

Git (brench)

📎 자료	<u>DB</u>
☰ 구분	DB
≡ 과목	

Branch

Branch란?

나무가지 처럼 여러갈래로 작업 공간을 분리해서

독립적으로 작업을 할 수 있게 도와주는 git 도구이다.

왜 사용해야 할까?

가장 큰 이유는 개발시 원본에 대한 안전성을 확보하기 위해서다.

예를 들어 서비스가 운영중인 상태에서 버그가 발견 되었을 때,

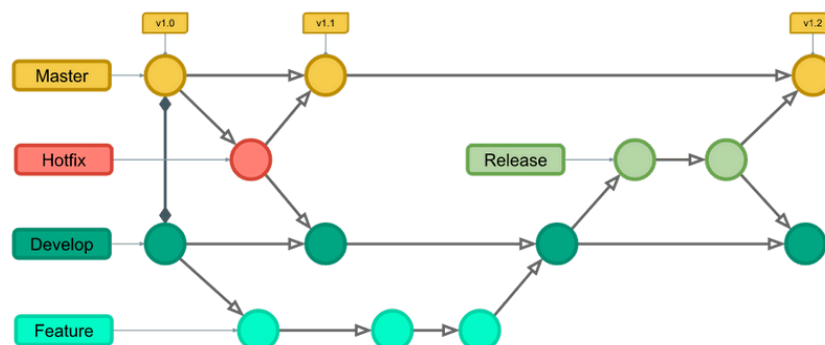
새로운 브랜치를 생성해서 버그를 수정하면 운영중인 서비스에 영향을 주지 않게 된다.

또 다른 이유는 협업에 도움이 된다.

한 프로젝트의 파트를 나눠 팀원들이 독립적으로 작업을 수행 할 수 있다.

뿐만아니라 운영되고 있는 서비스에 새로운 기능을 추가할 때에도

브랜치를 만들어 독립적으로 작업하고 완성이 된 후에 메인 브랜치에 병합해서 배포할 수 있다.



- Master banch : 제품으로 출시할 수 있는 브랜치로 상용화가 가능한 상태를 의미한다.
- Develop branch : 다음 버전 출시를 위한 준비를 하는 것이다.
 - 예를들어 새로운 UI 적용 및 작업환경 속도 개선으로 UX 경험 증가
- Feature branch : 다음 버전에 추가로 탑재 할 새로운 기능을 개발하는 branch
- Hotfix : 긴급 버그 수정

=====

Branch를 실습해보자.

Branch를 생성하고 그리고 병합하는 것을 연습 할 것이다.

먼저 사전셋팅 부터 해보자.

1. test.txt 를 생성하고 각각 master-1, master-2 그리고 master-3 이라는 내용을 순서대로 입력하면서 커밋을 총 3개를 작성한다.

```
$ touch test.txt

# 그 다음에는 test.txt 파일 안에 "master-1" 라고 작성합니다. 그 다음

$ git add .
$ git commit -m "master-1"

# test.txt 파일 안에 "master-2" 라고 작성합니다. 그 다음

$ git add .
$ git commit -m "master-2"

# test.txt 파일 안에 "master-3" 라고 작성합니다. 그 다음

$ git add .
$ git commit -m "master-3"
```

3. git log --oneline 을 입력했을 때 아래와 같이 나와야 정상입니다.
총 3개의 버전이 master 브랜치에 만들어졌습니다.

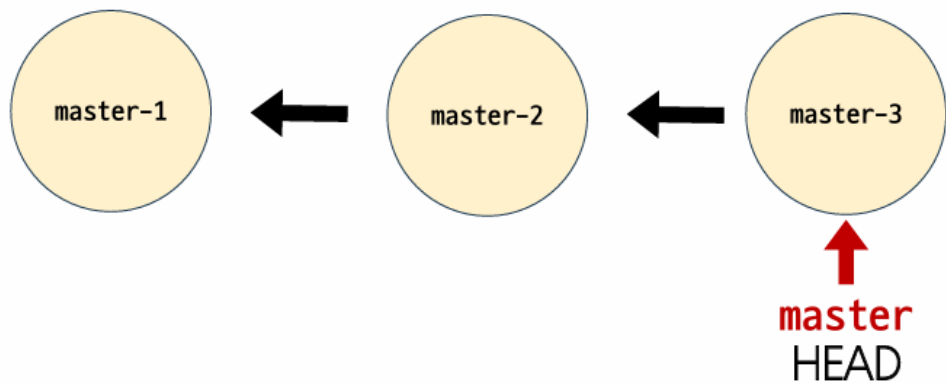
```
$ git log --oneline
```

```
0604dcd (HEAD → master) master-3
```

```
9c22c89 master-2
```

```
3d71510 master-1
```

현재까지의 결과를 그림으로 나타 내자면 다음과 같습니다.,



여기서 HEAD 가 의미하는 것은 지금 작업을 하고 있는 대상을 의미 합니다. HEAD가 가리키는 브랜치는 Master 이다.

위 그림을 보면 커밋 진행 방향과 화살표가 방향이 다르다는 것을 확인 할 수 있다. 이는 커밋을 하면 이전 커밋 이후에 변경사항만 기록한 것으로 새로 생성된 커밋은 이전 커밋에 종속되어 생성된 것이라는 것을 의미한다.

3. login 이라는 이름의 브랜치를 새로 생성 해 보자.

```
$ git branch login
```

그리고 branch가 잘 생성 되었는지 확인 합니다.

```
$ git branch
```

```
SSAFY@DESKTOP-MDJG1NJ MINGW64 ~/Desktop/test (master)
$ git branch
  login
* master
```

브랜치를 확인 하기 위해 git branch 라고 작성했고

현재 *이 가르키는 곳은 master 입니다. 이때 *은 head를 뜻합니다.

즉 작업하고 있는 가지는 Master 라는 뜻이다.

이번에는 \$ git log --oneline 이라고 작성 해 보겠다.

```
$ git log --oneline
```

```
SSAFY@DESKTOP-MDJG1NJ MINGW64 ~/Desktop/test (master)
$ git log --oneline
91ca616 (HEAD -> master, login) master-3
8b69c94 master-2
f9a0c7b master-1
```

마찬가지로

커밋 기록을 보면 head 가 가르키는 곳은 master 이고

login 이라는 branch 가 있다는 것을 확인 할 수 있다.

Master 브랜치에서 1개의 커밋을 더 생성해보자.

test.txt에 master-4 를 추가 작성후 add commit을 하자

```
# test.txt에 master-4 를 추가로 작성 후 add commit 한다.
```

```
$ git add .
```

```
$ git commit -m "master-4"
```

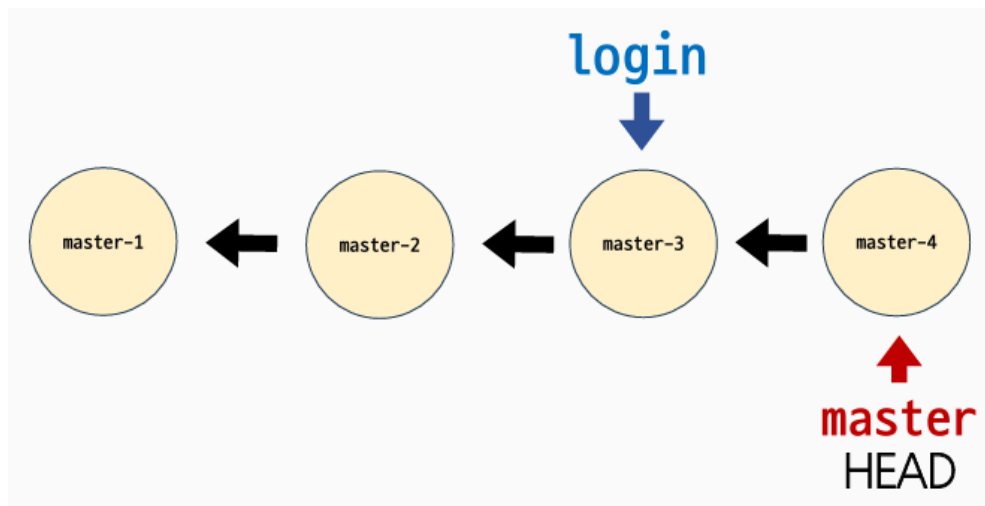
그리고 현재 커밋 상태를 한번 더 확인해 보자.

```
$ git log --oneline
```

참고로 `git log --oneline -5` 라고 작성하면
최근 5개의 커밋 내역만 확인을 하겠다는 코드다.

```
SSAFY@DESKTOP-MDJG1NJ MINGW64 ~/Desktop/test (master)
$ git log --oneline
0913847 (HEAD -> master) master-4
91ca616 (login) master-3
8b69c94 master-2
f9a0c7b master-1
```

지금 상황을 그림으로 살펴 보면 아래와 같다.



우리는

login 브랜치를 master-3 커밋 후 생성을 했고 그 이후
master 브랜치로 master-4 작업 했다.

자 이제 브랜치 이동을 해 보자.

지금 작업대상 Head가 가르키고 있는곳은 master 브랜치 이다.

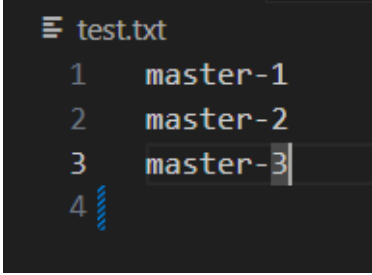
이제 Master 브랜치에서 → login 브랜치로 이동해 보자.

브랜치 이동은 switch 명령어를 사용하면 됩니다.

```
$ git switch login
```

```
SSAFY@DESKTOP-MDJG1NJ MINGW64 ~/Desktop/test (master)
$ git switch login
Switched to branch 'login'
```

이동을 하고 vscode 에 test.txt 파일을 확인해 보면
방금 만든 master-4 는 보이지 않는다.



```
test.txt
1  master-1
2  master-2
3  master-3
4
```

\$ git log --oneline 으로 지금 log 상황을 살펴보자.

```
git log --oneline
```

```
SSAFY@DESKTOP-MDJG1NJ MINGW64 ~/Desktop/test (login)
$ git log --oneline
91ca616 (HEAD -> login) master-3
8b69c94 master-2
f9a0c7b master-1
```

커밋 이력을 보면 master 브랜치가 보이지 않습니다. 그 이유는
지금 현재 head가 가르키는 대상이 login 브랜치이기 때문이다.

Master 브랜치가 사라질 리가 없으므로 git branch를 통해서 지금 branch 상황을 확인해 보
자.

```
$ git branch
```

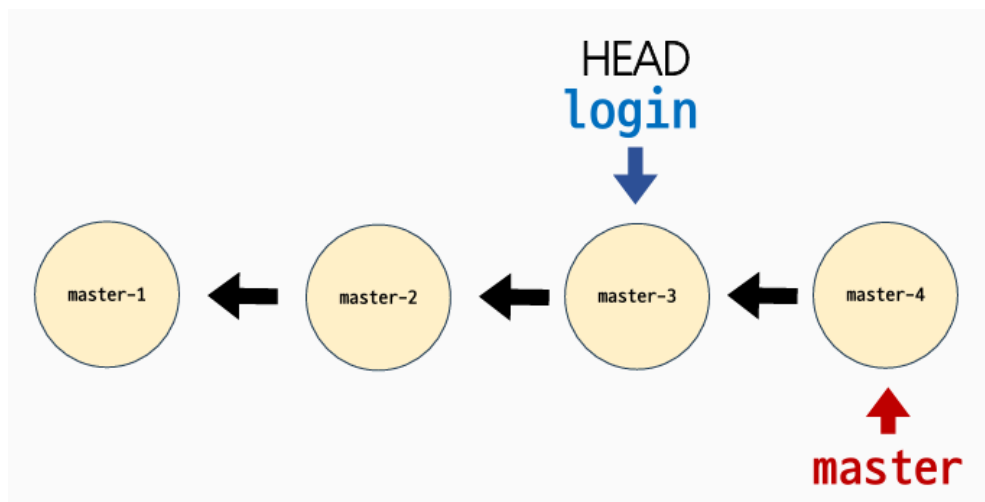
```
SSAFY@DESKTOP-MDJG1NJ MINGW64 ~/Desktop/test (login)
$ git branch
* login
master
```

*은 head 이며, head는 현재 작업 대상을 의미한다고 했다.

현재 head가 가르키는 곳은 login 이며,

Master 브랜치도 잘 있음을 눈으로 확인이 된다.

지금 상황을 그림으로 보자면 다음과 같을 것입니다.



이번에는 \$ git log --oneline 에 --all 옵션을 더해서

모~~든 브랜치의 커밋이력을 확인해 보자.

```
$ git log --oneline --all
```

```
SSAFY@DESKTOP-MDJG1NJ MINGW64 ~/Desktop/test (login)
$ git log --oneline --all
0913847 (master) master-4
91ca616 (HEAD -> login) master-3
8b69c94 master-2
f9a0c7b master-1
```

지금 head는 login을 가르키고 있다.

이제 login 브랜치에서 커밋을 생성 해보자.

test_login.txt 이름으로 새 파일을 생성하고

login-1이라고 작성해 보자.

```
$ echo login-1 > test_login.txt
```

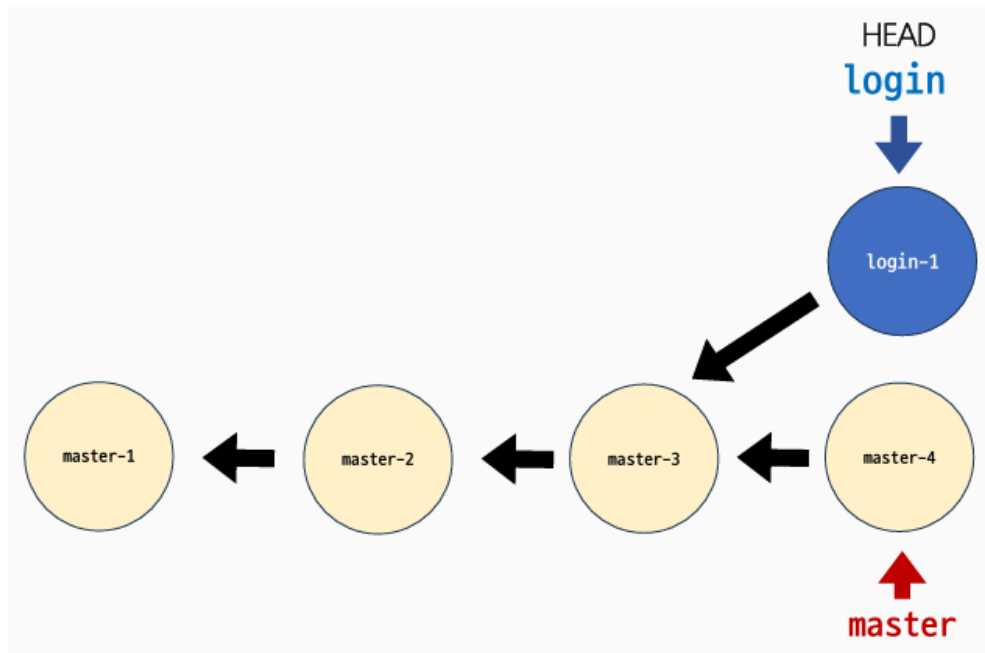
```
$ git add .
```

```
$ git commit -m "login-1"
```

이번에는 모든 브랜치의 커밋을 그래프로 확인해 보자.

```
$ git log --oneline --graph --all
```

```
SSAFY@DESKTOP-MDJG1NJ MINGW64 ~/Desktop/test (login)
$ git log --oneline --graph --all
* 7ca1d34 (HEAD -> login) login-1
| * 0913847 (master) master-4
|/
* 91ca616 master-3
* 8b69c94 master-2
* f9a0c7b master-1
```

이번에는 두 브랜치를 합치는 Merge 병합을 해보자.

login 브랜치의 내용을 master 브랜치로 병합 해 볼 것이다.

병합을 하면 login 브랜치에서 생성한 test_login.txt 파일을
Master 브랜치에서도 확인이 가능 할 것이다.

```
$ git switch master
```

```
$ git merge login
```

우리는 login 브랜치를 master 브랜치로 병합 할 것이기 때문에
HEAD, 가르키는 대상을 master 브랜치로 먼저 이동 후,
병합을 진행을 했다.

중요한 내용이다.

다시말해, 항상 Merge하기 전에 병합에 인수자와 피인수자가 있다면
인수자 브랜치로 이동 후 피 인수지를 Merge를 해야 한다.

[예시]

예를들어 branch1 그리고 branch2가 있다고 가정하자.

1. 현재 head 포인터가 branch1에 있다고 가정하자. 그리고
2. branch1 내용을 branch2 에 합칠 것이라고 가정하자. 그렇다면

1. git switch <branch2>
2. git merge <branch1> 이 된다.

즉, branch2로 이동 후 branch1을 가져와서 합치면 된다.

방금 login 브랜치를 master 브랜치로 병합을 했다.

vscode에 탐색기(explorer)를 확인해 보면 test_login.txt 파일이 추가 되어 있는 것을 확인할 수 있습니다.

병합이 잘 되었는지 커밋 이력을 확인해 보겠습니다.

```
$ git log --oneline --all --graph
```

```
SSAFY@DESKTOP-MDJG1NJ MINGW64 ~/Desktop/test (master)
$ git log --oneline --all --graph
*   5d1a31a (HEAD -> master) Merge branch 'login'
| \
| * 7ca1d34 (login) login-1
| * | 0913847 master-4
| /
* 91ca616 master-3
* 8b69c94 master-2
* f9a0c7b master-1
```

병합 후에는 5d1a31a 라는 해시코드와 함께 병합 후 새로운 커밋 이력이 생성 되었습니다. (해시코드는 각자 다르다. !!)

[주의사항]

병합을 위해서 login 브랜치에서 master 브랜치로 **switch**를 했다.

그런데 항상 git **switch** 하기 전에 확인해야 할 사항이 있다.

switch를 하기 전에 Working Directory 에 작업하던 파일이 있다면

반드시 버전관리가 되고 있는 상태 인지 확인해야 한다.

만약에 버전관리가 되고 있지 않는 파일이 있는 상태에서

switch를 진행 하면 버전관리가 되고 있지 않는 파일은 모든 파일이

switch가 된 브랜치에서도 파일이 똑같이 남아있기 때문이다.

따라서 **switch** 하기 전에

버전관리가 안되고 있는 파일이 있는지 반드시 확인하자.

이제 login 브랜치는 필요가 없으므로 삭제를 하자.

```
$ git branch -d login
```

삭제 후 확인까지 해보겠습니다.

```
$ git branch
```

```
SSAFY@DESKTOP-MDJG1NJ MINGW64 ~/Desktop/test (master)
$ git branch
* master
```

login 브랜치는 사라지고 master 브랜치만 존재한다.

그런데 한번 생각 볼 문제가 있다.

방금 실습은 master에서 login 브랜치 분기 후에 master-4 라고 내용을 추가했고 login 브랜치에서는 test_login.txt 라는 파일만 생성을 했다.

즉, 두 브랜치를 병합하기 전에 각각의 브랜치에서 수정했던 부분이 달라서 아무 문제 없이 병합이 잘 되었다.

그런데 만약에 login 브랜치에서 수정한 파일과 master 브랜치에서 수정한 파일이 같은 파일이라면? 어떻게 될까?

만약에 같은 파일을 login 브랜치 그리고 Master 브랜치에서 파일 내용 수정을 진행했더라도 수정한 부분이 다르다면 병합하는데 아무 문제가 발생하지 않지만, 수정한 부분이 겹치거나 서로의 파일에 영향을 준다면 merge conflict 라고 해서 충돌이 일어나게 된다. (곧 실습 해 볼 것이다)

만약에 수정한 부분이 겹쳤다면 git merge 후에 충돌 내용을 확인하고 수동으로 수정 후 다시 commit을 해야 한다.

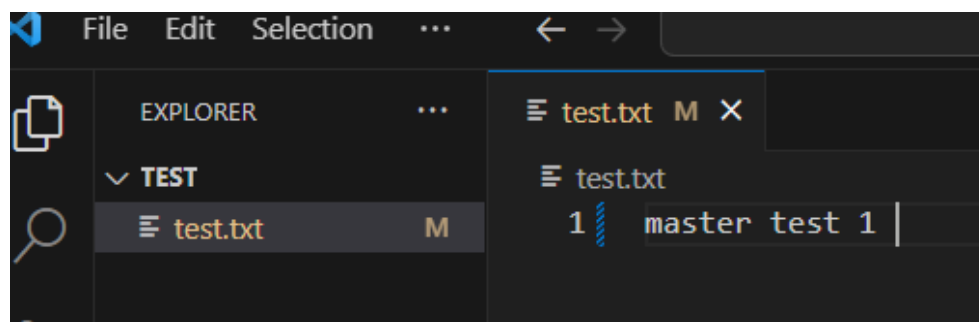
따라서 협업시 branch를 나눌 때에는, 병합 후 충돌이 나지 않도록 역할과 기능을 잘 분배하여 branch를 나눠야 하며 최대한 수정 부분이 겹치지 않도록 해야 할 것이다.

실습해보자.

다양한 merge 상황을 실습해 볼 것이다.

첫번째로 Fast-forward를 살펴보자.

사전설정으로



- test_login.txt 파일을 삭제하고

- test.txt 에 master test 1 을 입력 후 저장 하자.

```
$ git status  
$ git add .  
$ git commit -m "master test 1"
```

1. login branch 다시 생성하고 및 HEAD까지 이동 할 것이다.

git switch 이후에 -c 옵션까지 붙이면

login 브랜치를 생성과 동시에 head가 login으로 이동한다.

```
$ git switch -c login
```

2. login 브랜치에서 login.txt 파일을 생성 한다.

```
$ touch login.txt  
$ git add .  
$ git commit -m "login test 1"
```

3. master에 login을 병합 후 결과를 확인해 보자.

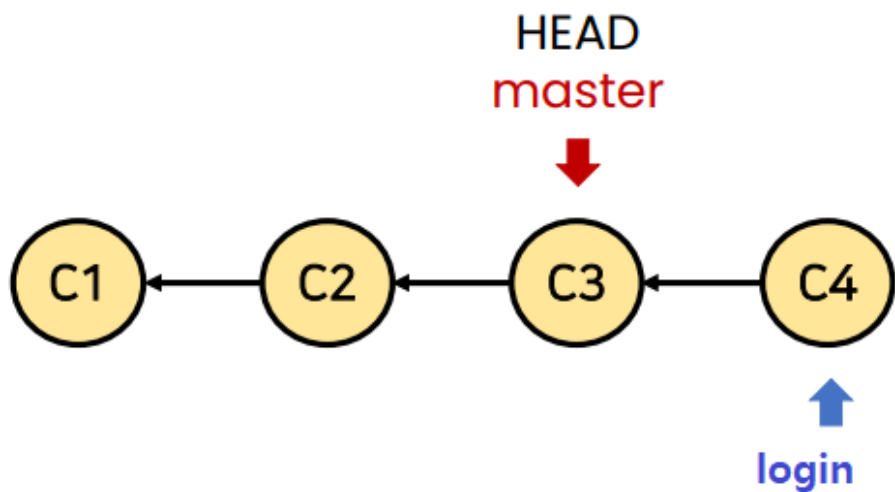
```
$ git switch master  
  
$ git merge login  
  
$ git log --oneline --all --graph
```

```
SSAFY@DESKTOP-MDJG1NJ MINGW64 ~/Desktop/test (master)
$ git log --oneline --all --graph
* 444ceb6 (HEAD -> master, login) login test 1
* 39a0e9f master test 1
* 5d1a31a Merge branch login
| \
| * 7ca1d34 login-1
* | 0913847 master-4
|/
* 91ca616 master-3
* 8b69c94 master-2
* f9a0c7b master-1
```

위 사진에서 빨간 박스 부분이 방금 실습한 내용 이다.

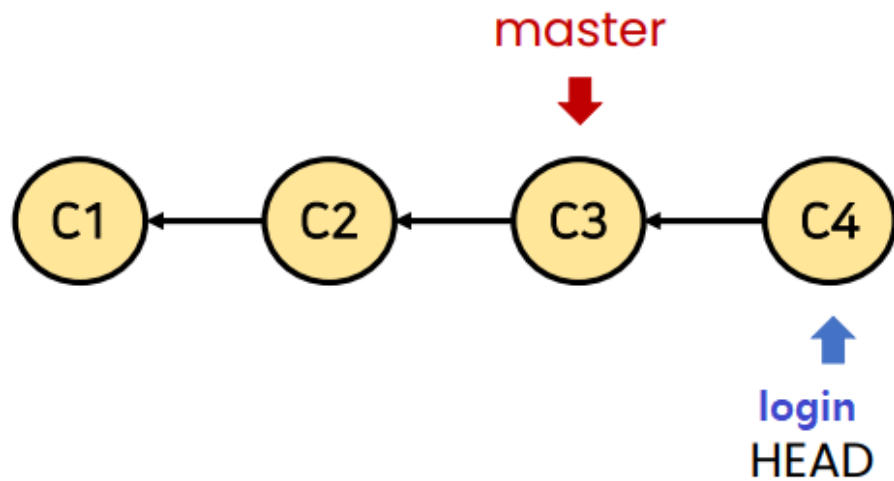
아래 그림으로 도 살펴 보겠습니다.

master에서 login 브랜치를 생성한 이후에 master 브랜치에서는 변경사항이 없었다.

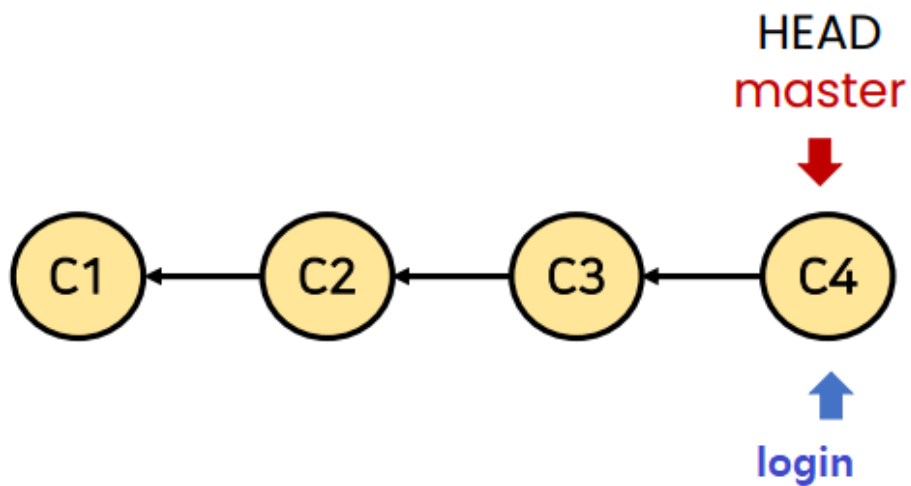


생성과 동시에 login 브랜치로 head가 옮겨 갔다.

그리고 login 브랜치에서만 변경사항이 있다. (login test 1 파일생성)



그리고 바로 master로 switch 를 한 이후에
login 브랜치를 master 브랜치로 merge 한 상황 이다.



이럴 경우는 결과를 그림으로 표현하면 위의 그림과 같다.

그저 Master 브랜치가 login 브랜치와 동일한 커밋을 가르키도록 이동만 한다. 이를 앞으로 빨리감기와 같다는 의미로 Fast-forward 라고 한다.

표현예시>

"머지할 때 fast-forward 되게 해줘."

"fast-forward 머지로 부탁해."

"충돌 없으면 fast-forward로 될 거야."

"브랜치 최신화해줘."

실습이 끝났으니까 login 브랜치 삭제 하도록 하자.

```
$ git branch -d login
```

자. 다음 상황이다.

아까 말했던 부분이다.

합병 중 수정한 부분이 같아서 충돌이 일어나는 경우를 살펴 보자.

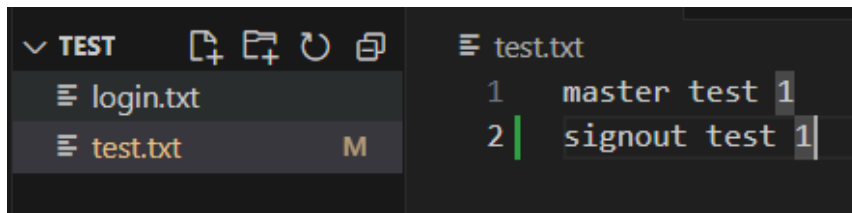


이번에는 signout 이라는 브랜치를 생성 후 이동해 보자.

```
$ git switch -c signout
```

현재 head는 signout을 가리키고 있을 것이다.

그리고



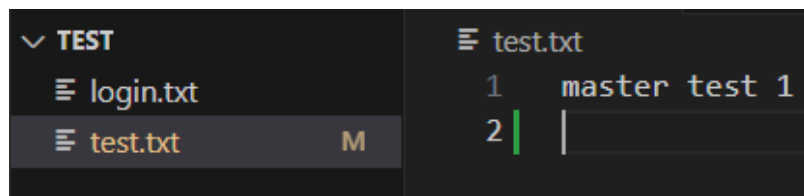
test.txt 파일 안에 signout test 1 라는 글을 추가 후 저장한다.

```
$ git add .  
$ git commit -m "signout test 1"
```

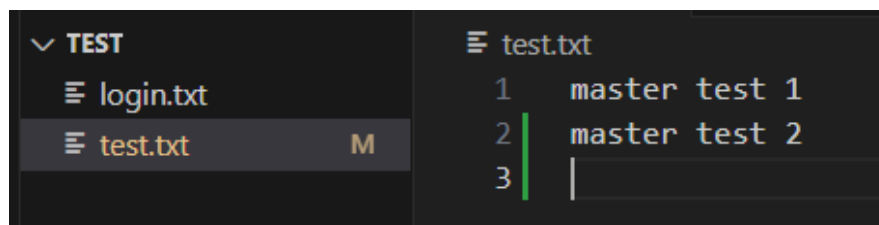
그리고 master 브랜치로 이동한다.

```
$ git switch master
```

그러면 test.txt 파일을 확인해 보면 아래 사진과 같이
방금 작성한 signout test 1 글이 보이지 않아야 정상이다.



이제 master test 2 내용을 추가 하도록 하자.



그리고 커밋을 진행 한다.

```
$ git add .
```

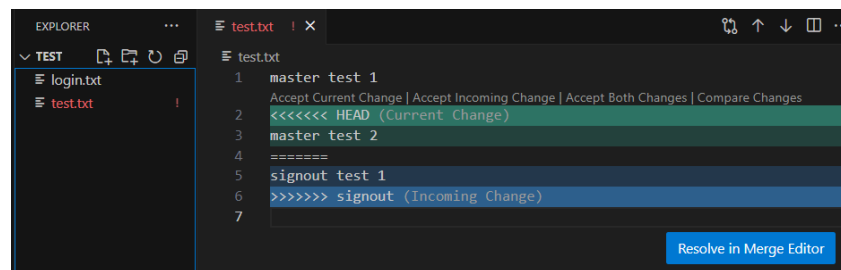
```
$ git commit -m 'master test 2'
```

지금 상황은 두 브랜치에서 같은 파일의 같은 라인을 수정했다.

merge를 진행 해 보자.

```
$ git merge signout
```

그랬더니 다음과 같은 충돌 메시지가 뜬다.



충돌 옵션을 살펴 보도록 하자.

accept current change 그리고 accept incoming change 등이 있다.

master test 1 아래에 수정 사항을 선택할 수가 있다.

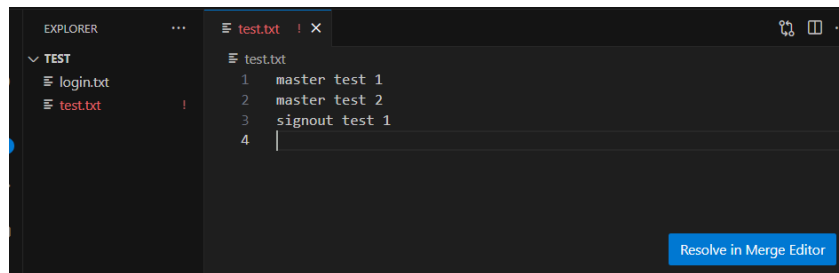
- 만약에 accept current change를 클릭하면

master test 1 아래에는 master test 2를 붙일 것이며

- 만약에 accept incoming change 를 클릭 한다면

master test 1 아래에는 signout test 1이 붙을 것이다.

accept both changes를 클릭시 두 변화가 모두 적용이 되고 수동으로 변경사항을 수정 할 수 있다.



수정이 완료 되었다면 resolve in merge Editor (파란박스) 클릭 후
complete merge를 클릭 하시면 완료 된다. 클릭 클릭 클릭
그리고 마지막으로 commit을 하면 병합 완료다.
만약에 vim 에디터 실행이 되면 ESC 누른 후 :wq 엔터

```
$ git commit -m '병합완료'
```

```
$ git log --oneline --all --graph
```

```
* 06db61d (HEAD -> master) 병합완료
| \
| * 2d28606 (signout) signout test 1
* | 988697f master test 2
|/
* 444ceb6 login test 1
* 39a0e9f master test 1
* 5d1a31a Merge branch 'login'
| \
| * 7ca1d34 login-1
* | 0913847 master-4
|/
* 91ca616 master-3
* 8b69c94 master-2
* f9a0c7b master-1
```

자 이렇게 branch와 병합을 실습 해 보았다.
필요없는 signout 브랜치를 삭제 하도록 하자.

```
$ git branch -d signout
```

이제 다른 팀원과 git hub와 같은 원격 저장소를 이용하여 협업을 할 준비가 완료되었다.

<끝>