git add를 한 뒤, git rebase -continue명령어를 입력하면 commit 창이 뜬다. 커밋 메시지까지 변경후 esc -> :wq로 저장하면 끝

**※ –amend로 commit시 author 수정 옵션**

|  |
| --- |
| **git commit --amend --author="woocheol-kim <jikimee64@naver.com>"** |

#### **[원격에 push 된 상태일때]**

직전의 ammend 명령어로 로컬 commit 이력을 수정했다면 force를 통해 수정된 커밋을 강제로 push 해야 한다.

|  |
| --- |
| git push --force 브랜치 이름 or git push 브랜치 이름 -f |

#### **주의점**

**하지만 이 방식은 최대한 사용하지 않아야 한다.** 기존의 push 된 커밋의 로그를 갖고 있던 다른 팀원들이 로그를 수동으로 수정해줘야 하기 때문이다. 잘못하다간 기존의 commit 이력이 날라갈 수도 있다.(근데 혼자 쓰는 브랜치면 괜찮다.)

### **활용 사례**

#### **[커밋이력의 메시지 혹은 빠뜨린 내용을 추가하고싶을때]**

- 가장최근 커밋 혹은 과거 커밋포함

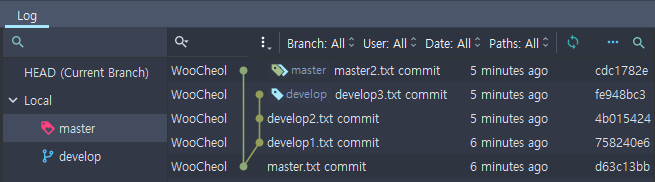
## 2.cherry-pick

git cherry-pick 명령어는 특정한 커밋이력을 정해서 현재 HEAD가 가리키는 branch에 추가하는 개념이다.

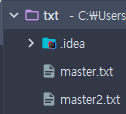
쉽게 말해서 **다른 branch에 있는 여러개의 commit들 중, 원하는 commit을** 현재 나의 branch에 가져와서 commit 할 수 있는 것이다.

### **간단 실습**

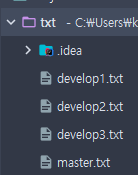
데이터 사전 세팅



master 브랜치에는 master.txt, master2.txt 각각 add 후 커밋



develop 브랜치는 “master.txt commit” 커밋을 기준으로 파생된 브랜치 develop1.txt, develop2.txt, develop3.txt 각각 add 후 커밋



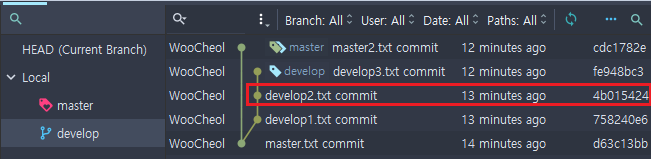
master 브랜치에서 develop 브랜치의 커밋 이력중 “develop2.txt commit” 이력만 가져오고 싶다면?

최종적으로 원하는 파일 목록은 아래와 같을 것이다. (.idea 제외)

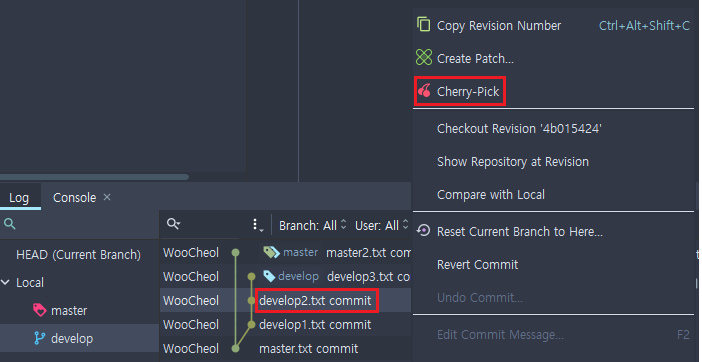
- master.txt

- master2.txt

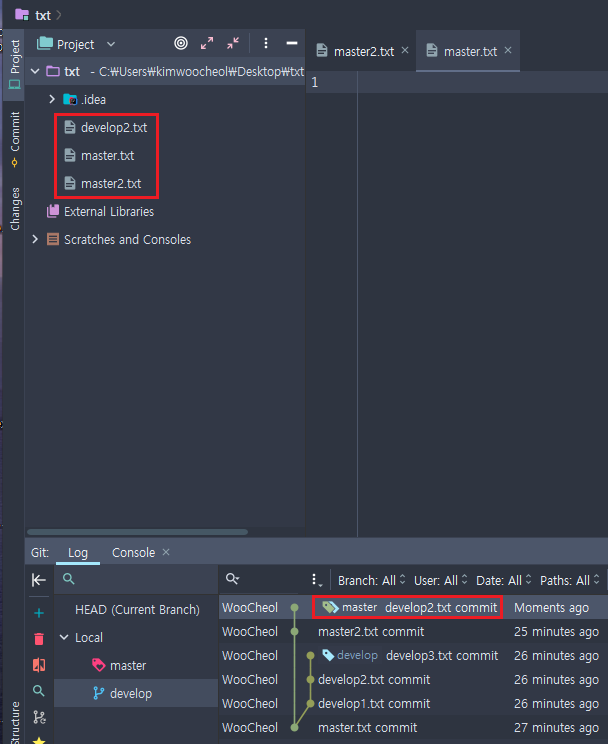
- develop2.txt



원하는 커밋 이력에서 오른쪽 마우스 클릭 -> Cherry-Pick를 클릭한다



아래와 같이 “develop2.txt commit” 이력만 master 브랜치에 추가되었고 파일도 원하는대로 추가된걸 확인할 수 있다.



※ 명령어

|  |
| --- |
| git cherry-pick {commit 해쉬명1}  git cherry-pick {commit 해쉬명1} {commit 해쉬명2} //2개 이상 선택  //해쉬1과 해쉬3커밋 사이의 모든 커밋  //해쉬1이 가장 최근 커밋, 해쉬1은 적용되지 않음  git cherry-pick {commit 해쉬명1}..{commit 해쉬명3} |

### **활용 사례**

#### **[프로젝트 협업]**

1. 하나의 서비스를 제공하기 위해선 프론트 기능과 디자인이 모두 필요하다고 가정

프론트 개발자 : 디자이너님 JS 개발은 다 완료 됬는데 html이랑 css 개발 완료 되셨을까요?

디자이너 : html은 다되서 커밋했구요. css는 지금 커밋 하면서 작업 중입니다.

프론트 개발자 : 그럼 html 관련된 커밋 부분만 cherry-pick으로 가져와서 적용해볼께요.

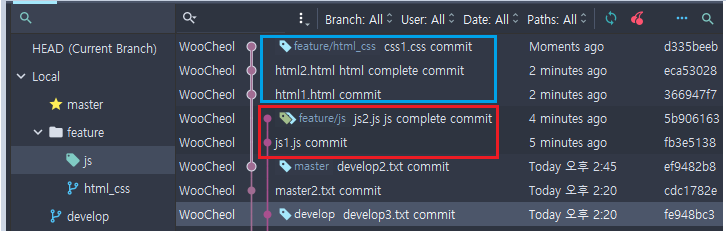
디자이너 : 네~

프론트 개발자는 js1.js js2.js를 개발한 후에 add, commit한 상태

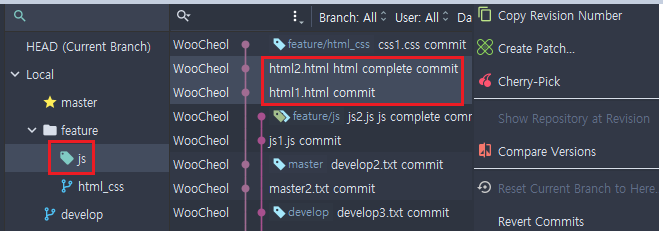
디자이너는 html1, html2, css1을 개발한 후에 차례대로 add, commit 한

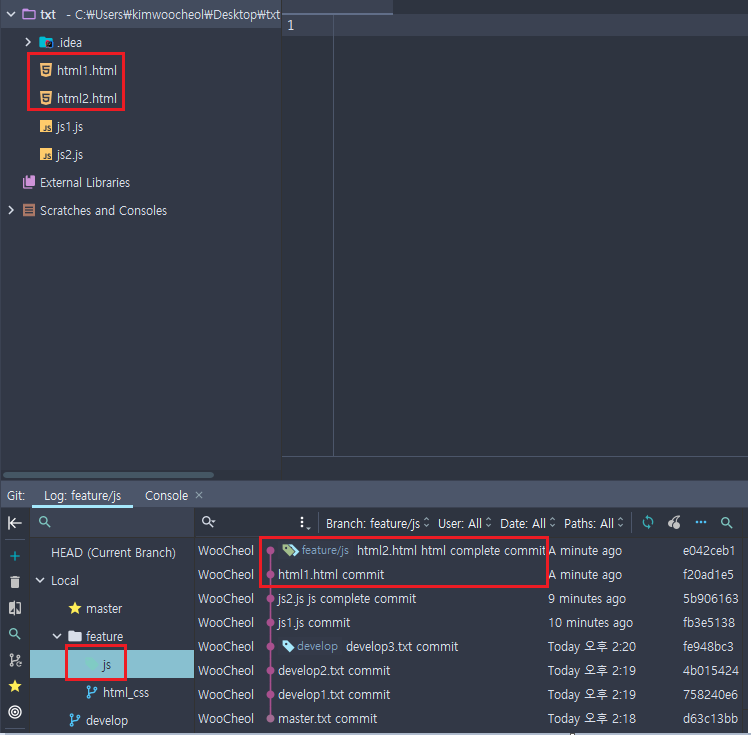
프론트 개발자는 디자이너가 개발한 html1, html2 만을 반영하고 싶은 상황

( 빨간색 박스 : 프론트 개발자 커밋이력, 파랑색 박스 : 디자이너 커밋 이력)



체리픽으로 선택할 커밋이력을 다중 선택(html1, html2) 후 오른쪽 마우스 -> Cherry-Pick 클릭



커밋 이력이 합쳐지면서 html1, html2를 정상적으로 가져온걸 확인 가능

#### **[버그수정]**

A 기능 개발이 완료되서 master 브랜치에 병합 후 배포한 상황

B기능을 개발하던 도중, A기능에서 갑자기 버그가 발견되서 빠르게 버그패치를 해야하는 상황

B기능과 관련된 커밋이력이 존재하는 상황이고, B 기능은 개발중이라 merge 하기는 힘든 상태

A기능에 대한 버그픽스 후 add commit한 후에 해당 커밋이력만 master에서 cherry-pick으로 가져와서 반영하면 됨

#### **[잘못된 브랜치에서 개발]**

feature/a 브랜치에서 새로운 기능을 개발해 커밋을 생성 했는데, 알고보니 feature/b 브랜치에서 작업을 했었어야 하는 상황

feature/b 브랜치로 이동한 후에, feature/a 브랜치에 커밋된 내용을 cherry-pick으로 가져오면 끝

#### **[반영되지 않은 pr]**

실수로 PR을 merge 하기 전에 close 했다면, cherry-pick을 통해 해당 commit을 다시 가져옴으로써 살릴 수 있다.

#### **[요구사항이 바뀌어 필요없는 커밋이 생겼을 경우]**

필요한 커밋들만 cherry-pick으로 가져오기

#### **[stage에 반영된 커밋내역들중 일부 내역들만 master에 배포하려고 할때]**

stage에 A서비스와 관련된 커밋과 B서비스와 관련된 커밋들이 합쳐져 있는 상황

master에는 A서비스만 반영을 해야될때 A서비스와 관련된 커밋들만 master에서 cherry-pick으로 가져와서 반영함하면 끝

#### 주의점

cherry-pick은 유용한 도구이지만, 남용하면 좋지 않다.

전통적인 merge 혹은 rebase 방법을 우선순위로 생각하고, 차선책으로 cherry-pick을 사용하자

### **예상치 못한 상황 대처방안**

#### **[충돌 발생]**

1. 충돌 해결

2. add (커밋은 다시 할 필요 없다.)

3 . git cherry-pick --continue 명령어를 통해 다시 체리픽을 진행

|  |
| --- |
| git add [path] git cherry-pick --continue |

#### **[중단(롤백)]**

git cherry-pick –abort 명령어로 체리픽을 중단하면 체리픽을 하기전 상태로 되돌아감

|  |
| --- |
| git cherry-pick --abort |

## 3.reset

커밋 이력을 과거 커밋으로 되돌아가는 명령어, 되돌아가려는 과거 이후의 커밋이력을 모두 지운다.

원격에 올리지 않고 로컬에만 커밋이 머무른 상태고, 혼자 사용하는 브랜치라면 reset를 써도 괜찮다.

#### 주의점

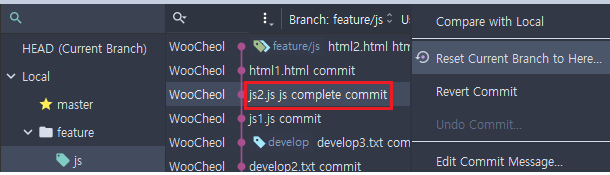
**하지만 원격에 올라간 커밋까지 포함해서 reset 명령어를 쓰는건 권장하지 않는 방법**

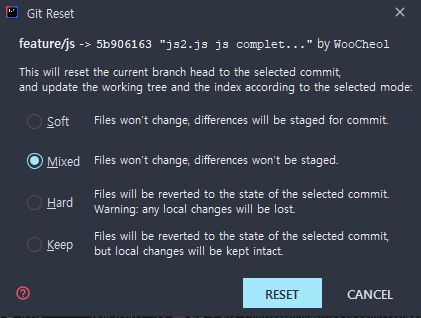
원격에 push한 과거의 이력이 날라가거나 다른 유저와 충돌이 일어날 수 있기 때문이다.

(하지만 혼자 쓰는 브랜치면 괜찮다)

### **간단실습**

html 관련 커밋이력을 지우고 “js2.js js complete commit” 시점으로 되돌아가고 싶을때 해당 커밋의 오른쪽 마우스 -> Reset Current Branch to here 클릭





Soft : 변경 이력은 모두 삭제하지만 변경 내용은 stage 상태로 남아 있다. add 명령어 없이 바로 commit 진행이 가능하다.

Mixed : 변경 이력은 모두 삭제하지만 변경 내용은 unStage 상태로 남아 있다. 원하는 코드를 다시 반영하려면 add -> commit해주면 된다.

Hard : 해당 시점이후의 커밋이력과 파일을 모두 지운다.

※ 명령어

git reset –-soft[hard,mixed] HEAD^ : 가장 최신 커밋 1개 삭제

* mixed가 default 값

git reset –-soft HEAD^& : 가장 최신 커밋 2개 삭제

* 꺽쇠 갯수에 따라 최신 커밋을 필요한만큼 순서대로 삭제

git reset –-soft HEAD~n : 가장 최신 커밋 n개 삭제

git reset –-soft HEAD^

#### **주의점**

**혼자 사용하는 저장소가 아니면 reset은 쓰지말자**

왜? 원격 저장소의 커밋 히스토리와 내가 날려버린 커밋 히스토리가 불일치 하기 때문에 다른 사람이 저장소에 push가 안될수도 있고 커밋이 꼬여버린다.

(혼자 쓰는 브랜치면 강제로 push –force 하면 된다.)

### **활용사례**

**※ 혼자 브랜치를 사용한다는 가정**

#### **[커밋이력을 지우고 싶을 경우]**

## 4.reflog

git으로 수행한 모든 이력을 볼수 있는 명령어

git log에 나오지 않는 이력으로 되돌아가고 싶을때 유용

가령 git rest -hard삭제되었던 커밋이력까지 다 보여주기 때문에, 복구 가능

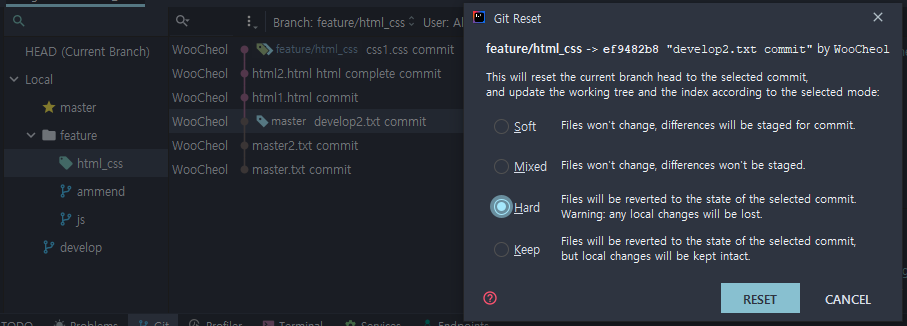
삭제된 브랜치까지도 복구 가능하다.

### **간단실습**

#### **[삭제된 로컬 커밋이력 및 파일 복구]**

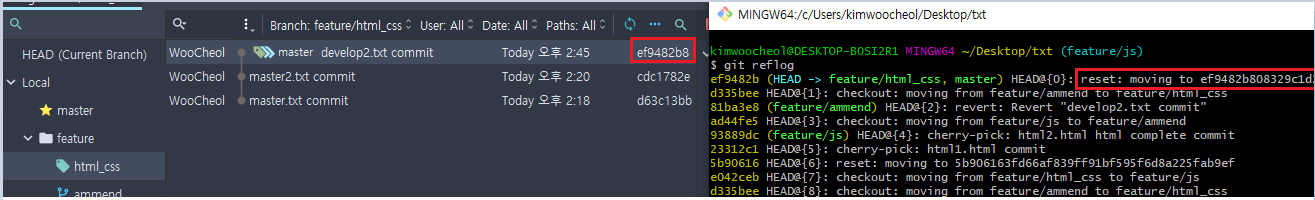
develop2.txt commit으로 reset –hard

그러면 html1, html2, css 작업한 커밋이 날라간다.



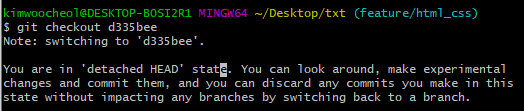
오른쪽 그림에서 맨위가 가장 최신 git 이력, git 이력 자체도 해쉬값이 존재

원하는 건 reset –hard 하기 바로 직전 이력인 d335bee로 복원하고 싶은 것



해당 지점으로 체크아웃 (detached-HEAD 상태)

|  |
| --- |
| git checkout d335bee |



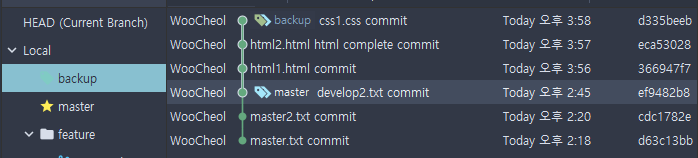
새로운 브랜치 생성(이름은 자유)

|  |
| --- |
| git branch backup |

민든 브랜치로 checkout

|  |
| --- |
| git checkout backup |

확인결과 날려진 이전 커밋 내역들이 복구된걸확인 가능

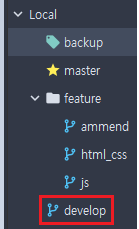


브랜치를 따로 만들지 않고 아래 명령어를 쓰면 현재 브랜치에서 바로 복구할 수 있다.

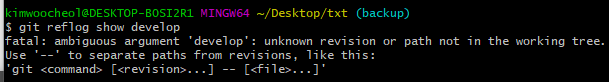
git reset –hard {commit 해쉬명}

#### **[삭제된 로컬 브랜치 복구]**

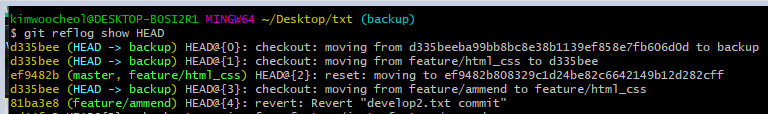
develop 브랜치를 삭제한다



삭제된 브랜치의 reflog 기록은 볼 수 없다.

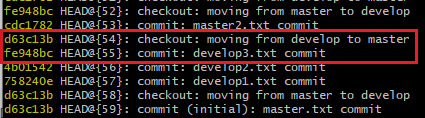


git reflog show HEAD 명령어로 HEAD의 변경이력을 볼 수 있다.

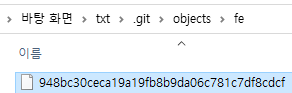


//이력 더 많이 있는데 생략..

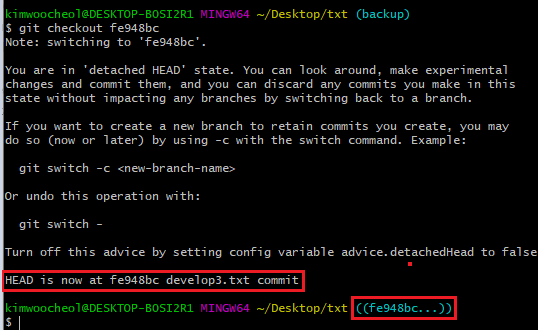
아래 빨간 박스를 유추해보니 develop에서는 f3948bc 이력인 해당 커밋이 develop에서의 마지막 커밋인 것으로 유추된다. 해당 해쉬값으로 복구 해보자



f3948bc 는 .git/objects 폴더에 저장되어 있기 때문에 언제든지 체크아웃하여 복구 가능하다.

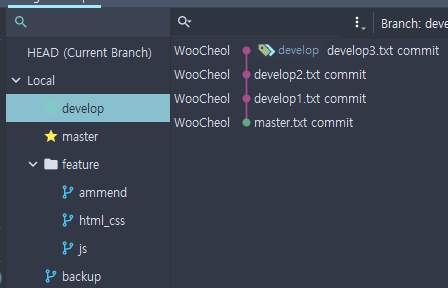


checkout하면 develop의 마지막 커밋이 복구된다.



새로 만든 develop 브랜치로 전향 하는 switch -c 명령어를 통해 브랜치 복구가 가능하다

|  |
| --- |
| git switch -c develop |



### **주의점**

**원격 브랜치 삭제는 복구가 안된다**

**브랜치가 삭제된 후 일정 이상의 시간이 지났을 경우 복구가 불가능하다**

### **활용사례**

#### **[삭제된 로컬 커밋이력 및 파일 복구]**

#### **[삭제된 로컬 브랜치 복구]**

#### **[git 이력 추적]**

## 5.revert

다른 사람과 공유하는 브랜치면 reset말고 revert를 쓰자

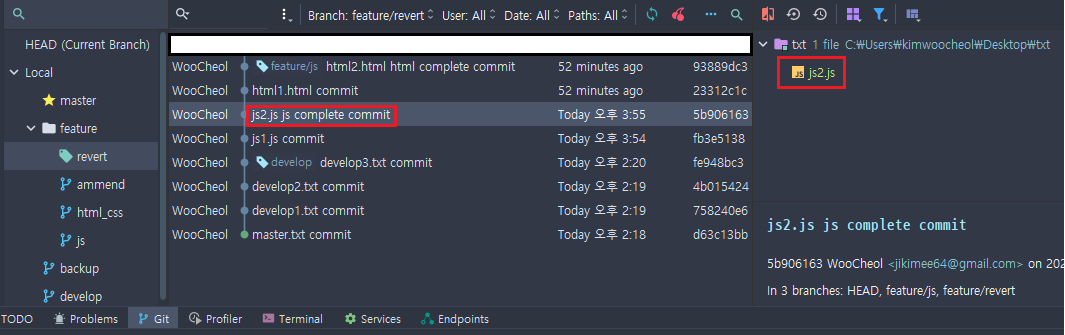
revert는 과거로 돌아가겠다는 **이력을 남겨두고** 원하는 시점으로 돌아간다.

잘못된 이력을 삭제하는게 아니라 원래대로 커밋 내용을 되돌리고 커밋 이력이 추가된다.

### **간단실습**

#### **[과거 하나의 커밋이력만 취소]**

우리는 “js2.js js complete commit” 커밋내용인 js2.js 파일을 취소하고 싶다.

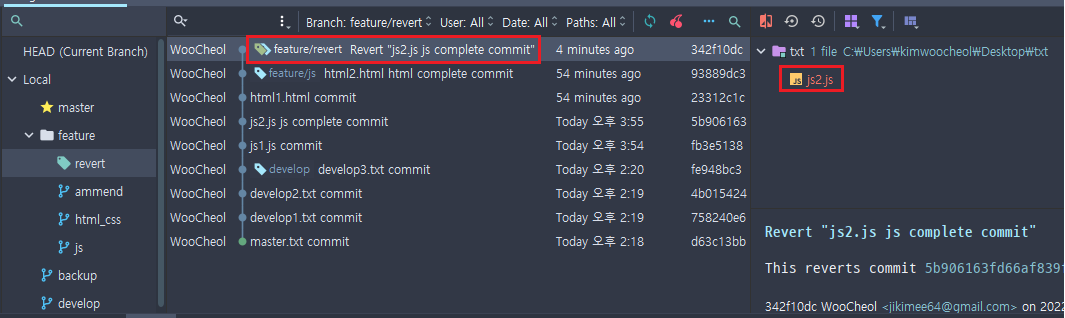


취소하고 싶은 시점의 커밋 오른쪽 마우스 클릭 -> Revert commit

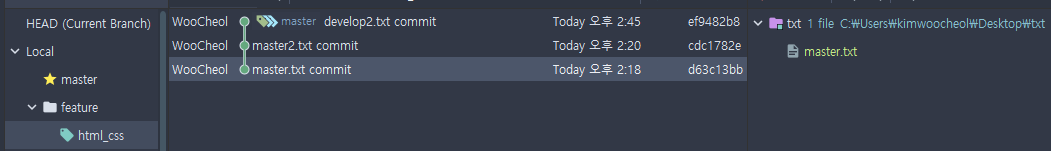


Revert "js2.js js complete commit" 라는 커밋 이력이 추가되었다.

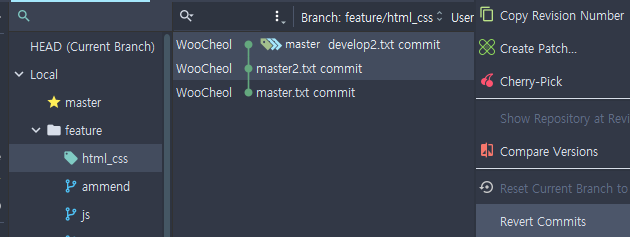
“js2.js js complete commit” 이력에 해당하는 j2.js 파일이 삭제된걸 알 수 있다.

**[과거 시점 까지의 모든 커밋이력 취소(undo)]**

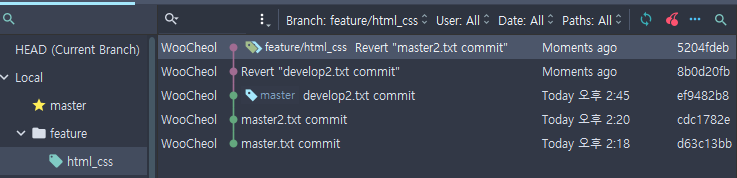
“master.txt commit”으로 되돌아 가고 싶다.



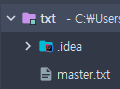
취소할 커밋 이력을 포함하여 오른쪽 마우스 클릭 -> Revert Commits



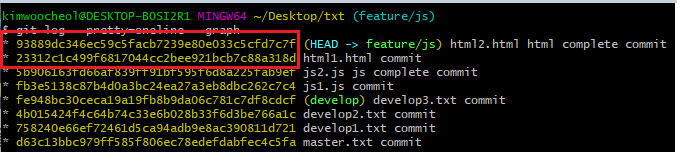
포함한 이력 개수만큼 revert한 커밋 이력이 추가되었고



master.txt파일 하나만 남은걸 알 수 있다.



※ 만약 자동으로 생성되는 revert 커밋 메시지를 직접 수정하려면 –no-commit 옵션을 사용하면 된다.



–no-commit 를 사용한 다음 commit -m “메시지”



–no-commit 를 사용한 다음 commit -m “메시지”



※ 2개의 커밋 이력을 한꺼번에 revert 할려했는데 방법을 찾지 못했다. 대신 하나씩 revert 해서 커밋 메시지를 수정하는게 좋다고 한다.

### **활용사례**

#### **[과거 시점 까지의 모든 커밋이력 취소]**

#### **[특정 과거의 커밋 이력 취소]**

## 6.stash

하던 작업을 임시로 저장 해두고 싶을 때 사용하는 명령어

특정 작업을 위해 불필요한 커밋 이력을 남기지 않아도 된다.

### **간단실습**

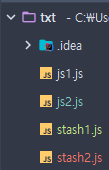
js1.js : 기존 파일 미수정

js2.js : 기존파일 수정

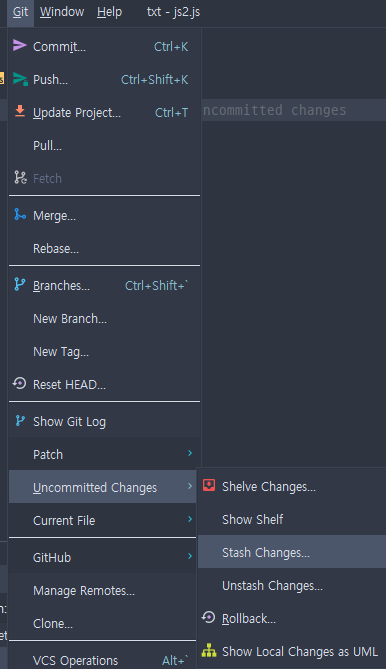
stash1.js 새로운 파일 add 상태

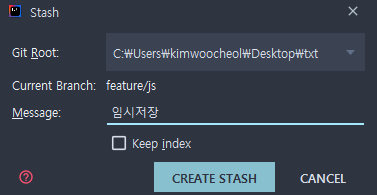
stash2.js : 새로운 파일 but add가 아닌 상태

이상태서 stash 기능을 쓰면 어느파일이 보관되는 걸까?



top 메뉴 git -> Uncommitted Changes -> Stash Changes



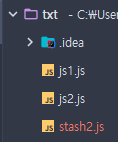


Add되지 않은 stash2.js는 그대로며

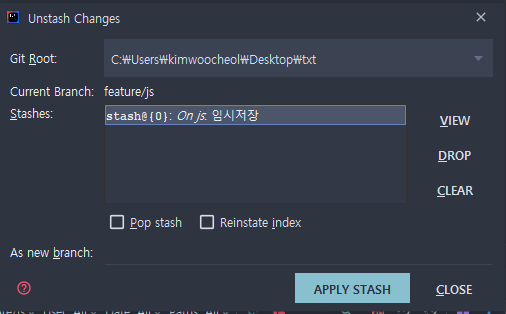
기존파일을 수정한 js2.js는 원본상태로 되돌아왔으며(초록색 -> 흰색)

새롭게 파일 추가 후 add한 stash1.js는 사라졌다.

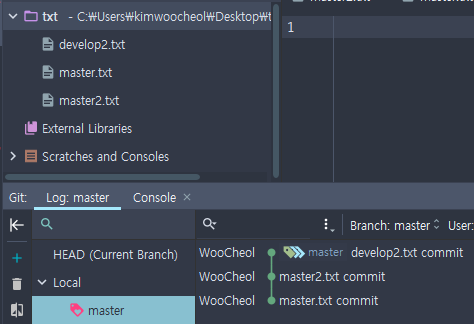
즉, js2.js의 변경사항과 stash1.js가 보관되었다는 걸 알 수 있다.(modified 혹은 트래킹 상태)



stash 이력에 추가된걸 확인 할 수 있다.

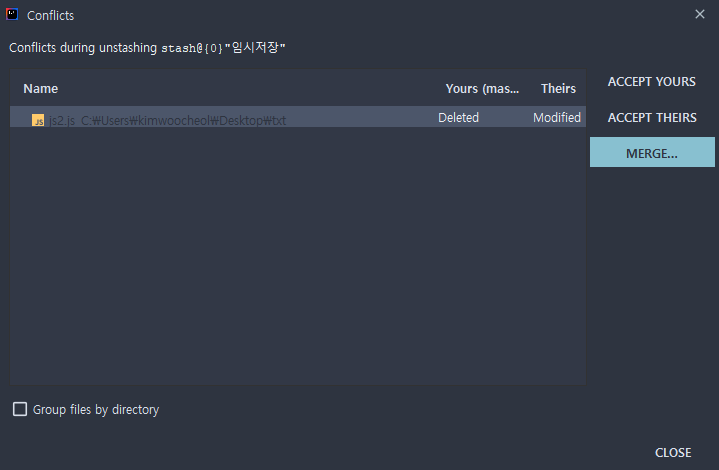


master 브랜치로 이동 후 stash 에 담긴걸 꺼내보자 (Unstash Changes 메뉴에서 선택)



※ 충돌

기존 master에서는 삭제된 상태인데 stash에서는 수정된 상태라서 싱크가 맞지않아 충돌 발생

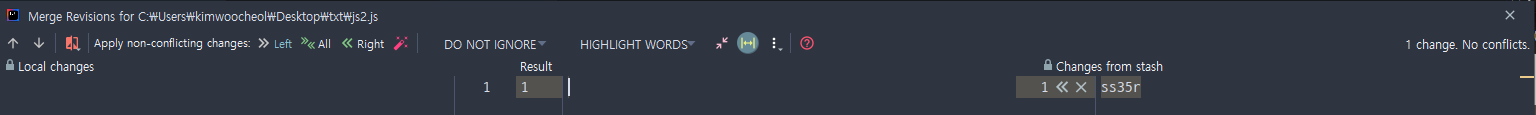


yours는 로컬기준, theirs는 stash 기준

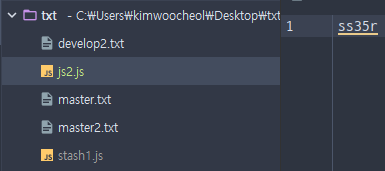
mege를 눌러 어떤 부분이 다른지 자세히 보자.

왼쪽은 로컬, 오른쪽은 theirs다.

가운데가 적용할 버전이며, 우리는 stash에 있는걸 적용하는걸 원하니 오른쪽에 있는 단어(?)를 가운데로 옮겨주자(화살표 클릭)



j2.js와 stash1.js가 반영된걸 알 수 있다.



### **심화(git bash)**

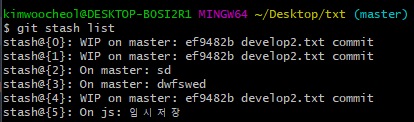
UnTracking인 파일도 포함해서 stash 하고 싶을때

|  |
| --- |
| git stash -all |



stash list 조회

|  |
| --- |
| **git stash list** |

****

새로운 stash를 스택에 만들어 하던 작업을 임시로 저장

|  |
| --- |
| git stash save |

stash stack에서 마지막 작업내용을 불러오되, 스택에 저장된 작업내용을 그대로 보존

(cf. pop은 워킹디렉토리에 작업내용을 불러온 뒤 stack에서는 작업내용을 삭제)

|  |
| --- |
| git stash apply |

stack에서 인덱스번호로 특정 작업내용을 불러오고 싶을 경우 사용

|  |
| --- |
| **git stash apply stash@{index}** |

stash stack에 남아있는 작업내용을 지울 때

|  |
| --- |
| git stash drop stash@{index} |

### **활용사례**

#### **[갑작스런 핫픽스 요청]**

가령 코드를 열심히 수정중인 와중에 갑작스런 핫픽스 요청이 들어오면 기존 수정중인 파일을 백업을 해야한다.

stash를 쓰지 않으면 변경사항 전체를 하드리셋하거나 저장소를 하나더 클론하거나 기능이 완성되지도 않은 코드를 커밋해야 한다.(커밋을 한다면 그 자체는 불필요한 커밋)

이럴때 stash 기능을 활용하고 브랜치를 변경하자

#### **[잘못된 브랜치 에서의 작업]**

master 브랜치에서 불나게 코딩한걸 뒤늦게 깨달았을때 진행중이던 파일들을 stash 에 보관한 후에 브랜치 변경 후 stash에서 꺼내면 끝

#### **[잠시 실험중이던 코드를 보관하고 싶을때]**

브랜치를 하나 파서 해당 브랜치에 커밋하는 방법도 있지만 좋은 Practice는 아니라는 생각이 든다.

### 주의점

git stash명령은 최소한 버전관리가 되고 있는 파일에 대해서만 수행이 가능

즉 기존 파일을 수정했거나 새롭게 파일을 추가후 add하여 Modified, Tracked 상태로 Staging된 파일들만 stash에 보관 가능하다

## 7.tag

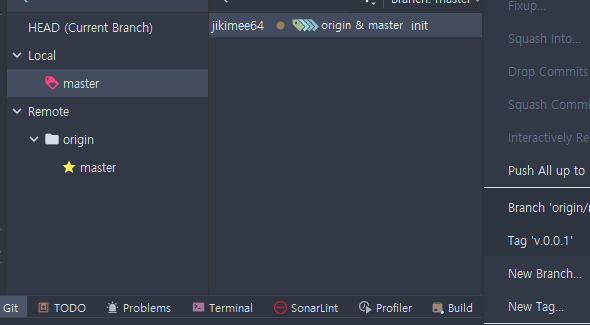
각 커밋에 달 수 있는 꼬리표 같은 것으로, 주로 릴리즈 버전을 표시하는데 사용된다.

태그가 달려있는지 여부와 해당 태그에 이름에 따라 어떤 버전이 중요한 버전인 지 확인할 수 있다.

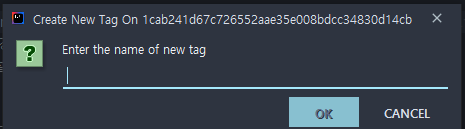
### **간단 실습**

**※ 인텔리제이**

태그를 달 원하는 커밋 오른쪽 클릭 -> New Tag



태그이름 등록



태그이름 확인 가능 (v0.0.1)



git push에서 하단에 tag를 push 하는 선택박스도 있다.



태그를 단 커밋의 오른쪽 마우스 클릭을 보면 삭제하는 기능도 있다.



명령어

태그 생성

|  |
| --- |
| git tag "태그 이름" |

태그 원격 저장소에 push

|  |
| --- |
| git push origin tag-name  git push —tags //모든 태그 push |

태그 삭제

|  |
| --- |
| git push -d "태그 이름" |

커밋 해시값을 통해 특정 커밋에 태그 달기

|  |
| --- |
| git tag -a v0.9 b87aded |

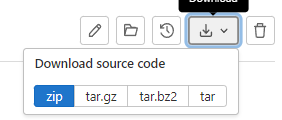
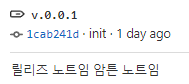
태그 체크아웃

<tags> 버전이 develop가 새롭게 생성되면서 반영(기존에 존재하는 브랜치명이랑 달라야 함)

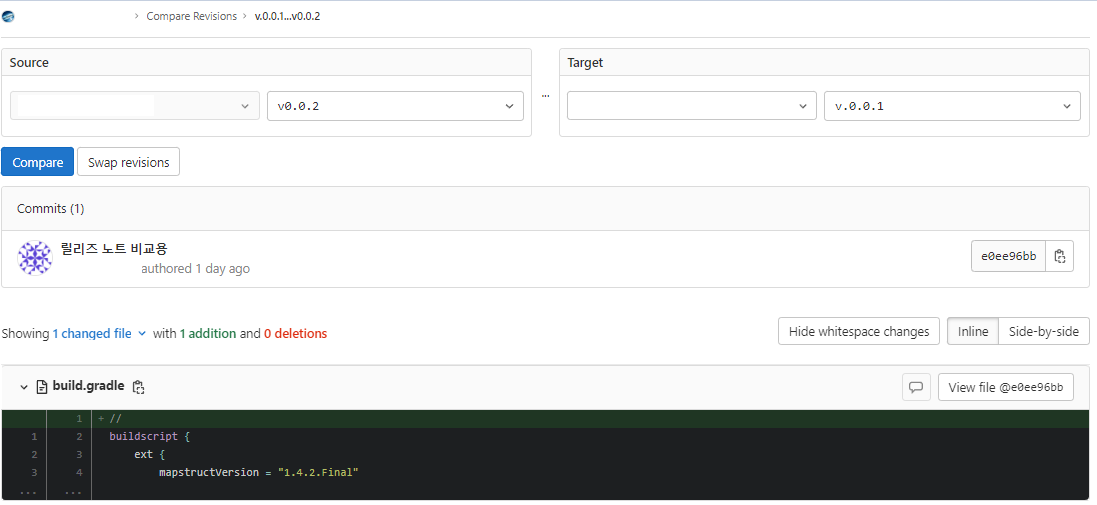
|  |
| --- |
| git checkout tags/<tags> -b develop //브랜치 생성 후 해당 브랜치에 태그코드 반영  git checkout tags/<tags> //특정 브랜치에 속하지 않고 'detached HEAD’ 인 상태로 태그 조회 |

### git 저장소 tag 활용

태그를 저장소에 등록후 관련된 버전에 대한 명시 및 알집 다운 등을 할 수 있다.



태그별로 코드를 compare 할 수 있다.(브랜치도 선택 가능)



### **태그이름 관례**

팀마다 다르겠지만 보통은 아래처럼 쓰지 않는가 싶다.

v1.0.0

날짜까지 붙이면 어떨까..?

v1.0.0-2021-11-11

### **활용사례**

#### **[tags를 이용한 이전 버전으로의 롤백]**

master 브랜치에 적용된 코드가 배포 된 이후에 문제가 발생하지 않는다면 태그를 생성후 저장소에 push

문제가 발생해서 이전 배포버전으로 롤백해야한다면

master -> hotfix 브랜치 생성

배포시 문제된 커밋 이력에 대해서 revert 명령어 수행(바로 즉전 배포버전으로 맞춰짐)

hotfix -> master PR 요청 후 merge 해서 원격 master에 반영

※ CI / CD가 구축되어 있다는 가정

#### **[배포 버전 생성]**

해당 버전에 대한 릴리즈 노트작성 가능

## 8.rebase

특정 브랜치의 커밋 이력을 기준으로 작업 브랜치의 커밋 이력을 재정렬할 때 사용한다.

즉, 브랜치의 커밋들을 재배치하는것

rebase 원리

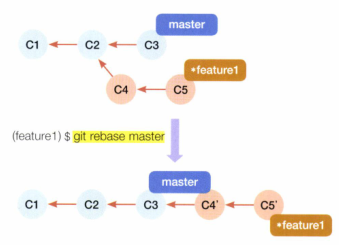
1.HEAD와 대상 브랜치의 공통 조상 찾기(C2)

2.공통 조상 이후에 생성된 커밋들(C4, C5 커밋)을 대상 브랜치 뒤로 재배치

(C4`, C5`로 된 이유는 기존 커밋과 다르기 때문, 해쉬값도 다름)

feature1 브랜치에서

git rebase master 명령어 입력



rebase는 주로 **로컬 브랜치**를 깔끔하게 정리하고 싶을때 사용

### **merge vs rebase**

정답은 없음

merge

모든 이력을 전부 관리할 것이다!!

장점 : 한번만 충돌 발생

단점 :트리가 지저분해짐

reabse

불필요한 이력은 버리고 깔끔하게 갈 것이다!!

장점 : 깔끔한 히슽리

단점 : 여러 번 충돌 발생 가능

### **간단 실습**

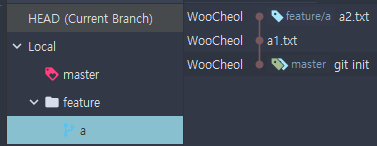
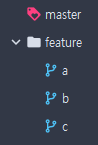
데이터 사전 세팅

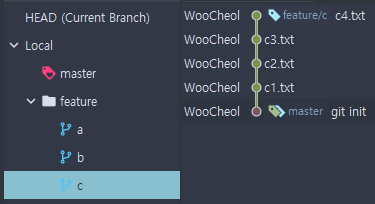
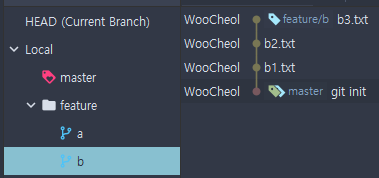
master브랜치, feature/a, feature/b, feature/c 브랜치

feature/a : a1.txt, C2.txt 생성

feature/b : b1.txt, b2.txt, b3.txt 생성

feature/c : c1.txt, c2.txt, c3.txt, c4.txt 생성

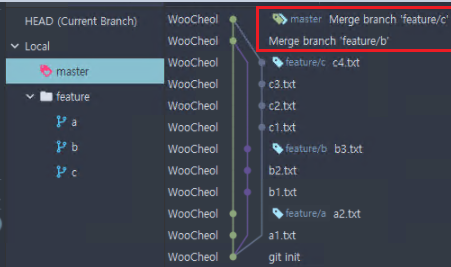




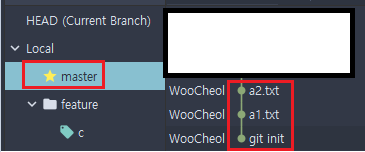
**merge 결과**

master < feature/a, feature/b, feature/c 각각 merge 한 결과

merge commit이 있음



**rebase 본격적인 실습**



1.feature/a로 이동 후 rebase 명령어 입력

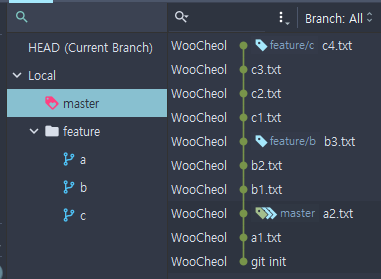
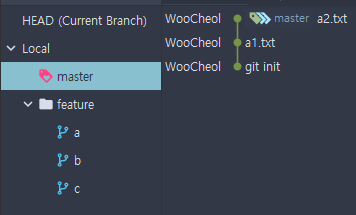
|  |
| --- |
| git rebase master |

※ 충돌이 발생하면 충돌을 해결한 뒤 아래 명령어를 입력

|  |
| --- |
| git add . git rebase --continue |

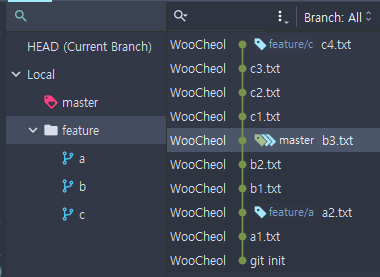
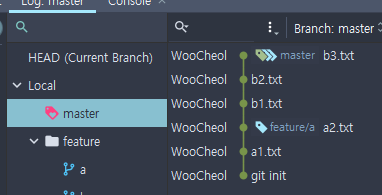
|  |
| --- |
| git checkout master git merge feature/a |

현재 master 및 전체 브랜치 상태(왼쪽 마스터, 오른쪽 전체)



위와 동일한 과정으로 feature/b 브랜치 rebase 수행

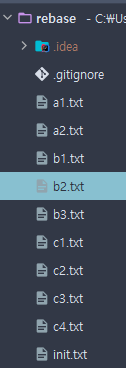
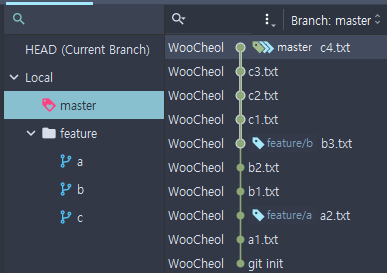
|  |
| --- |
| git checkout feature/b git rebase master git checkout master git merge feateure/b |



위와 동일한 과정으로 feature/c 브랜치 rebase 수행

|  |
| --- |
| git checkout feature/c git rebase master git checkout master git merge feateure/c |

rebase 최종 결과(master 브랜치) 모든 파일도 정상적으로 받은거 확인 가능



※

rebase 중단 명령어

|  |
| --- |
| git rebase -abort |

### 주의점

원격에 push된 브랜치를 대상으로 rebase 하지 않는 것이 원칙

왜? rebase를 하게되면 새로운 커밋해쉬값을 생성하는데 해당 커밋해쉬값이 원격 브랜치의 커밋과 중복 커밋으로 존재하기 때문에 충돌이 발생하고 해당 충돌을 해결하기위한 merge - rebase가 반복되어 히스토리가 꼬인다.

따라서 원격에 존재하지 않는 로컬의 브랜치들에만 적용하기를 권장

### **활용사례**

#### **[merge 커밋을 없에고 히스토리를 깔끔하게 가고 싶을때]**

develop 브랜치를 기준으로 feature/A기능-조회1, feature/A기능-조회2, feature/A기능-조회3

이런식으로 혼자 작업하는 브랜치를 여러개 만든 후 작업이 완료되었다고 가정

각각의 브랜치를 develop에 merge 시키지 말고 rebase를 이용해 3개의 feature 브랜치를 합친뒤

모두 합쳐진 하나의 브랜치만을 develop에 merge하면 됨

하나의 feature브랜치에서만 작업한뒤 develop에 반영하고자 할때도 rebase 쓰면 됨

#### **[다른 사람이 pull을 받지않아 동기화를 시키지 않고 commit & push 했을때]**

다른 사람이 develop 브랜치에 반영한 것을 로컬에서 내가 개발한 것과 합친다고 가정

X,Y가 다른사람이 작업한 내역

C,D가 내가 작업한 내역

A <-- B <-- X <-- Y (origin/develop)

A <-- B <-- C <-- D (develop)

그냥 pull을 받으면 아래와 같이 됩니다.

E 라는 merge 커밋 생성되었습니다.

A <-- B <-- X <-- Y <-- **E** (develop)

↑ ↓

←<- C <-- D <--←

히스토리를 깔끔하게 가고싶으면 아래 명령어를 입력합니다.

|  |
| --- |
| git pull --rebase origin develop |

그럼 아래와 같은 히스토리가 생성됩니다.

C`, D` 와 같이 로컬 저장소의 변경 내역이 재반영 되었고(새로운 커밋 생성) 원격 저장소와 합쳐짐

A <-- B <-- X <-- Y <-- C` <-- D` (master)