GIT

ZeroPage 23기 최다인

CONTENTS

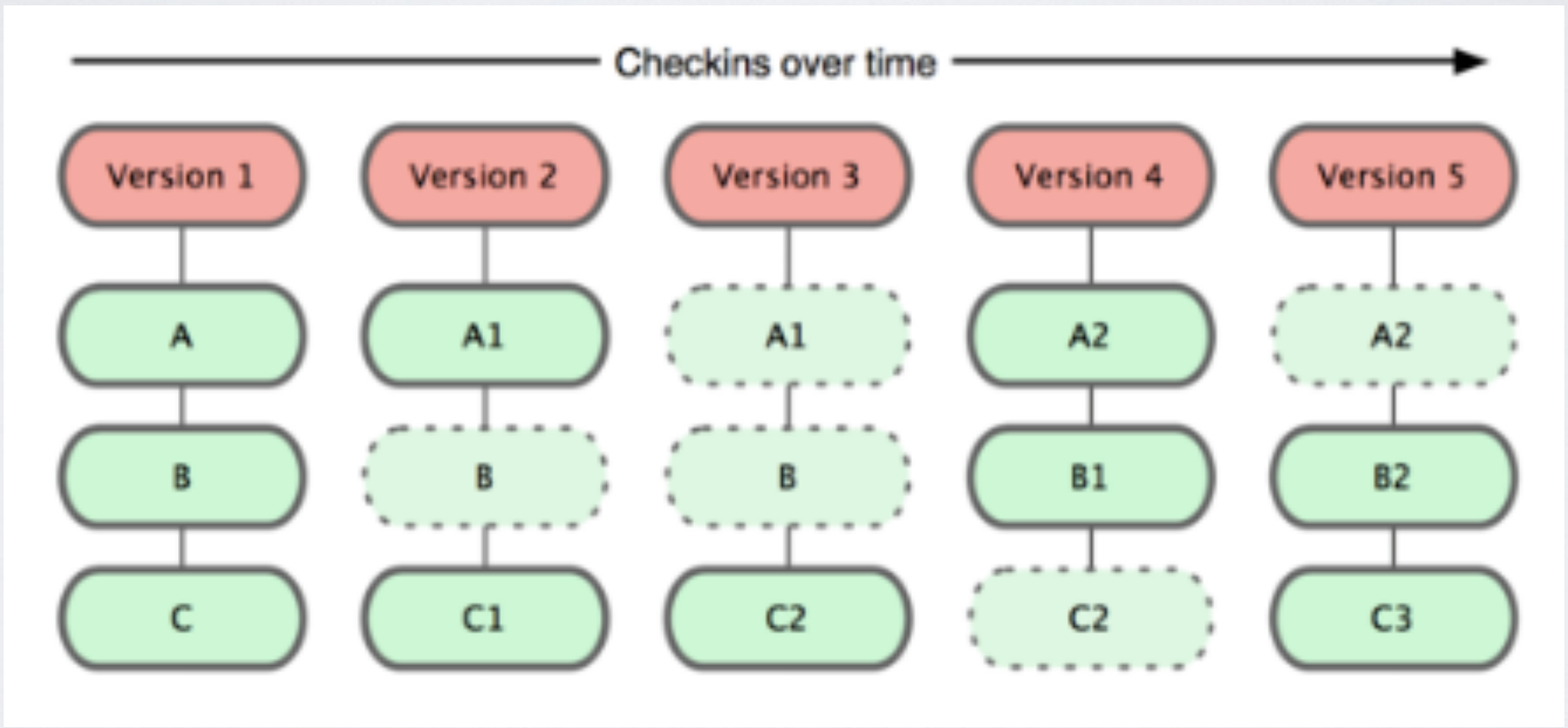
GIT이란 무엇인가?

- 버전 관리 시스템 (Version Control System)

- 분산형 VCS

GIT이란 무엇인가?

- 데이터를 Snapshot으로 저장

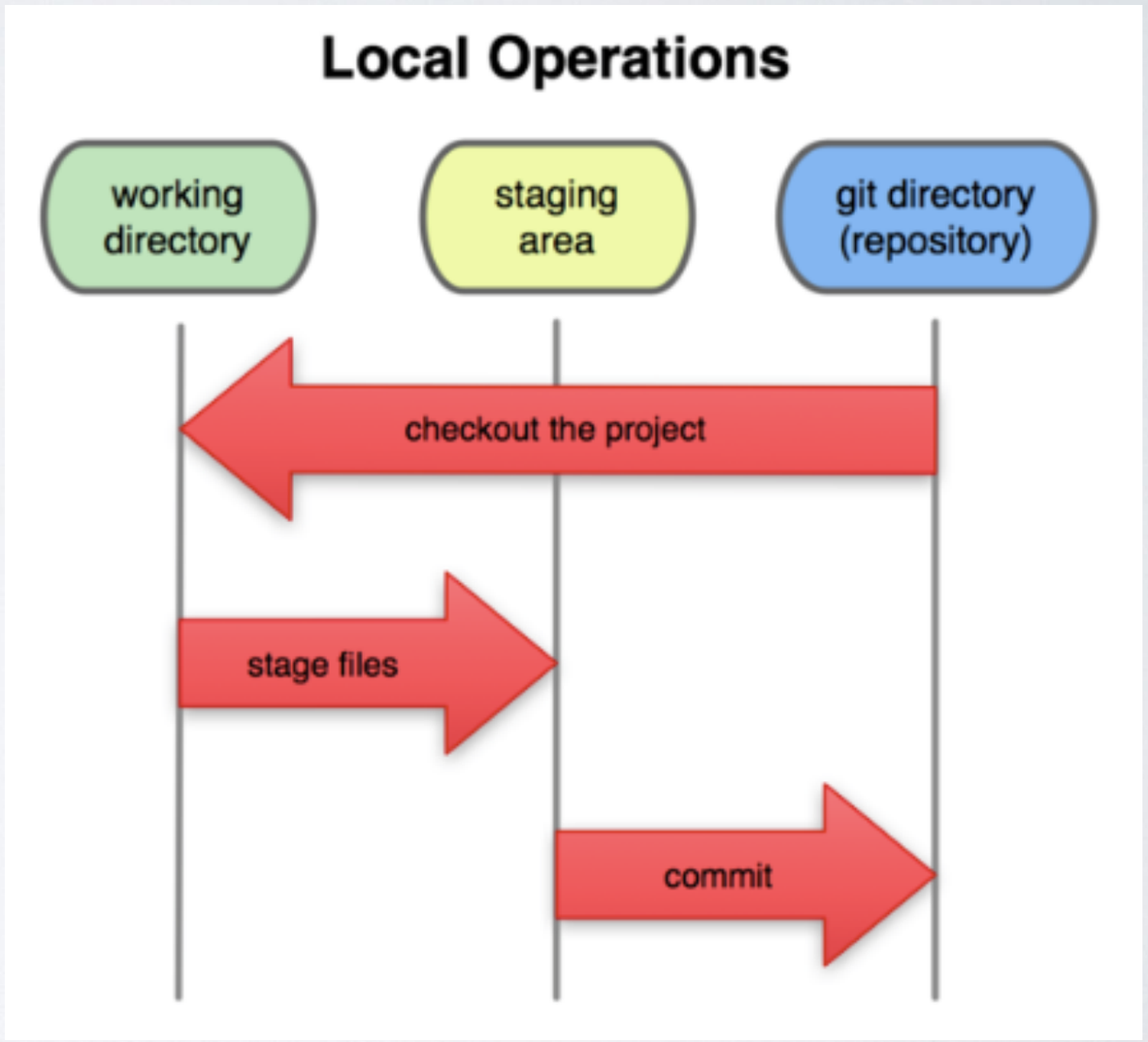


GIT이란 무엇인가?

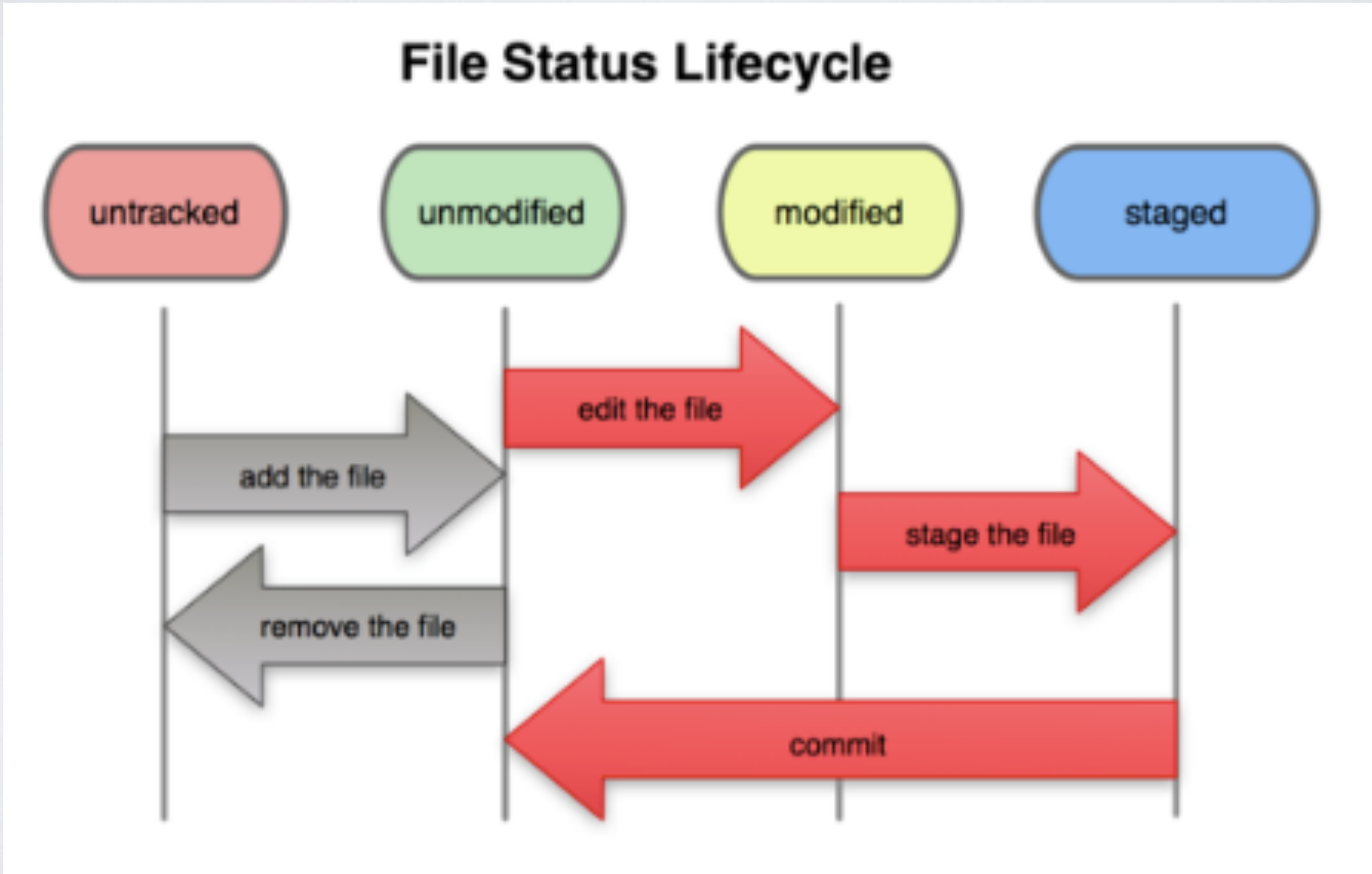
- 대부분의 명령을 로컬에서 실행
- Git의 무결성
  - SHA-1 Hash 사용
  - 파일을 이름으로 저장하지 않음
- Git은 데이터를 추가할 뿐

LOCAL OPERATIONS

- Working Directory에서 파일을 수정한다.
- Staging Area에 파일을 Stage 해서 Commit할 Snapshot을 만든다.
- Staging Area에 있는 파일들을 Commit해서 Git Directory에 영구적인 Snapshot으로 저장한다.



FILE STATUS LIFECYCLE



GIT 기초

- 저장소 만들기
- 파일의 변경
- 히스토리 조회
- 작업의 취소

```
* Nerrtica, Fri Jan 16 14:41:51 2015 +0900 - re
|  MemeticAlgo.xcodeproj/project.pbxproj |  16
|  MemeticAlgo/AvgCHI.c                  | 100
|  MemeticAlgo/Filter.c                  | 100
|  MemeticAlgo/StandardHeader.h          |   4
|  MemeticAlgo/Wrapper.c                 | 221
|  MemeticAlgo/algorithm.c               | 221
|  6 files changed, 331 insertions(+), 331 dele
Nerrticaui-Air:MemeticAlgo nerrtica$ git reset
HEAD is now at 601d5e7 implement label powerset
Nerrticaui-Air:MemeticAlgo nerrtica$ git diff
diff --git a/MemeticAlgo/Filter.c b/MemeticAlgo
index 24ca336..be05ace 100644
--- a/MemeticAlgo/Filter.c
+++ b/MemeticAlgo/Filter.c
@@ -31,9 +31,9 @@ void avgchi () {
     }

     //please modify condition statement to pri
-    /*for (i = 0; avg[i][0] >= 4; i++) {
+    for (i = 0; avg[i][0] >= 4; i++) {
         printf("%2d순위  : %2d번째  feature, Avg
-    }*/
+    }
 }

 void lpchi () {
diff --git a/MemeticAlgo/main.c b/MemeticAlgo/m
index 58bfa07..883388f 100644
--- a/MemeticAlgo/main.c
+++ b/MemeticAlgo/main.c
@@ -21,8 +21,8 @@ int main(int argc, const char
         randomizeData();                        //
     }

-    //avgchi();
```

GIT 저장소 만들기

- 이미 있는 디렉토리에서 새 저장소 만들기

```
$ git init
```

- 이미 있는 저장소를 Clone하기

```
$ git clone [url]
```

파일의 상태 확인하기

```
$ git status
On branch master
nothing to commit, working directory clean
```

- status 명령어로 파일의 상태를 확인할 수 있다.
- 현재 master 브랜치에 위치
- 파일을 하나도 수정하지 않았음

파일의 상태 확인하기

```
$ git status
On branch master

Untracked files:
    (use "git add <file>..." to include in what will be committed)
        README
nothing added to commit but untracked files present
```

새 파일을 추적하기

```
$ git add <file>
```

- git add README 를 실행할 경우,

```
$ git status
On branch master

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

파일 변경 후 STAGE하기

- README 파일을 수정했을 경우,

```
$ git status
On branch master

Changes not staged for commit:
    (use "git add <file>..." to update what will be committed)
    (use "git checkout -- <file>..." to discard changes ... )
            modified:     README
```

변경사항 커밋하기

```
$ git commit
```

```
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
#
# Initial commit
#
# Changes to be committed:
#       new file:       README
#
```

변경사항 커밋하기

- 커밋 메시지를 한 줄로 간단하게 적을 때

```
$ git commit -m “commit message”
```

- 뭐가 달라졌는지에 대한 내용도 추가하고 싶을 때

```
$ git commit -v
```

STAGING AREA 생략하기

```
$ git commit -a
```

- Tracked 상태의 파일을 모두 자동으로 Staging Area 에 넣은 후 Commit을 실행한다.

- git add 명령어를 수행하는 수고를 덜 수 있다.

파일 삭제하기

```
$ git rm <file>
```

- 실제 파일은 지우지 않고 추적만 종료하려면,

```
$ git rm --cached <file>
```

- 이후 .gitignore에 추가하여 track을 막을 수 있다.

파일 이름 변경하기

```
$ git mv <file_from> <file_to>
```

- 사실은 아래와 같다.

```
$ mv <file_from> <file_to>
$ git rm <file_from>
$ git add <file_to>
```

히스토리 조회

```
$ git log
```

- log 명령어의 옵션은 굉장히 많다.

- 참고 페이지 : 커밋 히스토리 조회

```
$ git reflog
```

작업의 취소

- 다시 커밋하기

```
$ git commit --amend
```

- 어떤 파일을 빼먹었거나, 메시지를 잘못 적었을 때

- 두번째 커밋이 첫번째 커밋을 덮어씀

작업의 취소

```
$ git status
On branch master

Changes to be committed:
    (Use “git reset HEAD <file>…” to unstage)
            modified:     README
```

```
Changes not staged for commit:
    (Use “git checkout -- <file>…” to unstage)
            modified:     README
```

작업의 취소

- 이전 커밋으로 돌아가기

```
$ git reset HEAD~<n>
```

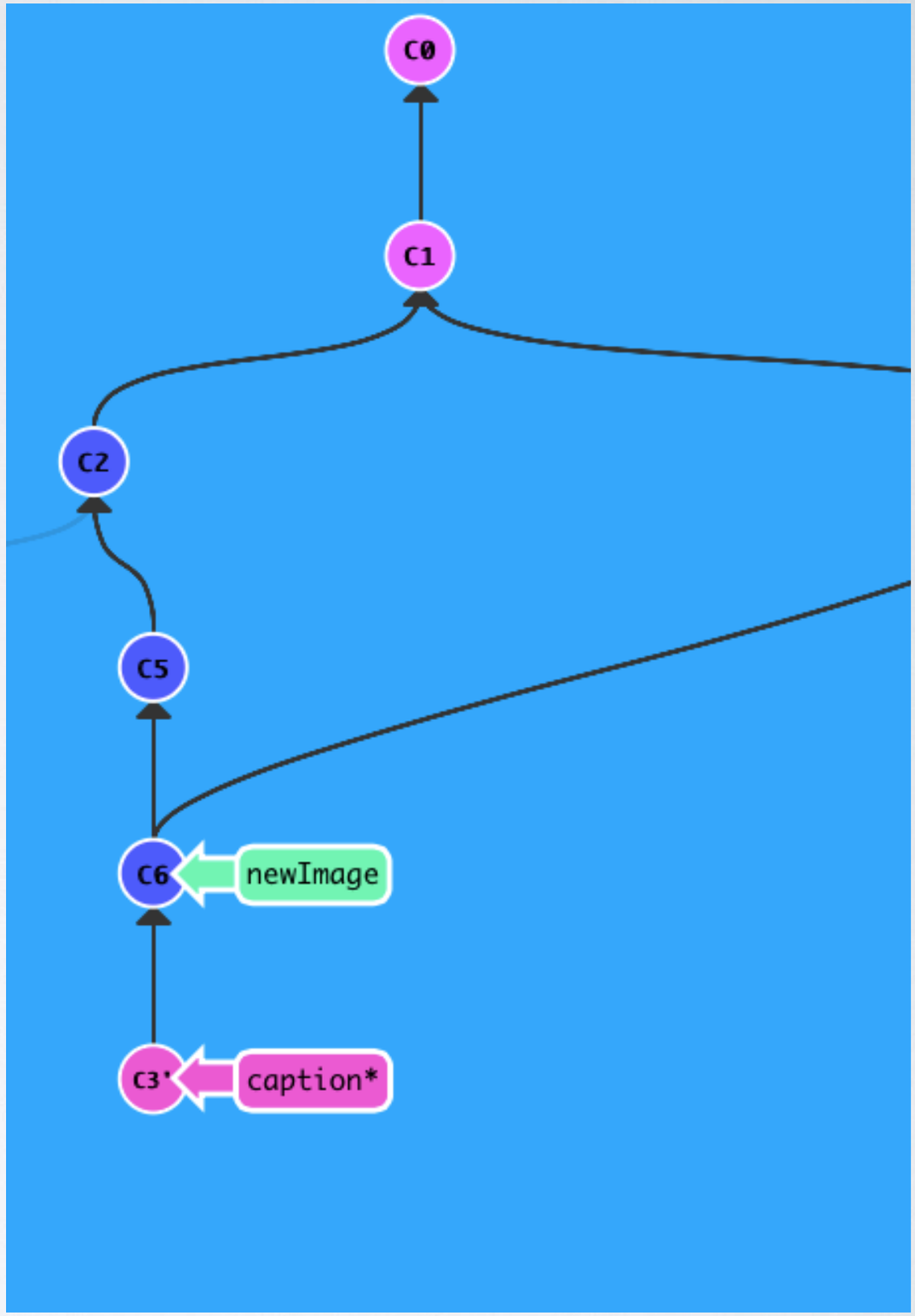
- n개 전의 커밋으로 돌아간다.

```
$ git revert HEAD~<n>
```

- 커밋을 취소하는 커밋을 새로 발행한다.

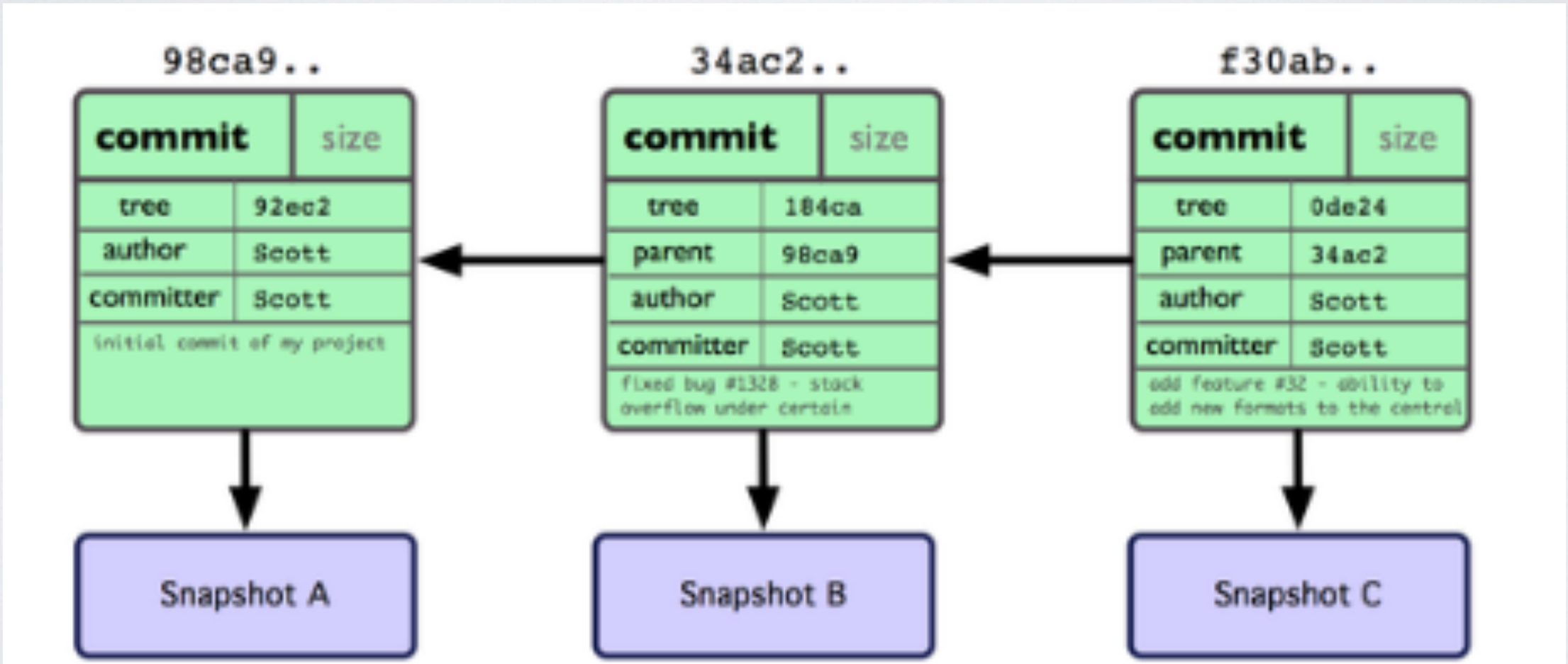
GIT 브랜치

- 브랜치란?
- 브랜치 관리
- 브랜치 합치기
- 충돌



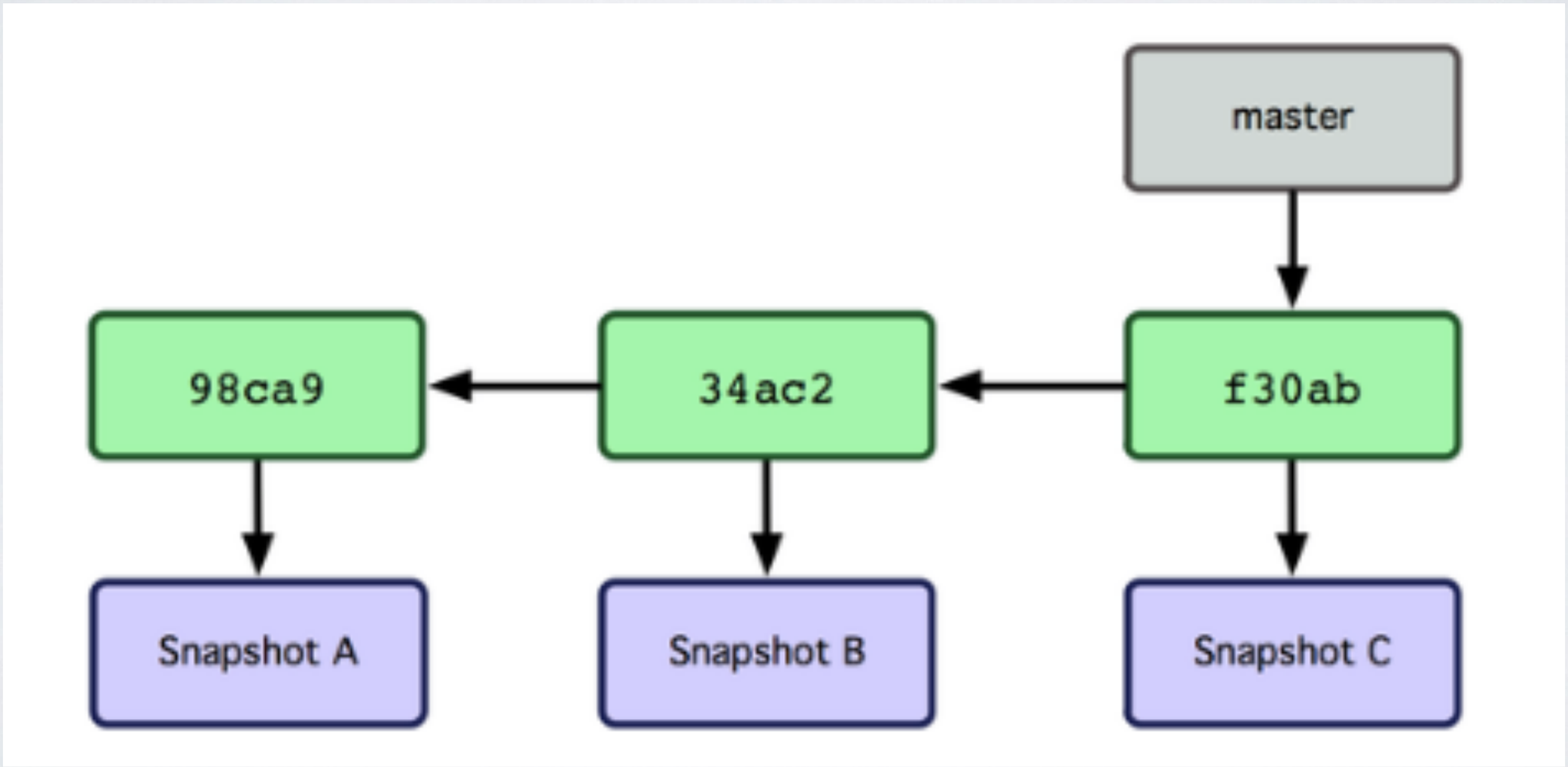
브랜치란?

- Git이 데이터를 저장하는 방식은 아래와 같음



브랜치란?

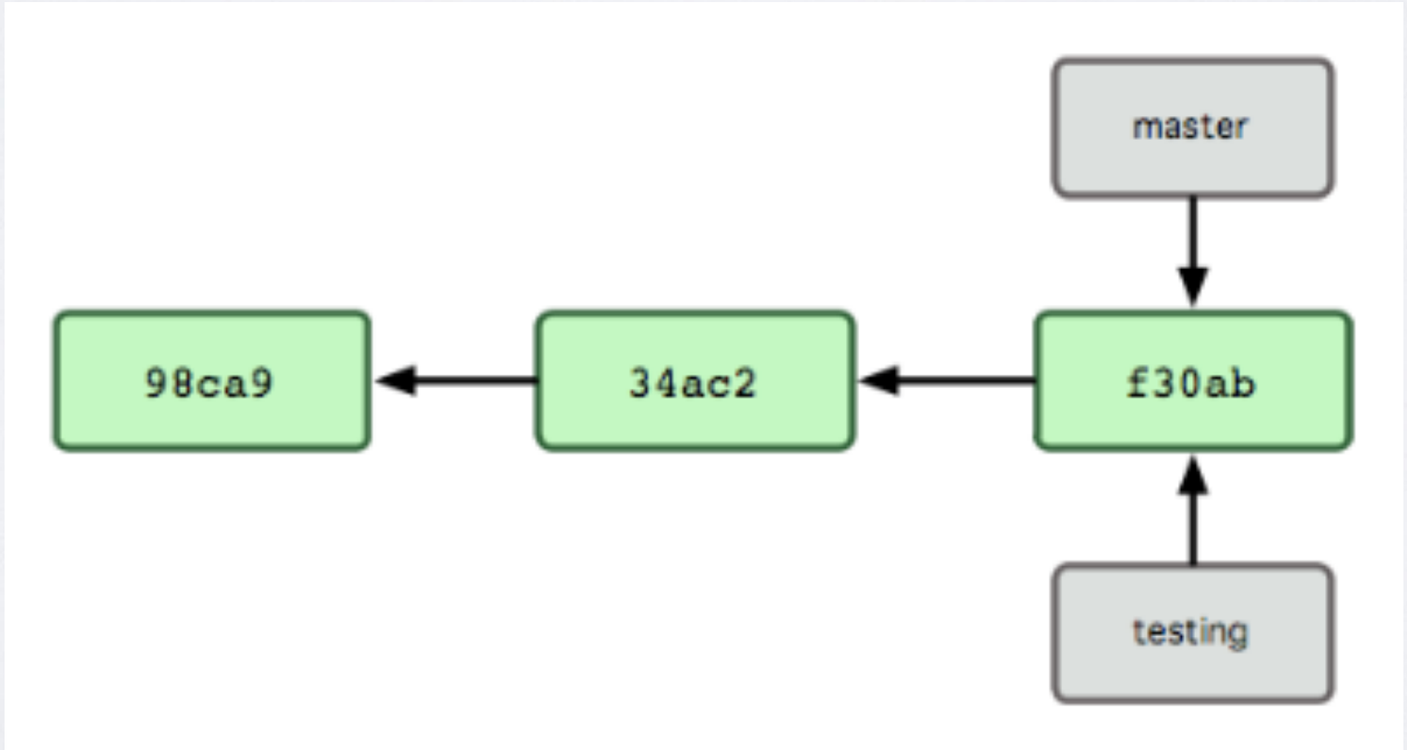
- 브랜치란, 커밋 사이를 이동할 수 있는 일종의 포인터



브랜치 생성

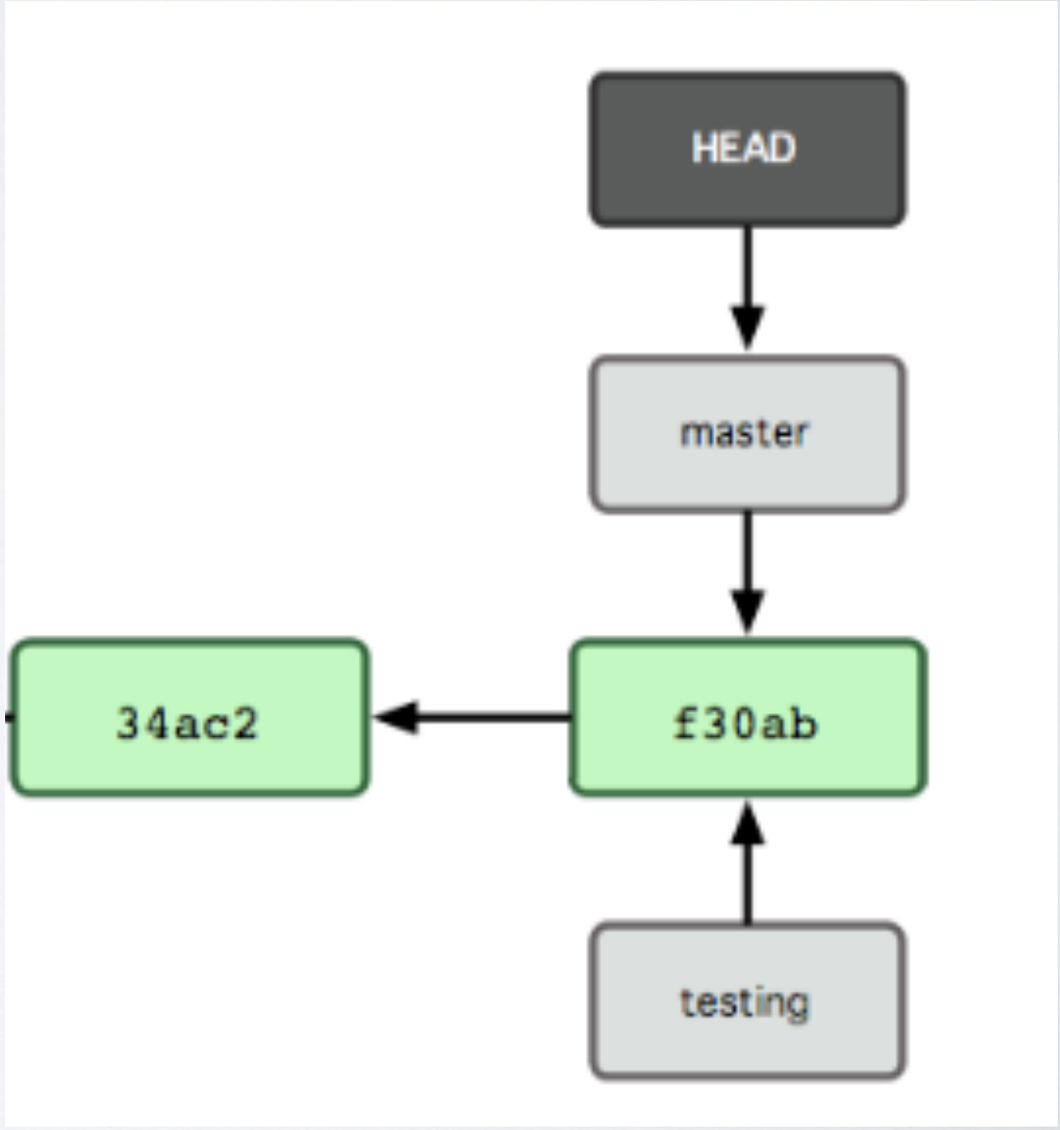
```
$ git branch testing
```

- 새로 만든 브랜치는 작업하고 있던 마지막 커밋을 가리킴



브랜치 생성

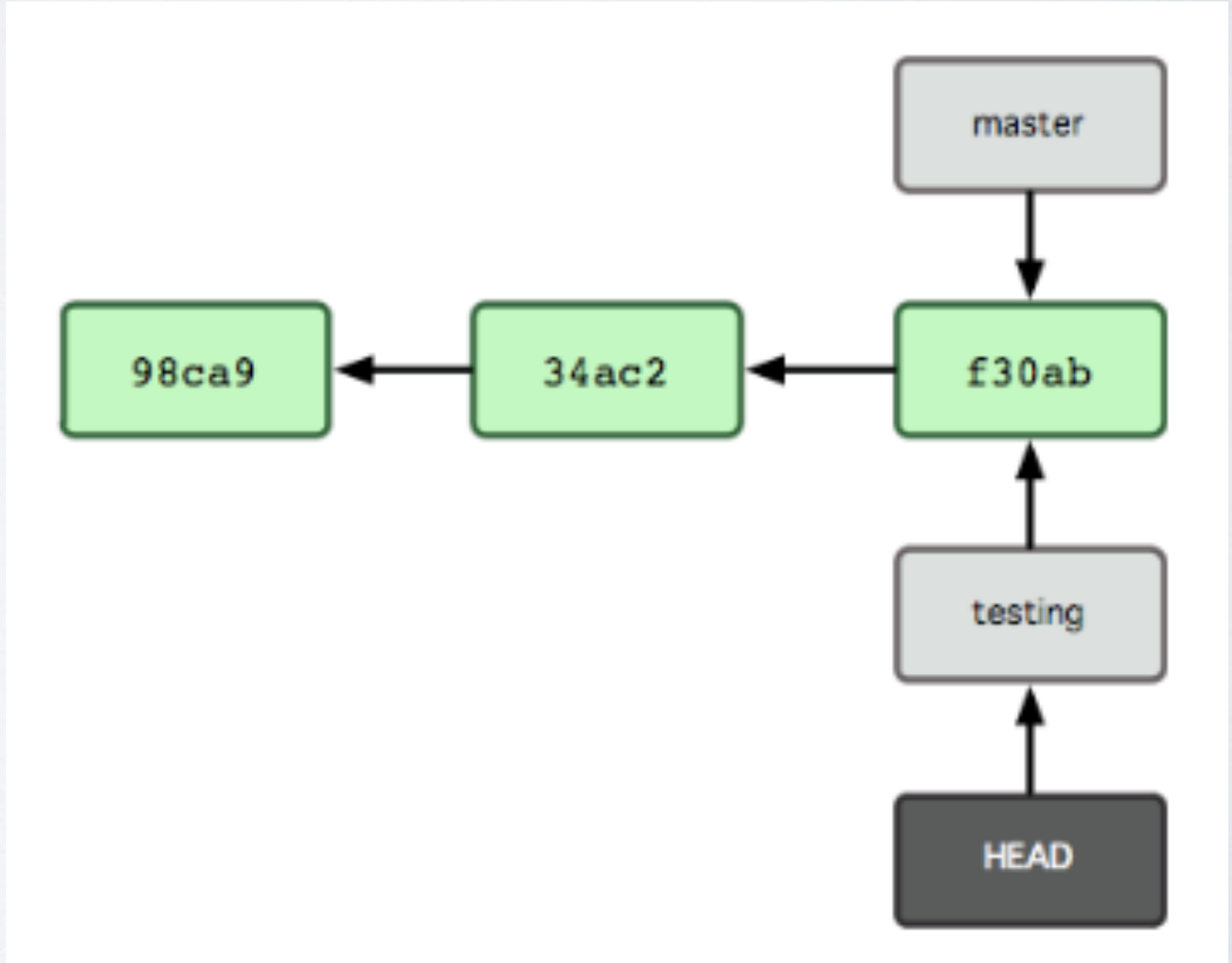
- 지금 작업중인 브랜치는 어떻게 알까?
- HEAD라는 특수한 포인터가 지금 작업하는 로컬 브랜치를 가리킨다.
- git branch 명령은 브랜치를 만들기만 하고 옮기지는 않는다.



브랜치 이동

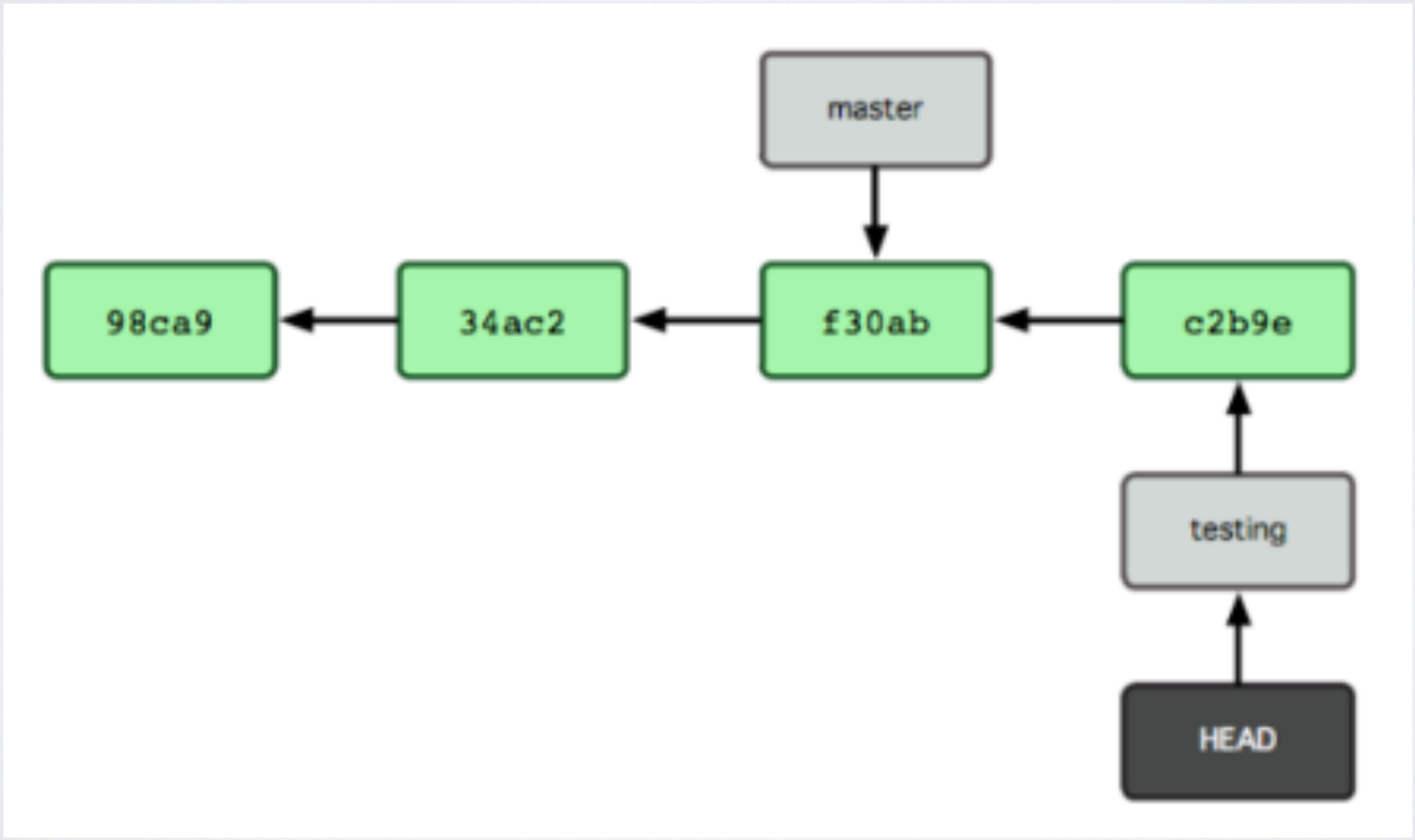
```
$ git checkout testing
```

- testing 브랜치로 HEAD가 이동



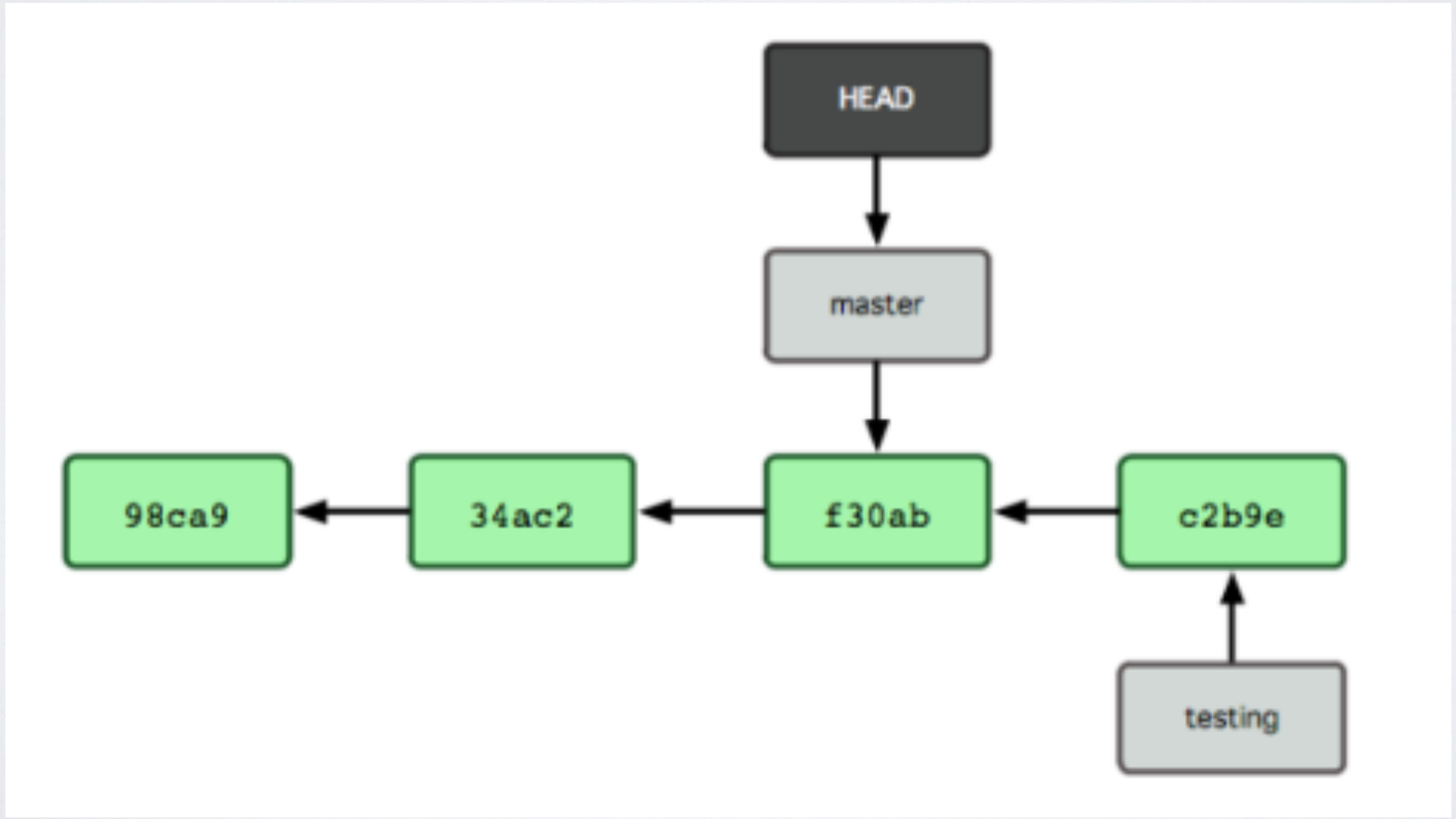
브랜치 이동

- 이후 파일을 수정하고 커밋을 새로 하면,



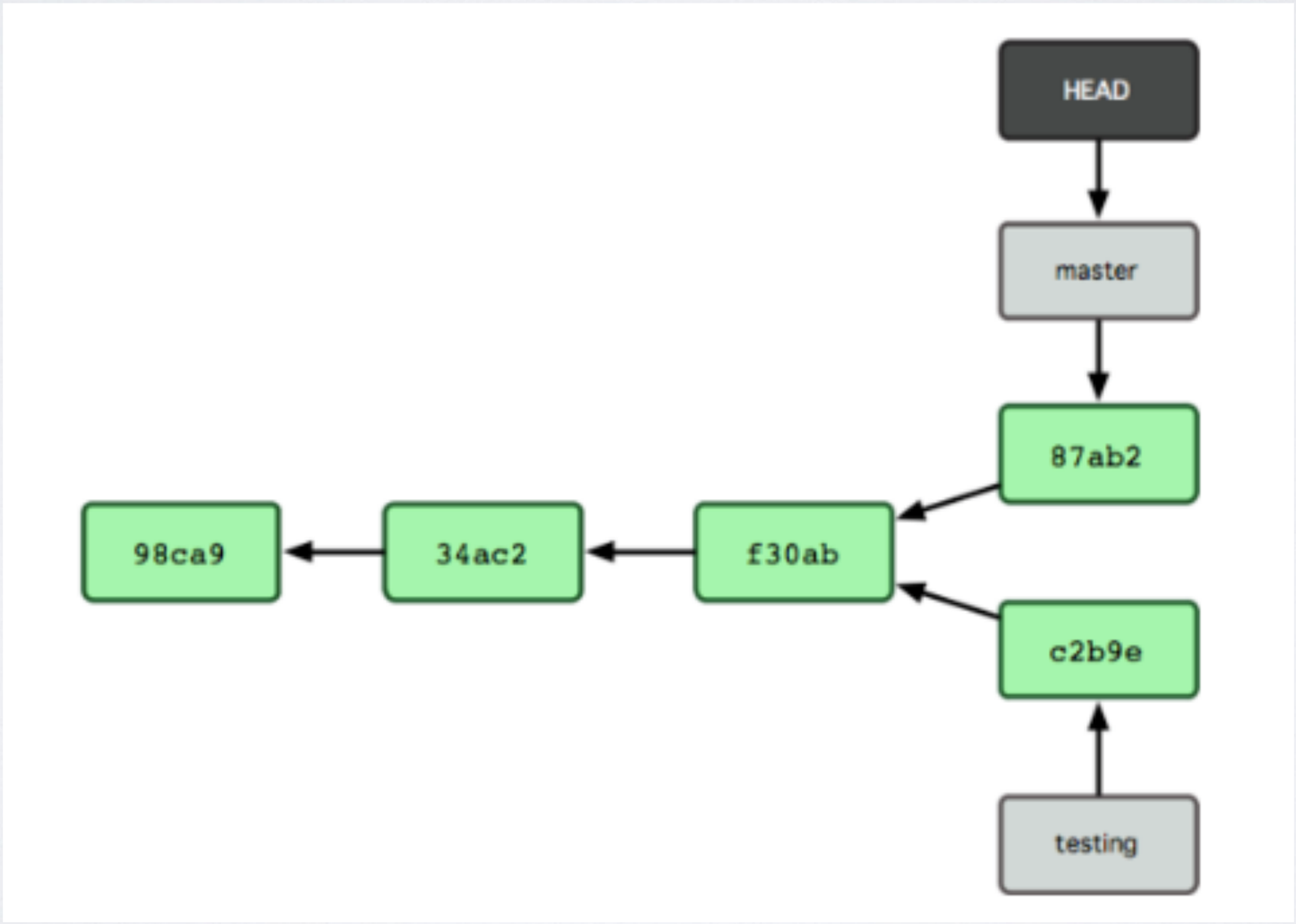
브랜치 이동

```
$ git checkout master
```



브랜치 이동

- 다시 한 번 파일을 수정하고 커밋



브랜치 합치기

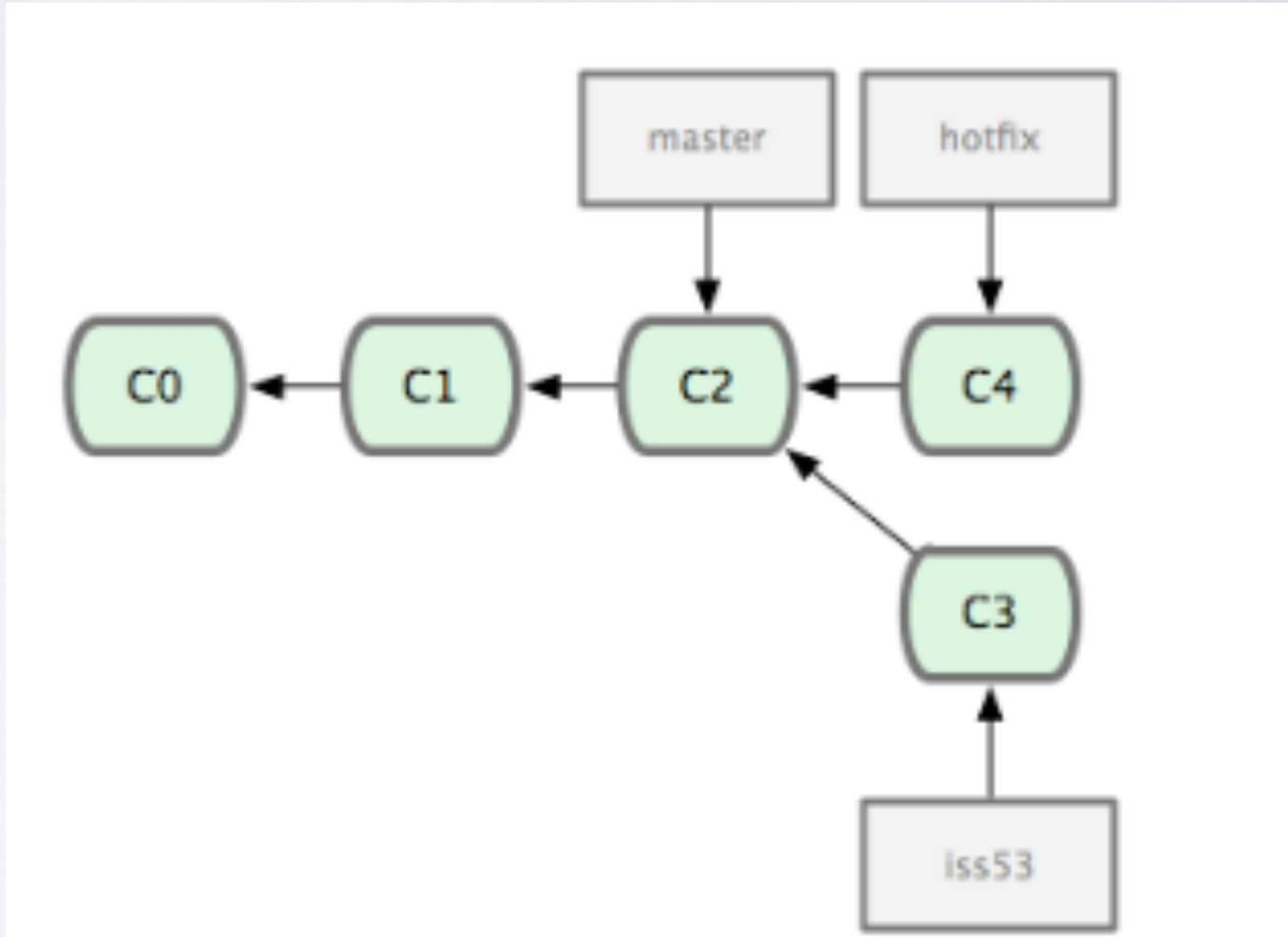
```
$ git merge testing
```

- 현재 브랜치에서 testing 브랜치로 merge
- 두 종류의 merge 방식이 존재
  - Fast forward & 3-way merge

브랜치 합치기

- master와 hotfix를 합친다면, Fast forward

- hotfix와 iss53을 합친다면, 3-way merge

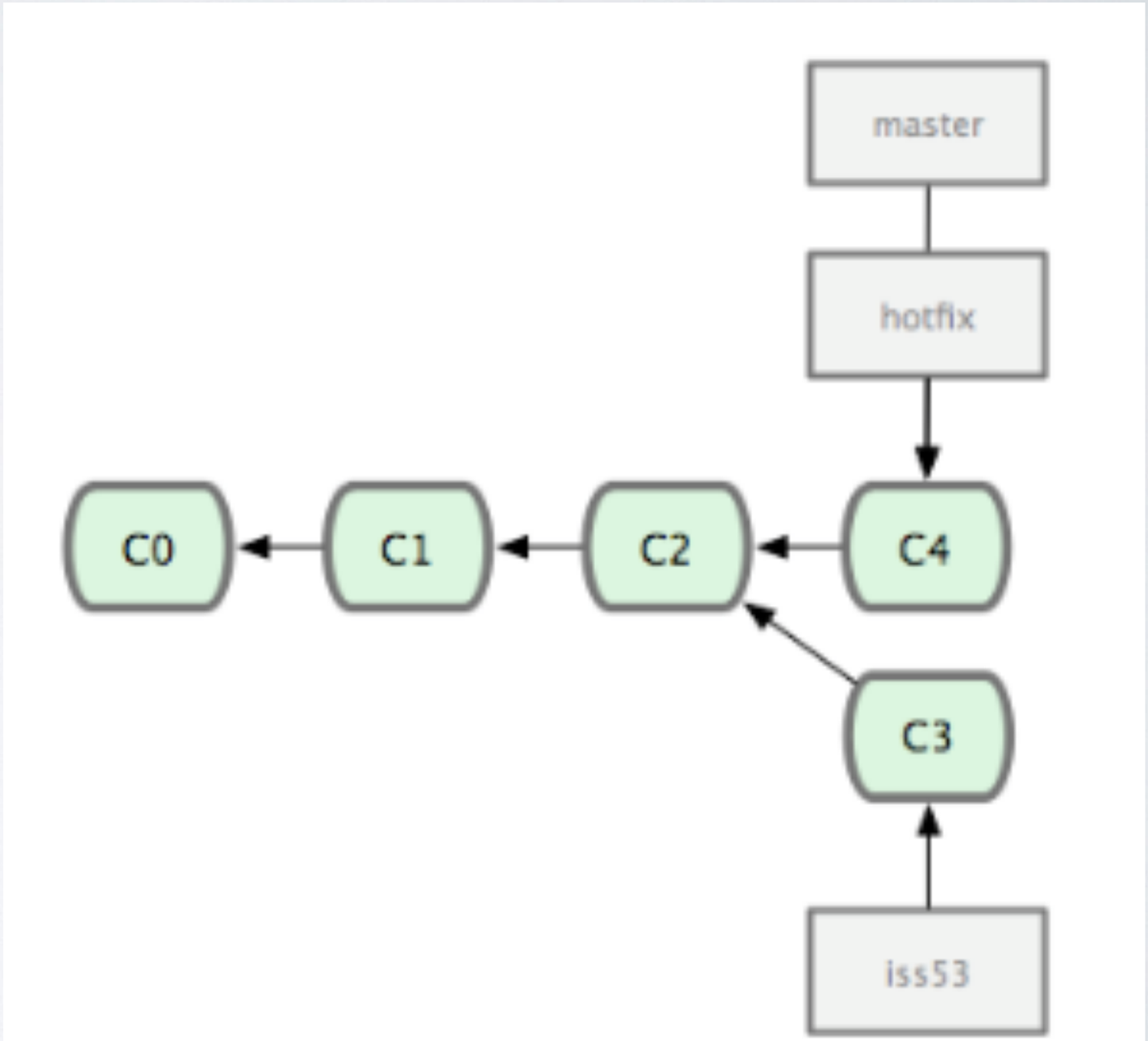


FAST FORWARD

```
$ git checkout master
```

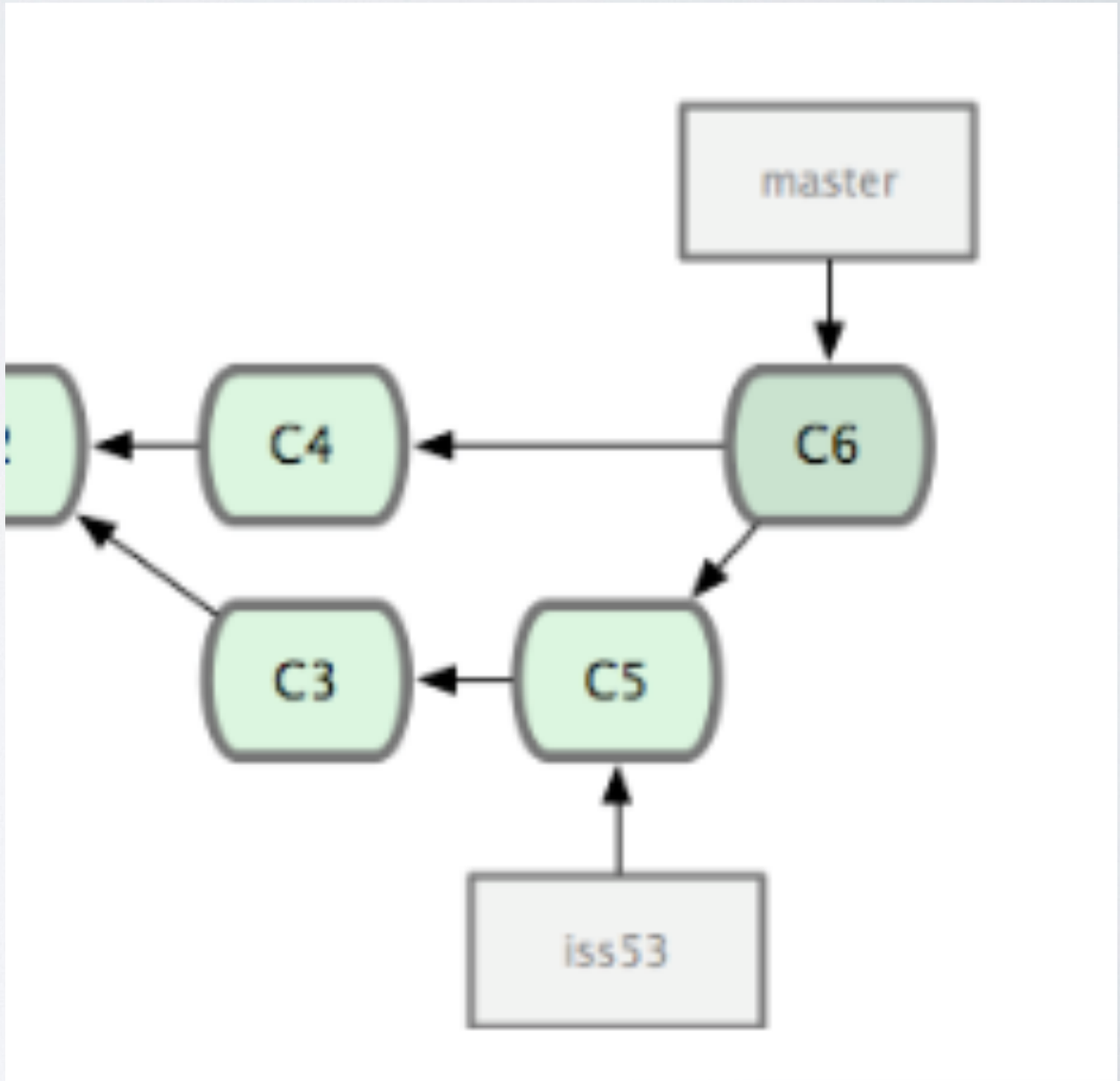
```
$ git merge hotfix
```

```
Updating …
Fast forward
…
```



3-WAY MERGE

- Git은 Merge할 때 Merge에 대한 정보가 들어있는 커밋을 하나 더 만듬
- C6는 Merge Commit



브랜치 합치기

```
$ git branch -d testing
```

- Merge가 끝나 필요없어진 브랜치는 삭제한다

```
$ git branch -D testing
```

- Merge하지 않았지만, 필요없다고 판단되는 브랜치는 -D 명령어로 강제 삭제한다

충돌

- Merge하는 두 브랜치에서 같은 파일의 한 부분을 동시에 수정하고 Merge하면, Git은 해당 부분을 Merge하지 못함

```
$ git merge testing
Auto-merging README
CONFILCT (content): Merge conflict in README
Automatic merge failed; fix conflicts and then commit the result.
```

충돌

```
$ git status
on branch master
You have unmerged paths.
    (fix conflicts and run “git commit”)

Unmerged paths:
    (use “git add <file>…” to mark resolution)

            both modified:     README

no changes added to commit
```

충돌

```
<<<<<<< HEAD
blabla…
=======
…blabla
>>>>>>> testing
```

- 각 브랜치의 내용이 담겨 있음
- 위와 아래 중 고르거나, 새로 작성

충돌

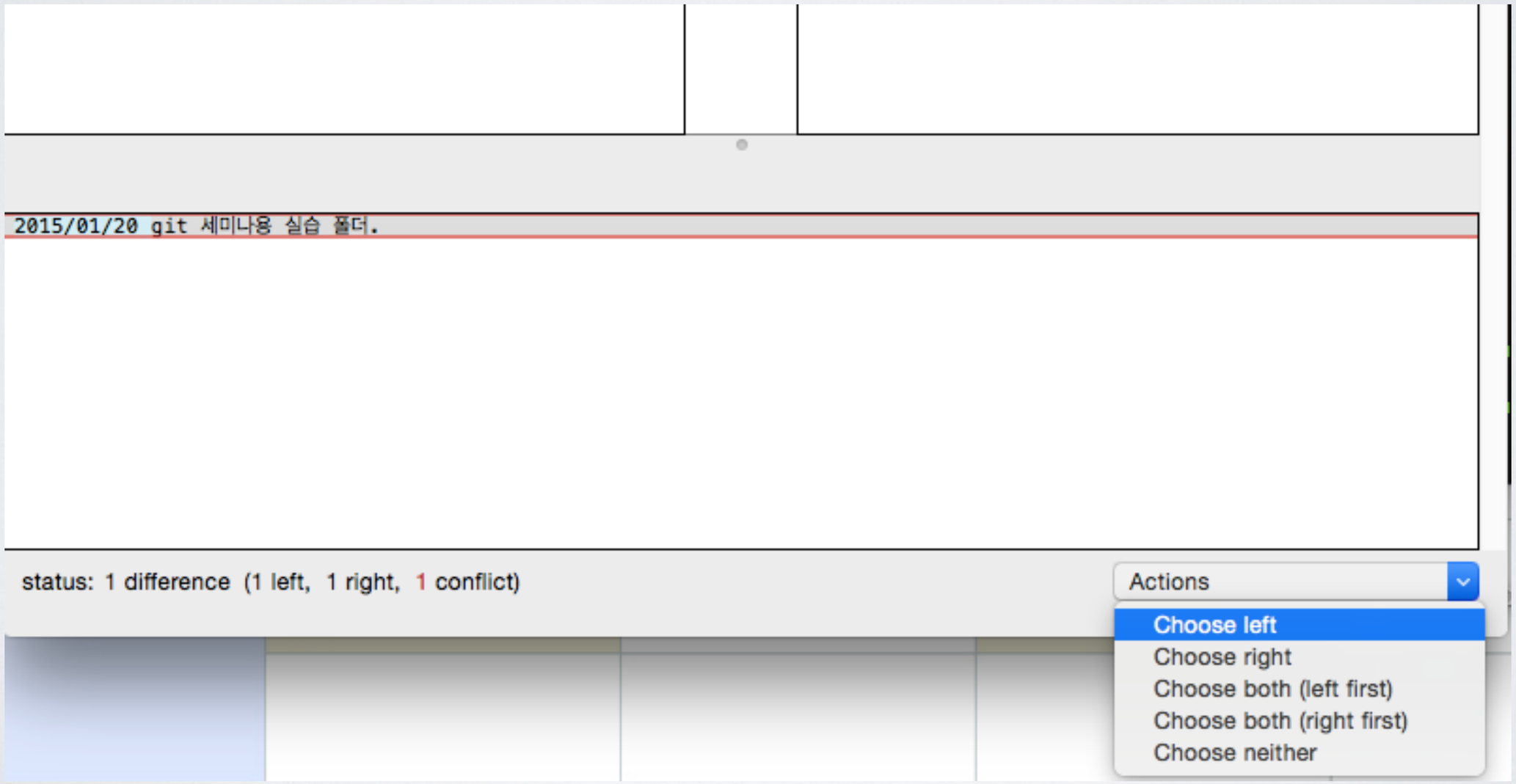
- 충돌한 부분을 해결했다면 git add 명령으로 다시 Git에 저장한 후 커밋
- 충돌을 쉽게 해결하기 위해 다른 tool도 사용 가능

```
$ git mergetool
```

- 본인의 컴퓨터에 툴이 깔려있는 경우에 사용

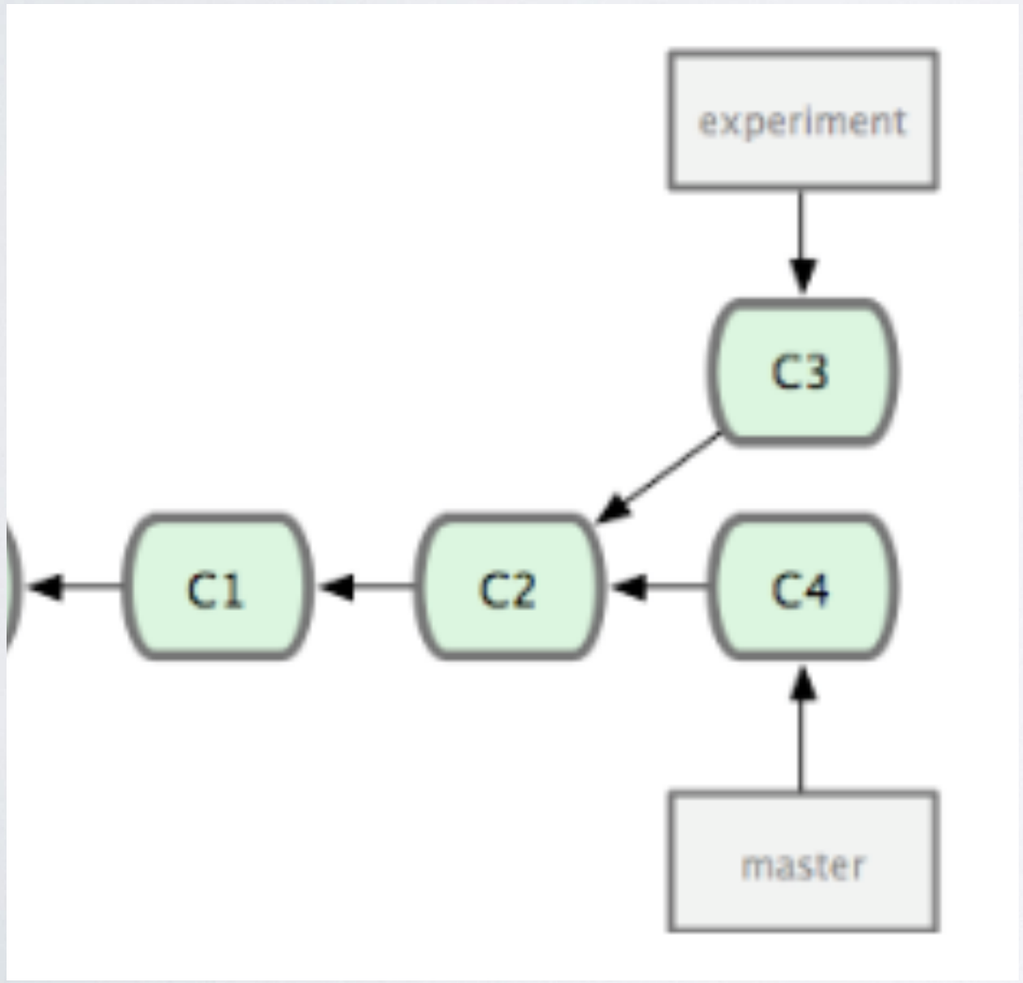
충돌

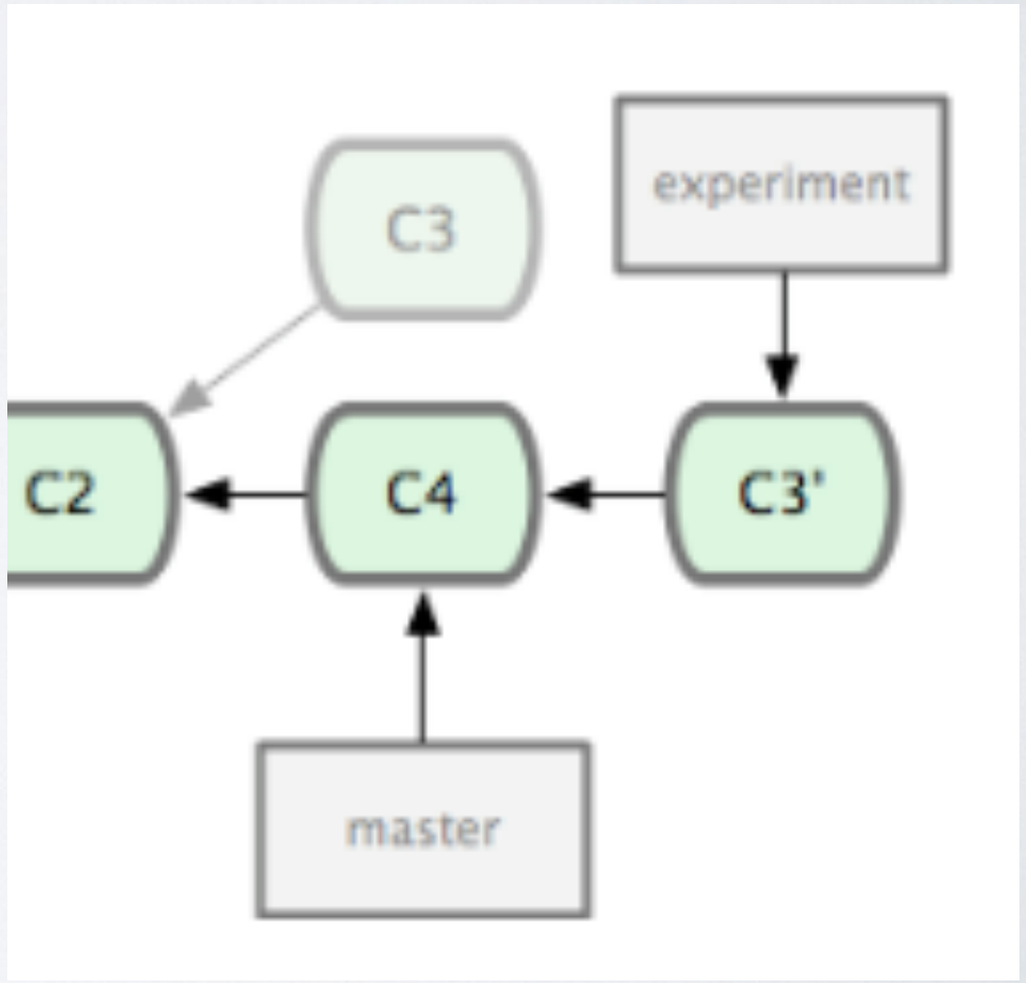
- Mac 기본 내장 opendiff 사용 예시



브랜치를 합치는 또 다른 방법

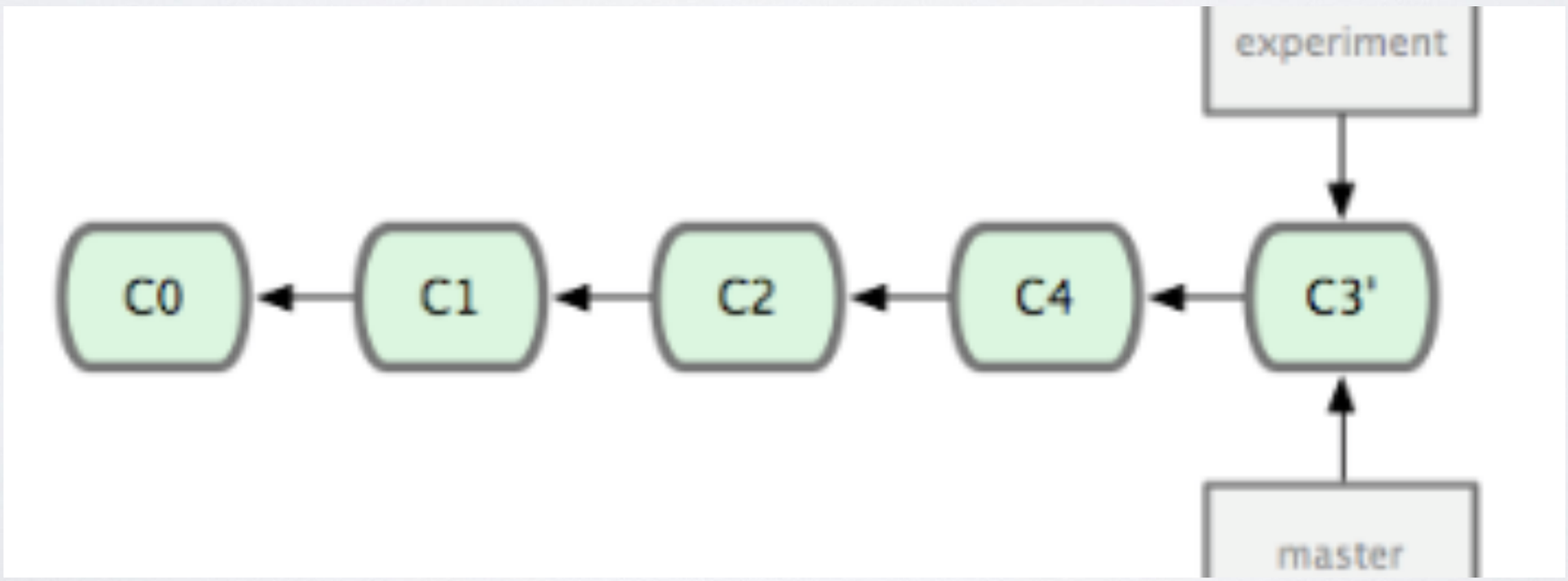
```
$ git rebase master
```





REBASE

- Rebase를 하든지, Merge를 하든지 최종 결과물은 같음
- 커밋 히스토리가 선형적으로 나오게 됨



REBASE

- Rebase의 위험성
  - 공개 저장소에 Push한 커밋을 Rebase하지 마라
- Rebase는 기존의 커밋을 사용하지 않고 내용은 같지만 다른 커밋을 새로 만듬

GIT 공부에 도움이 되는 사이트

- https://github.com : Git 서버 제공
- http://git-scm.com/book/ko/v1 : Progit 책
- http://learnbranch.urigit.com : 브랜치 사용 연습

감사합니다