

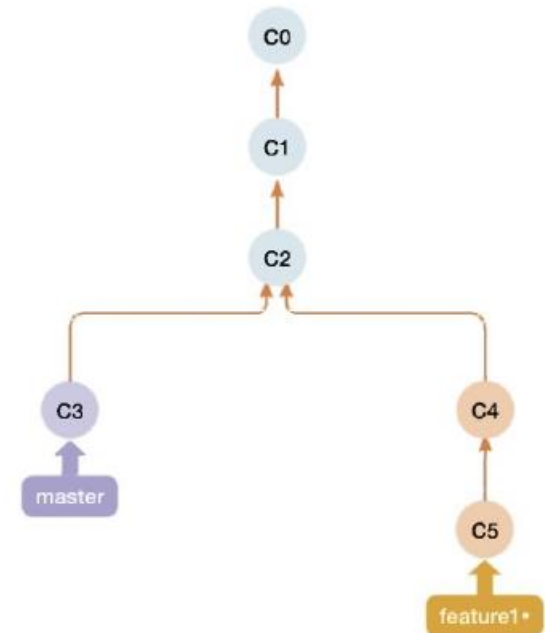


## 브랜치 생성 및 조작하기

# 1 CLI로 브랜치 생성하기

## ❖ branch 되돌아 보기

- 아래 그림은 <https://learngitbranching.js.org/> 이라는 브랜치를 사용하는 방법을 배울 수 있는 사이트를 이용해서 만든 커밋과 브랜치입니다.
- C0부터 C5까지 다섯 개의 커밋이 있고, 두 개의 브랜치(master, feature1)가 생성되어 있습니다.
- [feature1] 브랜치에는 \* 기호가 붙어 있는데 이 마크는 현재 작업 중인 브랜치, 즉 HEAD를 나타냅니다.



# 1 CLI로 브랜치 생성하기

## ❖ branch 되돌아 보기

- 그림으로 표시할 때는 최신 커밋에서 부모 커밋으로 화살표를 그립니다.
- 따라서 커밋은 부모 커밋에 대한 정보를 담고 있습니다.
- 반면 부모 커밋은 자식 커밋에 대한 정보를 담고 있지 않습니다.
- 어느 커밋에서 정보를 가져 오면 그 커밋의 부모 커밋은 알 수 있지만, 자식 커밋은 알 수 없습니다.
- 또한 병합을 통해 생성된 병합 커밋에는 부모 커밋이 두 개 존재합니다.
- 생각해 보면 당연한 얘기입니다.
- 여러분이 꼭 기억하셔야 할 내용은 두 가지 입니다.
  - 커밋하면 커밋 객체가 생깁니다. 커밋 객체에는 부모 커밋에 대한 참조와 실제 커밋을 구성하는 파일 객체가 들어 있습니다.
  - 브랜치는 논리적으로는 어떤 커밋과 그 조상들을 묶어서 뜻하지만, 사실은 단순히 커밋 객체 하나를 가리킬 뿐입니다. 앞의 그림에서 **[master]** 브랜치는 정확하게 **C3** 커밋 객체를, **[feature1]** 브랜치는 **C5** 커밋 객체 하나만을 가리킵니다.

# 1 CLI로 브랜치 생성하기

## ❖ 브랜치 생성하기

<code>git branch [-v]</code>	로컬저장소의 브랜치 목록을 보는 명령으로 -v 옵션을 사용하면 마지막 표시된 브랜치 중에서 이름 왼쪽에 *가 붙어 있으면 HEAD 브랜치입니다.
<code>git branch [-f] &lt;브랜치이름&gt; [커밋체크섬]</code>	새로운 브랜치를 생성합니다. 커밋체크섬 값을 주지 않으면 HEAD로부터 브랜치를 생성합니다. 이미 있는 브랜치를 다른 커밋으로 옮기고 싶을 때는 -f 옵션을 줘야 합니다.
<code>git branch -r[v]</code>	원격 저장소에 있는 브랜치를 보고 싶을 때 사용합니다. 마찬가지로 -v 옵션을 추가하여 커밋 요약도 볼 수 있습니다.
<code>git checkout &lt;브랜치이름&gt;</code>	특정 브랜치로 체크아웃할 때 사용합니다. 브랜치 이름 대신 커밋 체크섬을 쓸 수 있습니다. 하지만 브랜치 이름을 쓰는 방법을 강력히 권장합니다.
<code>git checkout -b &lt;브랜치이름&gt; &lt;커밋 체크섬&gt;</code>	특정 커밋에서 브랜치를 새로 생성하고 동시에 체크아웃까지 합니다. 두 명령을 하나로 합친 명령이기 때문에 간결해서 자주 사용합니다.
<code>git merge &lt;대상 브랜치&gt;</code>	현재 브랜치와 대상 브랜치를 병합할 때 사용합니다. 병합 커밋(merge commit)이 새로 생기는 경우가 많습니다.
<code>git rebase &lt;대상 브랜치&gt;</code>	내 브랜치의 커밋들을 대상 브랜치에 재배치시킵니다. 히스토리가 깔끔해져서 자주 사용하지만 조심해야 합니다. 이유는 추후에 설명합니다.
<code>git branch -d &lt;브랜치이름&gt;</code>	특정 브랜치를 삭제할 때 사용합니다. HEAD 브랜치나 병합이 되지 않은 브랜치는 삭제할 수 없습니다.
<code>git branch -D &lt;브랜치이름&gt;</code>	브랜치를 강제로 삭제하는 명령입니다. -d로 삭제할 수 없는 브랜치를 지우고 싶을 때 사용합니다. 역시 조심해야 합니다.

# 1 CLI로 브랜치 생성하기

## ❖ 브랜치 생성하기

- 자 이제, 드디어 본격적인 실습입니다.
- 브랜치를 만들어 봅시다.
- 이어서 작업을 하고 있다면 현재 두 개의 커밋이 있을 것입니다.
- 혼자서 작업할 때 가장 흔한 워크플로우를 만들어 볼 예정입니다.
- 실습 내용은 새로운 브랜치를 만들고 두 번 커밋을 한 후에 다시 마스터 브랜치로 병합합니다.
- 이 경우에는 마스터 브랜치로 빨리 감기 병합이 가능하므로 CLI로도 간단히 수행할 수 있습니다.

# 1 CLI로 브랜치 생성하기

## ❖ 브랜치 생성하기

- `git branch` 명령을 이용해서 현재 브랜치를 확인하고 새로운 브랜치를 만드는 예제입니다.

```
jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (master)
$ git log --oneline
1e40ead (HEAD -> master, origin/master) 두 번째 커밋
df4ff39 첫 번째 커밋

jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (master)
$ git branch
* master

jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (master)
$ git branch mybranch1

jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (master)
$ git branch
* master
  mybranch1

jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (master)
$ git log --oneline --all
1e40ead (HEAD -> master, origin/master, mybranch1) 두 번째 커밋
df4ff39 첫 번째 커밋

jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (master)
$ |
```

# 1 CLI로 브랜치 생성하기

## ❖ 브랜치 생성하기

- 위 실습 순서대로 알아보겠습니다.

- ① `git log` 명령을 통해 현재 커밋과 브랜치의 상태를 확인합니다. `[origin]`으로 시작하는 브랜치는 원격 브랜치이므로 현재 로컬에는 `[master]` 브랜치만 존재하는 것을 알 수 있습니다. 그리고 `HEAD`가 `[master]` 브랜치를 가리키는 것도 확인할 수 있습니다.
- ② `git branch` 명령을 수행했는데 `*master` 문구는 `HEAD → master`와 동일한 의미입니다. 그리고 프롬프트에 보이는 `(master)` 역시 `HEAD`가 `[master]` 브랜치라는 것을 알려 줍니다.
- ③ `git branch mybranch1` 명령을 통해 새로운 브랜치인 `[mybranch1]` 브랜치를 생성했습니다.
- ④⑤ `git branch`와 `log` 명령으로 결과를 확인합니다. 가장 최신 커밋인 `cflf4dl` 커밋에 `HEAD`, `master`, `mybranch1` 모두 위치하고 있는 것을 알 수 있습니다. 아직 체크아웃 전이기 때문에 여전히 `HEAD`는 `[master]` 브랜치를 가리키고 있습니다.

# 1 CLI로 브랜치 생성하기

## ❖ HEAD에 대해 반드시 기억할 점

- HEAD는 현재 작업 중인 브랜치를 가리킵니다.
- 브랜치는 커밋을 가리키므로 HEAD도 커밋을 가리킵니다.
- 결국 HEAD는 현재 작업 중인 브랜치의 최근 커밋을 가리킵니다.



## 2 CLI 로 checkout 하기

### ❖ CLI를 이용한 브랜치 체크아웃 및 새 커밋 생성

- 앞 절에서 만든 [mybranch1] 브랜치로 체크아웃을 하고, 새로운 커밋을 생성한 뒤에 결과를 확인해 봅니다.
- HEAD가 현재 작업 중인 브랜치의 최근 커밋을 가리킨다는 점을 기억하고 차근차근 명령어를 입력합니다.

```
jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (master)
$ git checkout mybranch1
Switched to branch 'mybranch1'

jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (mybranch1)
$ git branch
  master
* mybranch1

jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (mybranch1)
$ git log --oneline --all
1e40ead (HEAD -> mybranch1, origin/master, master) 두 번째 커밋
df4ff39 첫 번째 커밋

jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (mybranch1)
$ cat file1.txt
hello git
second

jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (mybranch1)
$ echo "third - my branch" >> file1.txt

jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (mybranch1)
$ cat file1.txt
hello git
second
third - my branch

jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (mybranch1)
$ git status
On branch mybranch1
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   file1.txt
```

## 2 CLI 로 checkout 하기

### ❖ CLI를 이용한 브랜치 체크아웃 및 새 커밋 생성

- 앞 절에서 만든 [mybranch1] 브랜치로 체크아웃을 하고, 새로운 커밋을 생성한 뒤에 결과를 확인해 봅니다.
- HEAD가 현재 작업 중인 브랜치의 최근 커밋을 가리킨다는 점을 기억하고 차근차근 명령어를 입력합니다.

```
jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (mybranch1)
$ git add file1.txt
warning: LF will be replaced by CRLF in file1.txt.
The file will have its original line endings in your working directory

jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (mybranch1)
$ git commit
[mybranch1 cf3a363] mybranch1의 첫 번째 커밋
1 file changed, 1 insertion(+)

jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (mybranch1)
$ git log --oneline -all
error: switch '-l' expects a numerical value

jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (mybranch1)
$ git log --oneline
cf3a363 (HEAD -> mybranch1) mybranch1의 첫 번째 커밋
1e40ead (origin/master, master) 두 번째 커밋
df4ff39 첫 번째 커밋
```

## 2 CLI 로 checkout 하기

### ❖ CLI를 이용한 브랜치 체크아웃 및 새 커밋 생성

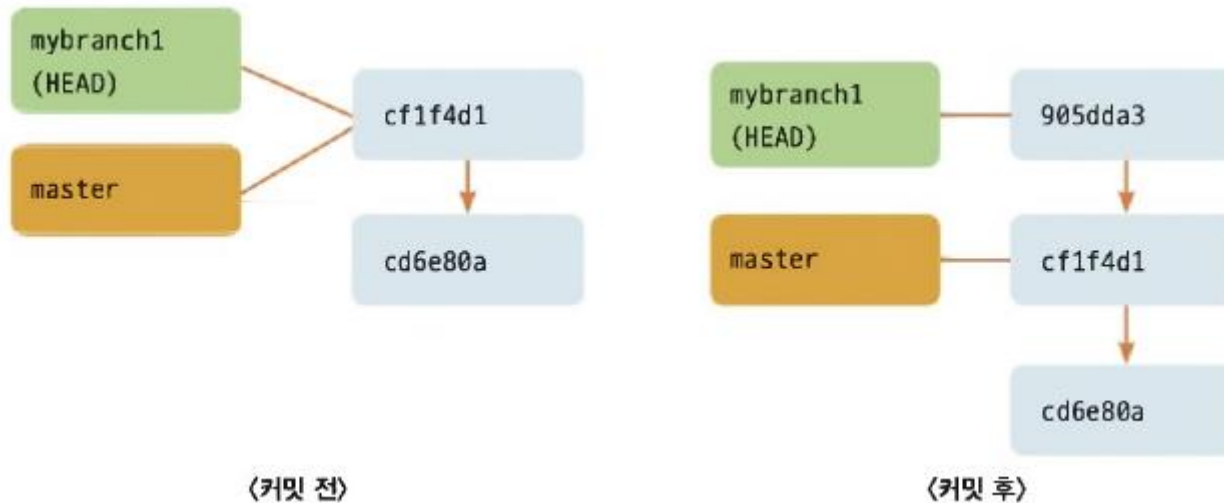
- 위 실행 과정을 설명하면 다음과 같습니다.

- ① `git checkout` 명령을 수행해서 브랜치를 변경했습니다. 보통 체크아웃을 하면 브랜치 변경과 동시에 작업 폴더의 내용도 함께 변경됩니다. 그렇지만 이번에는 `mybranch1` 커밋이 이전 브랜치였던 `[master]`의 커밋과 같은 커밋이라서 작업 폴더의 내용은 변경되지 않습니다.
- ② `git branch`로 현재 브랜치를 확인합니다.
- ③ `git log` 명령을 통해 `HEAD`가 `[mybranch1]`로 변경된 것을 확인할 수 있습니다. 여기서 주의깊게 살펴봤다면 ②에서 프롬프트도 `mybranch1`으로 변경된 것을 확인할 수 있었을 것입니다.
- ④ 파일을 편집하고 새로운 커밋을 생성했습니다.
- ⑤ 커밋 히스토리를 확인합니다.

## 2 CLI 로 checkout 하기

### ❖ CLI를 이용한 브랜치 체크아웃 및 새 커밋 생성

- 커밋 전과 커밋 후의 상태는 다음 그림과 같습니다.



## 2 CLI 로 checkout 하기

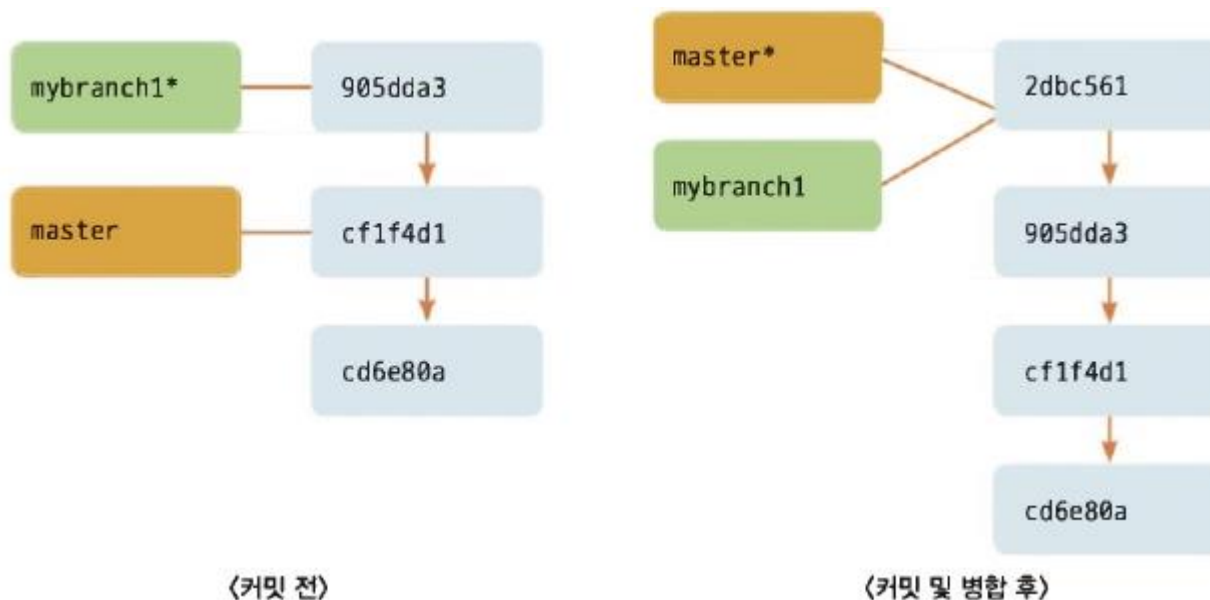
### ❖ 새로운 커밋을 생성하면

- 새로 커밋을 생성하면 그 커밋의 부모는 언제나 이전 HEAD 커밋입니다.
- 커밋이 생성되면 HEAD는 새로운 커밋으로 갱신됩니다.
- HEAD가 가리키는 브랜치도 HEAD와 함께 새로운 커밋을 가리킵니다.

## 2 CLI 로 checkout 하기

### ❖ CLI를 이용한 빨리 감기 병합

- 이번에는 CLI를 이용해서 병합을 해 보겠습니다.
- [mybranch1] 브랜치에서 추가로 한 번 더 커밋을 하고 [master] 브랜치로 체크아웃을 한 후에 [master] 브랜치와 [mybranch1] 브랜치를 병합합니다.



## 2 CLI 로 checkout 하기

### ❖ CLI를 이용한 빨리 감기 병합

```
jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (mybranch1)
$ echo "fourth - my branch" >> file1.txt

jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (mybranch1)
$ cat file1.txt
hello git
second
third - my branch
fourth - my branch

jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (mybranch1)
$ git status
On branch mybranch1
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   file1.txt

no changes added to commit (use "git add" and/or "git commit -a")

jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (mybranch1)
$ git add file1.txt
warning: LF will be replaced by CRLF in file1.txt.
The file will have its original line endings in your working directory

jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (mybranch1)
$ git commit
[mybranch1 2fbf033] mybranch1 두 번째 커밋
1 file changed, 1 insertion(+)

jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (mybranch1)
$ git log --oneline --graph
* 2fbf033 (HEAD -> mybranch1) mybranch1 두 번째 커밋
* cf3a363 mybranch1의 첫 번째 커밋
* 1e40ead (origin/master, master) 두 번째 커밋
* df4ff39 첫 번째 커밋
```

## 2 CLI 로 checkout 하기

### ❖ CLI를 이용한 빨리 감기 병합

```
jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (mybranch1)
$ cat file1.txt
hello git
second
third - my branch
fourth - my branch
```

- 위 명령에 대해 살펴봅니다.

- ① 새로운 커밋인 '2dbc561'을 생성했습니다.
- ② 로그를 보면 기존 커밋을 부모로 하는 새로운 커밋이 생성되었습니다. 그리고 HEAD는 mybranch1, mybranch1은 새 커밋을 각각 가리키는 것을 확인할 수 있습니다.
- ③ checkout 명령을 이용해서 [master] 브랜치를 체크아웃했습니다.
- ④ cat 명령을 통해 텍스트 파일의 내용이 이전으로 돌아간 것을 확인할 수 있습니다.
- ⑤ [master] 브랜치에 [mybranch1] 브랜치를 병합합니다. merge 명령의 출력 결과를 보면 'Updating cflf4dl..2dbc561 Fast-forward' 메시지를 확인할 수 있습니다.



## 2 CLI 로 checkout 하기

### ❖ reset --hard 로 브랜치 되돌리기

- 현재 브랜치를 특정 커밋으로 되돌릴 때 사용합니다. 이 중에서 많이 사용하는 `git reset --hard` 명령을 실행하면 현재 브랜치를 지정한 커밋으로 옮긴 후 해당 커밋의 내용을 작업 폴더에도 반영합니다.

`git reset --hard <이동할 커밋 체크섬>`

현재 브랜치를 지정한 커밋으로 옮긴다. 작업 폴더의 내용도 함께 변경된다.

- 위 명령을 통해 알 수 있는 것처럼 `git reset --hard` 명령을 사용하려면 커밋 체크섬을 알아야 합니다.
- 커밋 체크섬은 `git log`를 통해 확인할 수 있지만 CLI에서 복잡한 커밋 체크섬을 타이핑하는 건 꽤 번거로운 작업입니다.
- 이럴 때는 보통 `HEAD~` 또는 `HEAD^` 로 시작하는 약칭을 사용할 수 있습니다.

`HEAD~<숫자>`

`HEAD~`은 헤드의 부모 커밋, `HEAD~2`는 헤드의 할아버지 커밋을 말한다. `HEAD~n`은 `n`번째 위쪽 조상이라는 뜻이다.

`HEAD^<숫자>`

`HEAD^`은 똑같이 부모 커밋이다. 반면 `HEAD^2`는 두 번째 부모를 가르킨다. 병합 커밋처럼 부모가 둘 이상인 커밋에서만 의미가 있다.

## 2 CLI 로 checkout 하기

### ❖ reset --hard 로 브랜치 되돌리기

- 이번 절에서는 master를 두 커밋 이전 커밋으로 옮겨 보겠습니다.

```
jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (mybranch1)
$ git reset --hard HEAD~2
HEAD is now at 1e40ead 두 번째 커밋

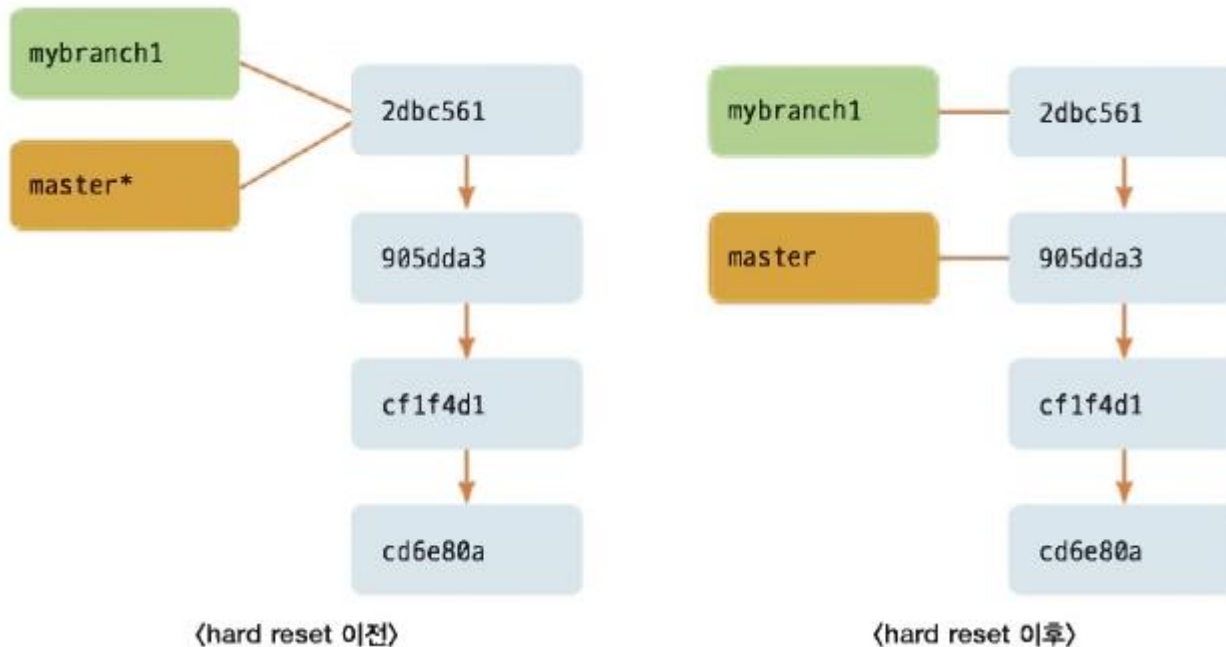
jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (mybranch1)
$ git log --oneline
1e40ead (HEAD -> mybranch1, origin/master, master) 두 번째 커밋
df4ff39 첫 번째 커밋
```

- ① `reset --HEAD~2` 를 실행해서 HEAD를 2단계 이전으로 되돌립니다.
- ② `log` 명령으로 확인해 보면 `HEAD → master`가 달라진 것을 알 수 있습니다.

## 2 CLI 로 checkout 하기

### ❖ `reset --hard` 로 브랜치 되돌리기

- 이번 절에 수행한 명령을 그림으로 나타내면 다음과 같습니다.



## 2 CLI 로 checkout 하기

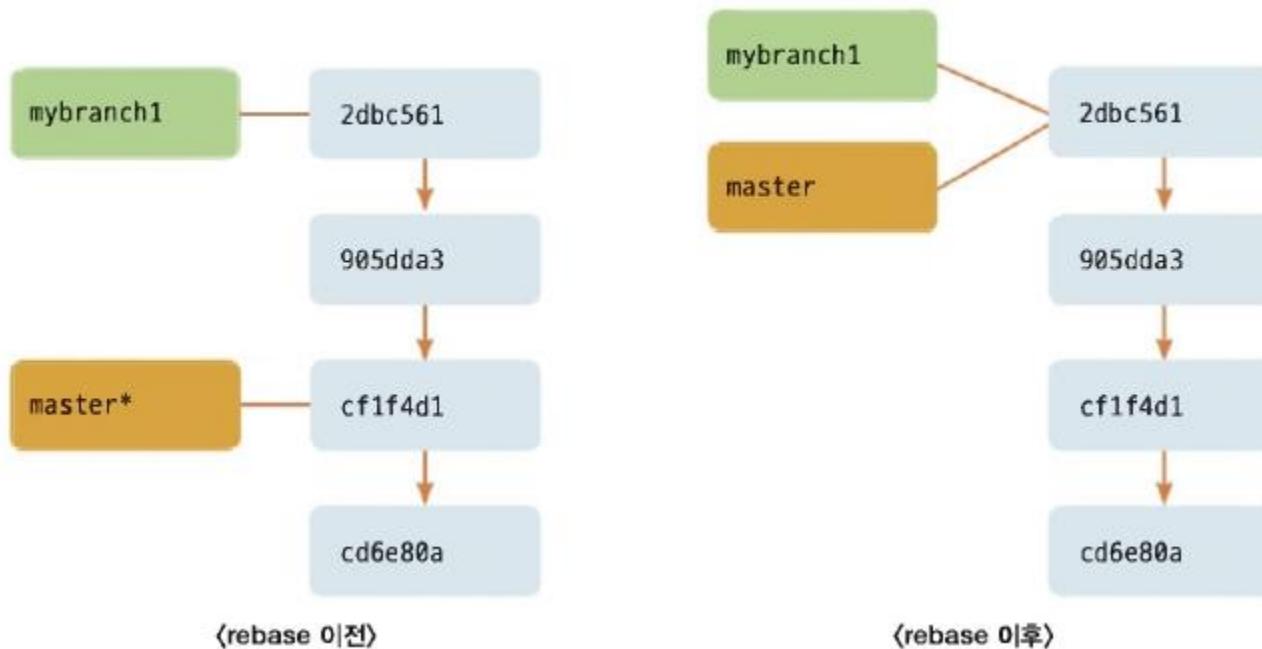
### ❖ 빨리 감기 병합 상황에서 **rebase** 해보기

- 이번에는 빨리 감기 병합이 가능한 상황에서 **rebase** 명령을 수행해 봅시다.
- **git rebase**<대상 브랜치>명령은 현재 브랜치에만 있는 새로운 거밋을 대상 브랜치 위로 재배치시킵니다.
- 그런데 현재 브랜치에 재배치할 커밋이 없을 경우 **rebase**는 아무런 동작을 하지 않습니다.
- 또한 빨리 감기 병합이 가능한 경우에는 **rebase** 명령을 수행하면 빨리 감기 병합을 합니다.

## 2 CLI 로 checkout 하기

### ❖ 빨리 감기 병합 상황에서 rebase 해보기

- 다음 그림은 빨리 감기 병합이 가능한 상황에서 rebase를 수행했을 때입니다.



## 2 CLI 로 checkout 하기

### ❖ 빨리 감기 병합 상황에서 rebase 해보기

- 이번에는 merge 명령 대신 rebase를 이용해 빨리 감기 병합(?)을 하고 [mybranch1] 브랜치를 제거해 보겠습니다.

```
jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (mybranch1)
$ git checkout mybranch1
Already on 'mybranch1'

jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (mybranch1)
$ git rebase master
Current branch mybranch1 is up to date.

jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (mybranch1)
$ git log --oneline
1e40ead (HEAD -> mybranch1, origin/master, master) 두 번째 커밋
df4ff39 첫 번째 커밋

jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (mybranch1)
$ git checkout master
Switched to branch 'master'
Your branch is up to date with 'origin/master'.

jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (master)
$ git rebase mybranch1
Current branch master is up to date.

jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (master)
$ git log --oneline
1e40ead (HEAD -> master, origin/master, mybranch1) 두 번째 커밋
df4ff39 첫 번째 커밋

jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (master)
$ git push
Everything up-to-date

jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (master)
$ git branch -d mybranch1
Deleted branch mybranch1 (was 1e40ead).
```

## 2 CLI 로 checkout 하기

### ❖ 빨리 감기 병합 상황에서 rebase 해보기

- 이번에는 **merge** 명령 대신 **rebase**를 이용해 빨리 감기 병합(?)을 하고 **[mybranch1]** 브랜치를 제거해 보겠습니다.

- ① **[mybranch1]** 브랜치로 체크아웃합니다.
- ② **[mybranch1]** 브랜치는 이미 **[master]** 브랜치 위에 있기 때문에 재배치할 커밋이 없습니다. 그래서 **rebase master**를 수행해도 아무 일도 일어나지 않습니다.
- ③ 다시 **[master]** 브랜치로 체크아웃합니다.
- ④ **rebase** 명령으로 **[master]** 브랜치를 **[mybranch1]** 브랜치로 재배치를 시도합니다. 빨리 감기가 가능한 상황이기 때문에 **rebase**는 **merge** 명령과 마찬가지로 빨리 감기를 하고 작업을 종료합니다.
- ⑤ **git push** 명령으로 **[master]** 브랜치를 원격에 push 합니다.
- ⑥ **git branch -d** 명령으로 필요 없어진 **[mybranch1]** 브랜치를 삭제합니다.

## 2 CLI 로 checkout 하기

### ❖ 배포 배전에 태깅하기

- 이번에는 CLI로 태깅을 해 보겠습니다.
- 태그는 사실 주석 있는 태그와 간단한 태그의 두 종류가 있습니다.
- 일반적으로 주석있는 태그의 사용을 권장하기 때문에 우리도 주석 있는 태그를 사용해 봅시다.
- 실습 전에 관련된 명령부터 살펴볼까요?

`git tag -a -m <간단한 메시지> <태그 이름> [브랜치 또는 체크섬]`

-a 로 주석 있는(annotated) 태그를 생성합니다. 메시지와 태그 이름은 필수이며 브랜치 이름을 생략하면 HEAD에 태그를 생성합니다.

`git push <원격저장소 별명> <태그 이름>`

원격 저장소에 태그를 업로드합니다.



## 2 CLI 로 checkout 하기

### ❖ 배포 배전에 태깅하기

- 실습을 통해 **tag**를 사용해 보고 **GitHub**에서도 확인해 보겠습니다.

```
jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (master)
$ git log --oneline
1e40ead (HEAD -> master, origin/master) 두 번째 커밋
df4ff39 첫 번째 커밋

jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (master)
$ git tag -a -m "첫 번째 태그 생성" v0.1

jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (master)
$ git log --oneline
1e40ead (HEAD -> master, tag: v0.1, origin/master) 두 번째 커밋
df4ff39 첫 번째 커밋

jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (master)
$ git push origin v0.1
Enumerating objects: 1, done.
Counting objects: 100% (1/1), done.
Writing objects: 100% (1/1), 180 bytes | 180.00 KiB/s, done.
Total 1 (delta 0), reused 0 (delta 0), pack-reused 0
To https://github.com/jjin300/hello-git-cli.git
* [new tag]          v0.1 -> v0.1
```

- 태그는 차후에 거밋을 식별할 수 있는 유용한 정보이므로 잘 활용하는 것이 좋습니다.
- 태그를 사용하면 **GitHub**의 **[Tags]** 탭에서 확인할 수 있고, **[Release]** 탭에서 다운받을 수 있다는 것도 기억하고 있죠?

## 2 CLI 로 checkout 하기

### ❖ 배포 배전에 태깅하기

- 지금까지 **CLI**를 통해서 브랜치를 생성하고 체크아웃과 빨리 감기 병합을 했습니다.
- 그리고 빨리 감기가 가능한 상황에서 **merge**와 **rebase**는 같은 동작을 보인다는 것도 확인했습니다.
- 마지막으로 태그의 사용법에 대해서도 알아보았습니다.
- 태그는 정말 유용한 기능이니까 잘 배워서 꼭꼭 활용해 주세요.
- 이제 조금 더 복잡한 상황에서의 **merge**와 **rebase**를 알아볼 것입니다.
- 특히 입문자 분들이 싫어하는 충돌 상황과 히스토리가 꼬인 상황에서의 해결법에 대해서도 알아보겠습니다.
- 겁먹지 말고 차근차근 천천히 보도록 합시다.

# 3 CLI 로 3-way 병합하기

## ❖ 긴급한 버그 처리 시나리오

- 갑작스레 버그를 발견한 상황을 생각해 봅시다.
- 보통 이 경우 하나 이상의 브랜치로 다른 기능 개선을 하고 있을 것입니다.
- 이런 상황에서 버그 수정은 다음과 같은 단계로 이루어 집니다.
  - (옵션) 오류가 없는 버전(주로 Tag가 있는 커밋)으로 롤백
  - [master] 브랜치로부터 [hotfix] 브랜치 생성
  - 빠르게 소스 코드 수정 / 테스트 완료
  - [master] 브랜치로 병합 (Fast-forward) 및 배포
  - 개발 중인 브랜치에도 병합 (충돌 발생 가능성이 높음)
- 버그가 발생한 상황에서는 원래 작업 중이던 브랜치도 [master] 브랜치로부터 시작했기 때문에 같은 버그를 가지고 있을 것입니다.
- 때문에 [hotfix] 브랜치의 내용은 [master] 브랜치와 개발 브랜치 모두에 병합되어야 합니다.
- 보통 [master] 브랜치의 병합은 빨리 감기이기 때문에 쉽게 되는 반면 개발 중인 브랜치의 병합은 병합 커밋이 생성되고, 충돌이 일어날 가능성이 높습니다.

# 3 CLI 로 3-way 병합하기

## ❖ 긴급한 버그 처리 시나리오

- 이러한 상황을 가정하고 실습을 해 보겠습니다.
- 먼저 **[feature1]** 브랜치를 만들고 커밋을 하나 생성합니다.

```
jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (master)
$ git checkout master
Already on 'master'
Your branch is up to date with 'origin/master'.

jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (master)
$ git checkout -b feature1
Switched to a new branch 'feature1'

jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (feature1)
$ echo "기능 1 추가" >> file1.txt

jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (feature1)
$ git add file1.txt
warning: LF will be replaced by CRLF in file1.txt.
The file will have its original line endings in your working directory

jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (feature1)
$ git commit
[feature1 b2fb77d] 새로운 기능 1 추가
1 file changed, 1 insertion(+)

jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (feature1)
$ git log --oneline --graph -n2
* b2fb77d (HEAD -> feature1) 새로운 기능 1 추가
* 1e40ead (tag: v0.1, origin/master, master) 두 번째 커밋
```

# 3 CLI 로 3-way 병합하기

## ❖ 긴급한 버그 처리 시나리오

- 지금 이 시점에서 장애가 발생했습니다.
- 그나마 다행인 점은 이미 커밋을 한 상태에서 장애가 발생했다는 점입니다.
- 현실에서는 거밋을 하기 모호한상황에서 장애가 발생하게 됩니다.
- 이럴 때는 **stash**를 사용할 수 있지만 **stash**에 대해서는 뒤 장에서 설명할테니 일단 커밋을 한 직후에 장애가 발생했다고 가정합니다.
- 이제 버그를 고치기 위해 **[master]** 브랜치에서 **[hotfix]** 브랜치를 먼저 만들어야 합니다.
- 그리고 버그를 고친 후에 커밋을 합니다.
- 그리고 **[hotfix]** 브랜치를 **[master]** 브랜치에 병합합니다.
- **[master]** 브랜치의 최신 커밋을 기반으로 **[hotfix]** 브랜치 작업을 했기 때문에 빨리 감기 병합이 가능한 상황입니다.

# 3 CLI 로 3-way 병합하기

## ❖ 긴급한 버그 처리 시나리오

- hotfix 브랜치 생성, 커밋, master에 병합

```
jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (feature1)
$ git checkout -b hotfix master
Switched to a new branch 'hotfix'

jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (hotfix)
$ git log --oneline -n2
1e40ead (HEAD -> hotfix, tag: v0.1, origin/master, master) 두 번째 커밋
df4ff39 첫 번째 커밋

jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (hotfix)
$ echo "some hot fix" >> file1.txt

jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (hotfix)
$ git add file1.txt
warning: LF will be replaced by CRLF in file1.txt.
The file will have its original line endings in your working directory

jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (hotfix)
$ git commit
[hotfix b1288af] hotfix 실습
1 file changed, 1 insertion(+)

jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (hotfix)
$ git log --oneline -n1
b1288af (HEAD -> hotfix) hotfix 실습

jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (hotfix)
$ git checkout master
Switched to branch 'master'
Your branch is up to date with 'origin/master'.
```

```
jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (master)
$ git merge hotfix
Updating 1e40ead..b1288af
Fast-forward
 file1.txt | 1 +
1 file changed, 1 insertion(+)

jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (master)
$ git push
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Writing objects: 100% (3/3), 276 bytes | 276.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
To https://github.com/jjin300/hello-git-cli.git
1e40ead..b1288af master -> master
```

# 3 CLI 로 3-way 병합하기

## ❖ 긴급한 버그 처리 시나리오

- 아직 추가 작업이 남아 있습니다.
- 물론 긴급한 작업은 끝났으니 한시름 놓은 상태입니다.
- hotfix의 커밋은 버그 수정이었기 때문에 이 내용을 현재 개발 중인 [feature1] 브랜치에도 반영해야 합니다.
- 그런데 [feature1] 브랜치와 [master] 브랜치는 아래 그래프에서 보듯이 서로 다른 분기로 진행되고 있습니다.
- 이 경우에는 빨리 감기 병합이 불가능하므로 3-way 병합을 해야 합니다.
- 따라서 병합 커밋이 생성되겠죠?
- 거기다가 모든 3-way 병합이 충돌을 일으키는 것은 아닙니다만 이번 실습에서는 고의적으로 두 브랜치 모두 file1.txt를 수정했기 때문에 충돌이 발생합니다.

master 브랜치의 file1.txt	feature1 브랜치의 file1 .txt
hello git second third - my branch fourth some hot fix	hello git second third - my branch fourth 신규기능 1

# 3 CLI 로 3-way 병합하기

## ❖ 긴급한 버그 처리 시나리오

- 일단 3-way 병합을 해 봅시다.

```
jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (master)
$ git checkout feature1
Switched to branch 'feature1'

jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (feature1)
$ git log --oneline
b2fb77d (HEAD -> feature1) 새로운 기능 1 추가
1e40ead (tag: v0.1) 두 번째 커밋
df4ff39 첫 번째 커밋

jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (feature1)
$ git merge master
Auto-merging file1.txt
CONFLICT (content): Merge conflict in file1.txt
Automatic merge failed; fix conflicts and then commit the result.

jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (feature1|MERGING)
$ git status
On branch feature1
You have unmerged paths.
  (fix conflicts and run "git commit")
  (use "git merge --abort" to abort the merge)

Unmerged paths:
  (use "git add <file>..." to mark resolution)
    both modified:   file1.txt

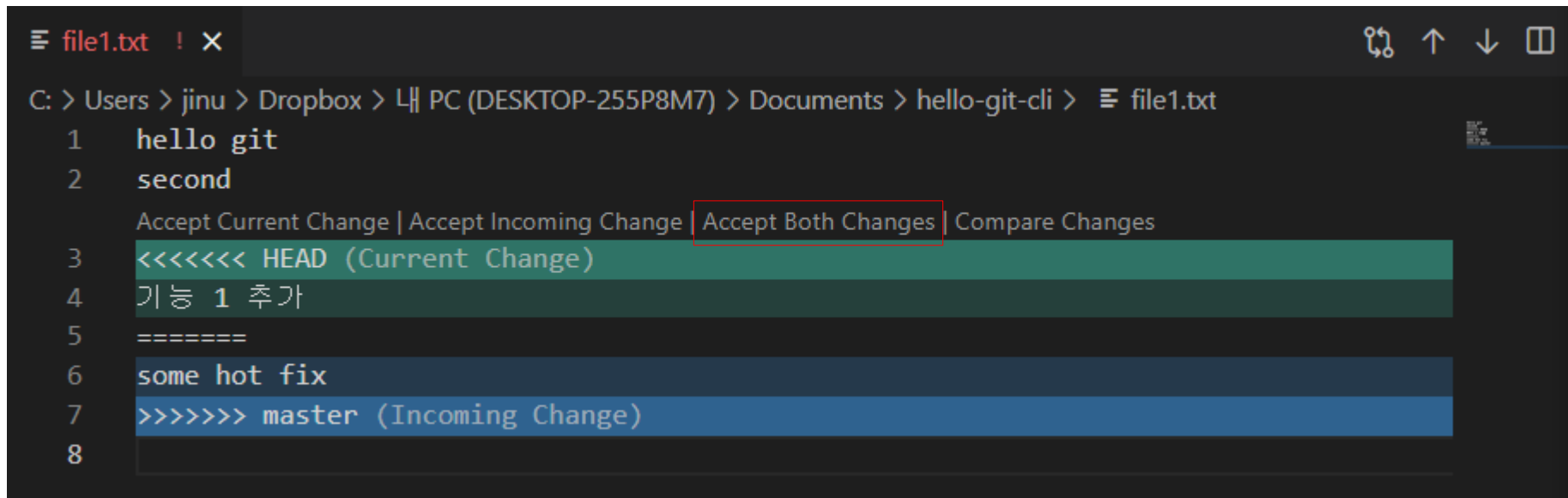
no changes added to commit (use "git add" and/or "git commit -a")
```



# 3 CLI 로 3-way 병합하기

## ❖ 긴급한 버그 처리 시나리오

- 이제 살펴본 것처럼 충돌이 일어난 파일을 비주얼 스튜디오 코드로 열면 아래처럼 충돌 부분이 다른 색으로 표시되고 위 쪽에는 흐릿한 글씨로 4개의 선택 메뉴가 보입니다.
- 첫 번째는 HEAD의 내용만 선택, 두 번째는 master의 내용만 선택, 세 번째는 둘 다 선택, 네 번째는 다른 내용을 확인하는 버튼입니다.



```
file1.txt ! X
C: > Users > jinu > Dropbox > 내 PC (DESKTOP-255P8M7) > Documents > hello-git-cli > file1.txt
1  hello git
2  second
   Accept Current Change | Accept Incoming Change | Accept Both Changes | Compare Changes
3  <<<<<<< HEAD (Current Change)
4  기능 1 추가
5  =====
6  some hot fix
7  >>>>>>> master (Incoming Change)
8
```

# 3 CLI 로 3-way 병합하기

## ❖ 긴급한 버그 처리 시나리오

- 여기서는 둘 다 필요한 내용이므로 [두 변경 사항 모두 수락]을 선택하면 최종적인 모습은 아래와 같이 됩니다.
- 지저분한 <<<<HEAD, =====, >>>> master 같은 내용을 자동으로 제거해 줘서 한결 보기 편해졌습니다.

≡ file1.txt ! ✕

C: > Users > jinu > Dropbox > 내 PC (DESKTOP-255P8M7) > Documents > hello-git-cli > ≡ file1.txt

```
1  hello git
2  second
3  기능 1 추가
4  some hot fix
5  
```

# 3 CLI 로 3-way 병합하기

## ❖ 긴급한 버그 처리 시나리오

- 이제 변경 내용을 저장하고 다시 스테이지에 추가 및 커밋을 하면 수동 3-way 병합이 완료됩니다.

```
jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (feature1|MERGING)
$ cat file1.txt
hello git
second
기능 1 추가
some hot fix

jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (feature1|MERGING)
$ git add file1.txt

jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (feature1|MERGING)
$ git status
On branch feature1
All conflicts fixed but you are still merging.
(use "git commit" to conclude merge)

Changes to be committed:
  modified:   file1.txt

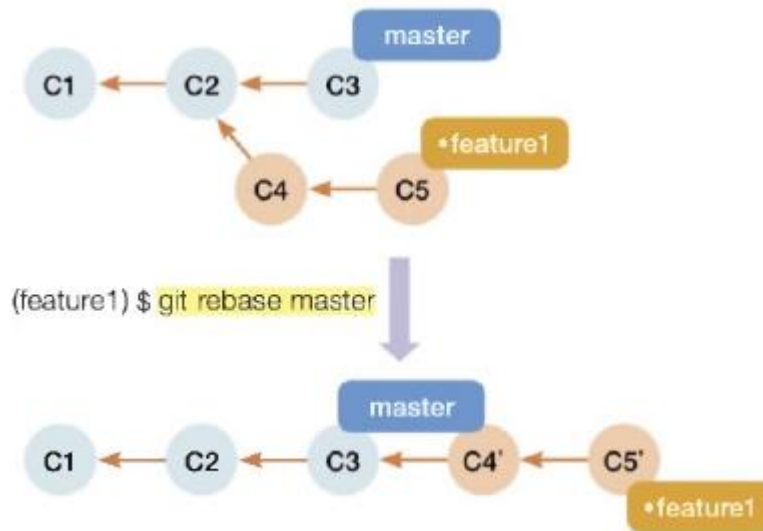
jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (feature1|MERGING)
$ git commit
[feature1 bfdba70] Merge branch 'master' into feature1

jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (feature1)
$ git log --oneline --graph -n4
*   bfdba70 (HEAD -> feature1) Merge branch 'master' into feature1
| \
|  * b1288af (origin/master, master, hotfix) hotfix 실습
* | b2fb77d 새로운 기능 1 추가
|/
* 1e40ead (tag: v0.1) 두 번째 커밋
```

# 4 CLI 로 rebase 해보기

## ❖ rebase 사용하기

- 3-way 병합을 하면 병합 커밋이 생성되기 때문에 트리가 다소 지저분해진다는 단점이 있습니다.
- 이럴 때 트리를 깔끔하게 하고 싶다면 **rebase**를 사용할 수 있습니다.
- **rebase**의 원리를 다시 살펴보면 다음과 같습니다.
  - ① HEAD와 대상 브랜치의 공통 조상을 찾는다. (아래 그림의 C2)
  - ② 공통 조상 이후에 생성한 커밋들(C4, C5 커밋)을 대상 브랜치 뒤로 재배치한다.



# 4 CLI 로 rebase 해보기

## ❖ rebase 사용하기

- 위 그림을 유심히 살펴봅시다.
- 먼저 **[feature 1]** 브랜치는 **HEAD**이므로 \*이 붙어 있습니다.
- 여기서 **git rebase master** 명령을 수행하면 공통 조상인 **C2** 이후의 커밋인 **C4**와 **C5**를 **[master]** 브랜치의 최신 커밋인 **C3** 뒤 쪽으로 재배치를 수행합니다.
- 그런데 재배치된 **C4**와 **C5** 커밋은 각각 **C4'**와 **C5'**가 되었습니다.
- 이 말은 이 커밋은 원래의 커밋과 다른 커밋이라는 뜻입니다.
- 실습을 할 때도 **rebase** 전과 후에 커밋 체크섬을 확인해 보면 값이 달라진 것을 직접 확인할 수 있습니다.
- **rebase** 명령어는 주로 로컬 브랜치를 깔끔하게 정리하고 싶을 때 사용합니다.
- 원격에 푸시한 브랜치를 **rebase**할 때는 조심해야 합니다.
- 여러 **Git** 가이드에서 원격 저장소에 존재하는 브랜치에 대해서는 **rebase**를 하지 말 것을 권하고 있습니다.

# 4 CLI 로 rebase 해보기

## ❖ rebase 사용하기

- 일단 앞 절에서 만들었던 병합 거밋을 되돌리고 rebase를 해 보겠습니다.
- [feature1] 브랜치를 한 단계 되돌릴 때는 `git reset --hard` 명령을 사용합니다.

```
jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (feature1)
$ git checkout feature1
Already on 'feature1'

jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (feature1)
$ git reset --hard HEAD~
HEAD is now at b2fb77d 새로운 기능 1 추가

jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (feature1)
$ git log --oneline --graph -n3
* b2fb77d (HEAD -> feature1) 새로운 기능 1 추가
* 1e40ead (tag: v0.1) 두 번째 커밋
* df4ff39 첫 번째 커밋

jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (feature1)
$ git rebase master
Auto-merging file1.txt
CONFLICT (content): Merge conflict in file1.txt
error: could not apply b2fb77d... 새로운 기능 1 추가
hint: Resolve all conflicts manually, mark them as resolved with
hint: "git add/rm <conflicted_files>", then run "git rebase --continue".
hint: You can instead skip this commit: run "git rebase --skip".
hint: To abort and get back to the state before "git rebase", run "git rebase --abort".
Could not apply b2fb77d... 새로운 기능 1 추가
```

## 4 CLI 로 rebase 해보기

### ❖ rebase 사용하기

- ① HEAD를 [feature1] 브랜치로 전환합니다.
- ② `git reset --hard HEAD~` 명령으로 커밋을 한 단계 이전으로 되돌렸습니다.
- ③ 이렇게 하면 병합 커밋이 사라집니다.
- ④ 로그를 통해 커밋 체크섬을 확인합니다.
- ⑤ 재배포 대상 커밋의 체크섬 값이 'b2fb77d'라는 것을 알 수 있습니다.
- ⑥ `rebase`를 시도하지만 `merge`에서와 마찬가지로 충돌로 인해 `rebase`는 실패합니다. 여기서 실패 메시지를 잘 보면 수동으로 충돌을 해결한 후에 스테이지에 추가를 할 것을 알려줍니다. 그리고 난 후 `git rebase --continue` 명령을 수행하라는 것도 알려줍니다.

# 4 CLI 로 rebase 해보기

## ❖ rebase 사용하기

- 다시 충돌을 해결하고 rebase를 계속해 보겠습니다.

```
jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (feature1|REBASE 1/1)
$ git status ①
interactive rebase in progress; onto b1288af
Last command done (1 command done):
  pick b2fb77d 새로운 기능 1 추가
No commands remaining.
You are currently rebasing branch 'feature1' on 'b1288af'.
  (fix conflicts and then run "git rebase --continue")
  (use "git rebase --skip" to skip this patch)
  (use "git rebase --abort" to check out the original branch)

Unmerged paths:
  (use "git restore --staged <file>..." to unstage)
  (use "git add <file>..." to mark resolution)
    both modified:   file1.txt

no changes added to commit (use "git add" and/or "git commit -a")

jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (feature1|REBASE 1/1)
$ git add file1.txt ②

jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (feature1|REBASE 1/1)
$ git stauts
git: 'stauts' is not a git command. See 'git --help'.

The most similar command is
  status
```

```
jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (feature1|REBASE 1/1)
$ git status
interactive rebase in progress; onto b1288af
Last command done (1 command done):
  pick b2fb77d 새로운 기능 1 추가
No commands remaining.
You are currently rebasing branch 'feature1' on 'b1288af'.
  (all conflicts fixed: run "git rebase --continue")

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    modified:   file1.txt

jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (feature1|REBASE 1/1)
$ git rebase --continue ③
[detached HEAD 4e0bd48] 새로운 기능 1 추가
1 file changed, 4 insertions(+)
Successfully rebased and updated refs/heads/feature1.

jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (feature1)
$ git log --oneline --graph -n2 ④
* 4e0bd48 (HEAD -> feature1) 새로운 기능 1 추가
* b1288af (origin/master, master, hotfix) hotfix 실습

jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (feature1)
$ git checkout master
Switched to branch 'master'
Your branch is up to date with 'origin/master'.

jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (master)
$ git merge feature1 ⑤
Updating b1288af..4e0bd48
Fast-forward
 file1.txt | 4 ++++
1 file changed, 4 insertions(+)
```



## 4 CLI 로 rebase 해보기

### ❖ rebase 사용하기

- ① 충돌 파일을 확인하고 이전과 같은 방식으로 비주얼 스튜디오 코드를 이용해서 수동으로 파일 내용을 수정하고 저장합니다.
- ② 스테이지에 변경사항을 추가합니다.
- ③ **git rebase --continue** 명령을 수행해서 이어서 리베이스 작업을 진행합니다. **merge**는 마지막 단계에서 **git commit** 명령을 사용하지만, **rebase**는 **git rebase --continue** 명령을 사용해야 합니다. 주의하기 바랍니다.
- ④ 로그를 확인합니다. **merge**와는 달리 병합 커밋도 없고 히스토리도 한 줄로 깔끔해졌습니다. 또한 **[feature1]** 브랜치가 가리키는 커밋의 체크섬 값이 **'4e0bd48'**로 바뀐 것을 볼 수 있습니다. 이는 앞서 설명한 것처럼 리베이스를 하면 커밋 객체가 바뀌기 때문입니다.
- ⑤ 마지막으로 **[master]** 브랜치에서 **[feature1]** 브랜치로 병합합니다. 한 줄이 되었기 때문에 빨리 감기 병합을 수행합니다.

## 4 CLI 로 rebase 해보기

### ❖ rebase 사용하기

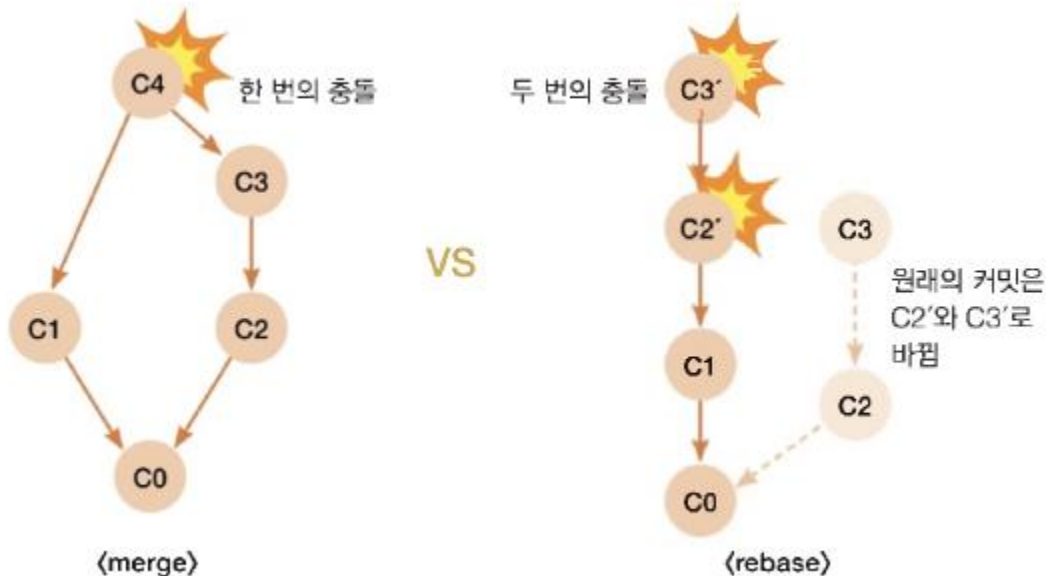
- **rebase**와 **merge**의 마지막 단계에서 명령어가 다른 것이 이상하다고 여길 수 있는데요.
- **3-way** 병합은 기존 커밋의 변경 없이 새로운 병합 커밋을 하나 생성합니다.
- 따라서 충돌도 한 번만 발생합니다.
- 충돌 수정 완료 후 **git commit** 명령을 수행하면 **merge** 작업이 완료됩니다.
- 그러나 **rebase**는 재배치 대상 커밋이 여러 개일 경우 여러 번 충돌이 발생할 수 있습니다.
- 또한 기존의 커밋을 하나씩 단계별로 수정하기 때문에 **git rebase continue** 명령으로 충돌로 인해 중단된 **rebase**를 재개하게 됩니다.
- 여러 커밋에 충돌이 발생했다면 충돌을 해결할 때마다 **git rebase --continue** 명령을 매번 입력해야 합니다.
- 복잡해지고 귀찮기 때문에 이런 경우에는 병합을 수행하는 것이 더 간단할 수도 있습니다.

# 4 CLI 로 rebase 해보기

## ❖ rebase 사용하기

	3-way 병합	rebase
특징	머지 커밋 생성	현재 커밋들을 수정하면서 대상 브랜치 위로 재배포함
장점	한 번만 충돌 발생	깔끔한 히스토리
단점	트리가 약간 지저분해짐	여러 번 충돌이 발생할 수 있음

- 다음 그림을 보고 3-way 병합과 rebase의 차이를 다시 한번 천천히 생각해 보기 바랍니다.



## 4 CLI 로 rebase 해보기

### ❖ 유용한 rebase의 사용법: 뺀어나온 가지 없애기

- 지금 같은 경우는 불필요하게 병합 커밋이 생긴 상황입니다.
- 그렇다면 위 상황을 어떻게 깔끔하게 정리할 수 있을까요?
- 답은 `reset -hard`로 병합 커밋을 되돌리고 `rebase`를 사용하는 것입니다.
- 정말 간단하고 효과적인 방법으로 종종 사용되니 잘 기억하고 활용하십시오.
- 먼저 가지 커밋을 하나 만들어 봅시다.
- 가지를 만들기 위해 정상인 커밋을 만들고 푸시합니다.

# 4 CLI 로 rebase 해보기

## ❖ 유용한 rebase의 사용법: 뺏어나온 가지 없애기

- 먼저 가지 커밋을 하나 만들어 봅시다.
- 가지를 만들기 위해 정상인 커밋을 만들고 푸시합니다.

```
jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (master)
$ echo "master1" > master1.txt

jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (master)
$ git add master1.txt
warning: LF will be replaced by CRLF in master1.txt.
The file will have its original line endings in your working directory

jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (master)
$ git commit -m "master 커밋 1"
[master 82443bd] master 커밋 1
1 file changed, 1 insertion(+)
create mode 100644 master1.txt

jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (master)
$ git push origin master
Enumerating objects: 8, done.
Counting objects: 100% (8/8), done.
Delta compression using up to 16 threads
Compressing objects: 100% (4/4), done.
Writing objects: 100% (6/6), 614 bytes | 614.00 KiB/s, done.
Total 6 (delta 0), reused 0 (delta 0), pack-reused 0
To https://github.com/jjin300/hello-git-cli.git
b1288af..82443bd master -> master

jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (master)
$ git log --oneline -n1
82443bd (HEAD -> master, origin/master) master 커밋 1

jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (master)
$ ls
file1.txt  master1.txt
```

# 4 CLI 로 rebase 해보기

## ❖ 유용한 rebase의 사용법: 뺏어나온 가지 없애기

- 일단 평범하게 커밋을 하나 생성했습니다.
- 이제 **reset --hard**를 이용해서 한 단계 이전 커밋으로 가서 다시 커밋을 생성하면 가지가 하나 생겨날 것입니다.

```
jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (master)
$ git reset --hard HEAD~ ①
HEAD is now at 4e0bd48 새로운 기능 1 추가

jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (master)
$ echo "master2" > master2.txt

jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (master)
$ git add .
warning: LF will be replaced by CRLF in master2.txt.
The file will have its original line endings in your working directory

jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (master)
$ git commit -m "master2 커밋" ②
[master 239589b] master2 커밋
1 file changed, 1 insertion(+)
create mode 100644 master2.txt

jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (master)
$ git log --oneline --graph -n3 --all
* 871f96c (HEAD -> master) Merge branch 'master' of https://github.com/jjin300/hello-git-cli
|
| * 82443bd (origin/master) master 커밋 1
* | 239589b master2 커밋
|/
```

- ① **hard reset**을 이용해서 **[master]** 브랜치를 한 단계 되돌립니다.
- ② **master2.txt** 파일을 생성하고 커밋을 합니다.

## 4 CLI 로 rebase 해보기

### ❖ 유용한 rebase의 사용법: 뺏어나온 가지 없애기

- 지금 상황에서 `git pull`을 하면 어떻게 될까요?
- `git pull` = `git fetch` + `git merge`이기 때문에 가지를 병합하기 위해서 병합 커밋이 생기고 괜히 커밋 히스토리가 지저분해 집니다.

```
jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (master)
$ git pull
Merge made by the 'ort' strategy.
 master1.txt | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 master1.txt

jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (master)
$ git log --oneline --graph -n4 --all
*   871f96c (HEAD -> master) Merge branch 'master' of https://github.com/jjin300/hello-git-cli
|\
| * 82443bd (origin/master) master 커밋 1
| * | 239589b master2 커밋
|/
* 4e0bd48 (feature1) 새로운 기능 1 추가
```

- ① `git pull` 명령을 수행합니다. 자동으로 병합 커밋이 생성됩니다. 병합 커밋 생성 시 에디터가 뜨는데 그냥 닫으면 됩니다.
- ② 로그를 확인해 보면. 병합 커밋이 생성된 것을 알 수 있습니다.

## 4 CLI 로 rebase 해보기

### ❖ 유용한 rebase의 사용법: 뺀어나온 가지 없애기

- 저는 예전에는 불필요한 병합 커밋이 싫어서 `git pull`을 잘 실행하지 않았는데, 최근에는 `git pull`을 그냥 사용하게 되었습니다.
- 사실 병합 커밋이 생기는 빈도는 그리 높지 않습니다.
- 그렇기 때문에 생성되면 그때 `hard reset`을 이용해 되돌리고 `rebase`를 하면 됩니다.
- 이제 병합 커밋을 되돌린 후에 `rebase`로 가지를 없애 보겠습니다.

```
jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (master)
$ git reset --hard HEAD~ ①
HEAD is now at 239589b master2 커밋

jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (master)
$ git rebase origin/master ②
Successfully rebased and updated refs/heads/master.

jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (master)
$ git log --oneline --graph -n3 --all
* 4641593 (HEAD -> master) master2 커밋
* 82443bd (origin/master) master 커밋 1
* 4e0bd48 (feature1) 새로운 기능 1 추가

jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (master)
$ git push ③
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 16 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 317 bytes | 317.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
To https://github.com/jjin300/hello-git-cli.git
82443bd..4641593 master -> master
```



## 4 CLI 로 rebase 해보기

### ❖ 유용한 rebase의 사용법: 뺀어나온 가지 없애기

- ① `reset --hard HEAD~` 명령을 이용해서 커밋을 하나 되돌립니다. 이 경우 마지막 커밋은 병합 커밋이었으므로 병합되기 전 커밋으로 돌아가게 됩니다. 이 커밋이 튀어나온 커밋이니까 어딘 가에 재배포를 해야 합니다
- ② `git abase origin/master` 명령을 수행하면 로컬 `[master]` 브랜치의 가지 커밋이 `[origin/master]` 브랜치 위로 재배포됩니다.
- ③ 로그를 확인하고 `origin`에 푸시합니다.
  - 이제 튀어나온 가지가 사라졌습니다. 보기 좋죠?

## 4 CLI 로 rebase 해보기

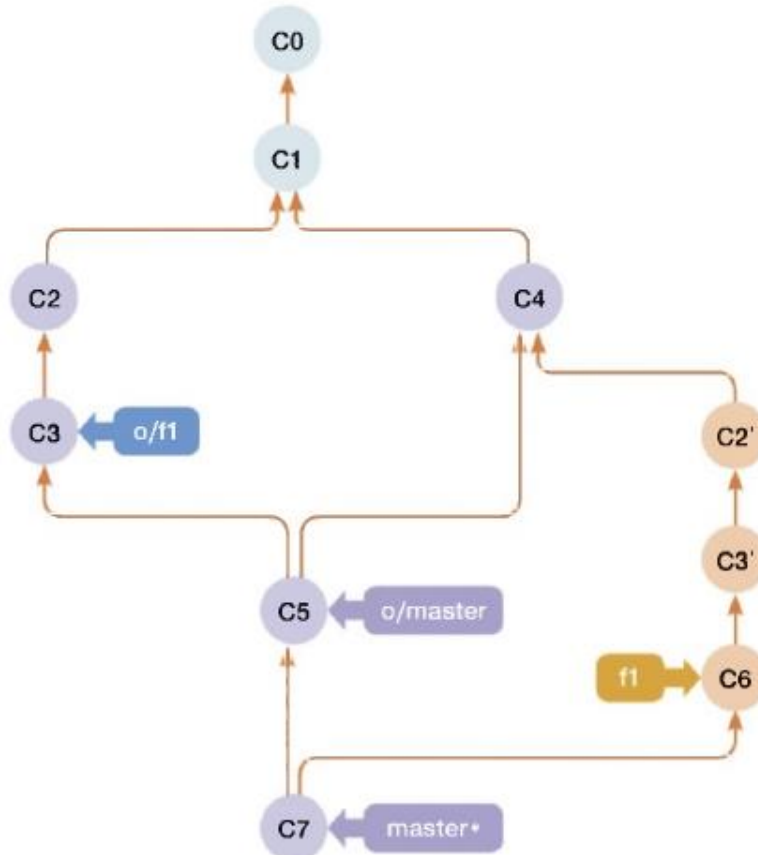
### ❖ rebase 주의사항

- **rebase**에는 중요한 주의사항이 있습니다.
- 원격 저장소에 푸시한 브랜치는 **rebase**하지 않는 것이 원칙입니다.
- 예를 들어 **C1** 커밋을 원격에 푸시하고 **rebase**를 하게 되면 원격에는 **C1**이 존재하고 로컬에는 다른 커밋인 **C1'**가 생성됩니다.
- 이때 내가 아닌 다른 사용자는 원격에 있던 **C1**을 병합할 수 있습니다.
- 그런데 변경된 **C1'**도 언젠가는 원격에 푸시되고 그럼 원격에는 실상 같은 커밋이었던 **C1** 과 **C1'**이 동시에 존재하게 됩니다.
- 이 상황에서 또 누군가는 충돌을 해결하기 위해 **merge**와 **rebase**를 사용하게 되는데..., 정말 끔찍한 상황이 일어나게 됩니다.
- 동일한 커밋의 사본도 여러 개 존재하고, 충돌도 발생하고, 히스토리는 꼬여만 갑니다.

# 4 CLI 로 rebase 해보기

## ❖ rebase 주의사항

- 따라서 **rebase**와 **git**의 동작 원리를 잘 이해하기 전까지는 가급적 **rebase**는 아직 원격에 존재하지 않는 로컬의 브랜치들에만 적용하기를 강력하게 권장합니다.



(중복 커밋이 존재하는 저장소 상태)

## 4 CLI 로 rebase 해보기

### ❖ 임시 브랜치 사용하기

- 많은 입문자가 충돌 해결. **merge**, **rebase** 등을 할 때 막연히 걱정부터 합니다.
- 소스가 깨지거나 열심히 한 작업의 내용이 사라지는 두려움, 그리고 **Git**의 커밋 히스토리가 꼬일 것 같은 느낌이 동시에 들기 때문이죠.
- 이럴 때 걱정을 덜 수 있는 아주 쉬운 방법이 있습니다.
- 임시 브랜치를 활용하는 것입니다.
- 원래 작업하려고 했던 브랜치의 커밋으로 임시 브랜치를 만들고 나면 해당 브랜치에서는 아무 작업이나 막 해도 전혀 상관없습니다.
- 나중에 그 브랜치를 삭제하기만 하면 모든 내용이 원상 복구됩니다.
- 임시 브랜치가 필요 없어지는 시점에 CLI에서 **git branch -D <브랜치 이름>** 명령으로 삭제할 수 있습니다.

# 4 CLI 로 rebase 해보기

## ❖ 임시 브랜치 사용하기

### ■ 임시 브랜치 생성 및 삭제

```
jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (master)
$ git branch test feature1

jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (master)
$ git checkout test
Switched to branch 'test'

jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (test)
$ echo "아 무 말 대 잔 치" > test.txt

jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (test)
$ git add .
warning: LF will be replaced by CRLF in test.txt.
The file will have its original line endings in your working directory

jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (test)
$ git commit -m "임시 커밋"
[test e2df3c5] 임시 커밋
1 file changed, 1 insertion(+)
create mode 100644 test.txt

jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (test)
$ git log --oneline --graph -n4 --all
* e2df3c5 (HEAD -> test) 임시 커밋
| * 4641593 (origin/master, master) master2 커밋
| * 82443bd master 커밋 1
|/
* 4e0bd48 (feature1) 새로운 기능 1 추가

jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (test)
$ git checkout master
Switched to branch 'master'
Your branch is up to date with 'origin/master'.

jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (master)
$ git branch -D test
Deleted branch test (was e2df3c5).

jinu@DESKTOP-255P8M7 MINGW64 ~/Documents/hello-git-cli (master)
$ git log --oneline --graph -n3 --all
* 4641593 (HEAD -> master, origin/master) master2 커밋
* 82443bd master 커밋 1
* 4e0bd48 (feature1) 새로운 기능 1 추가
```

## 4 CLI 로 rebase 해보기

### ❖ 임시 브랜치 사용하기

- 위의 예제는 임시 브랜치인 **[test]** 브랜치를 생성하고 커밋한 후에 다시 **[master]** 브랜치로 돌아가서 **[test]** 브랜치를 삭제한 결과입니다.
- 최종적으로 보이는 것처럼 아무 작업도 남지 않았습니다.
- **commit, merge, rebase** 등 다양한 작업을 미리 테스트해 보고 싶을 때 간단하게 임시 브랜치를 만들어서 사용하고 불필요해지면 삭제하는 것은 좋은 **Git** 활용 팁입니다.
- 이번 장에서는 이를 이용해서 브랜치 생성하기, 병합, **rebase** 등 브랜치와 커밋 관리에 대한 주요 기능을 살펴보았습니다.
- **CLI**든, **GUI**든 결국 **Git**의 동작 원리를 정확하게 이해한다면 사용이 크게 다르지 않습니다.
- **Git**을 익숙하게 사용하기 위해서는 브랜치와 커밋의 개념을 잘 이해하고 상황에 맞춰서 이를 조작하는 연습을 많이 해야 합니다.



**Thank You !**