Git03_BranchMerge

Git 온라인 리소스

Git - git-branch Documentation

Git - git-merge Documentation

https://git-scm.com/book/ko/v2 Ebook

Branch : 브랜치는 저장소의 새로운 분할

명령어	설명	비고
git branch <branch-name>	새로운 브랜치 생성.	현재 브랜치에서 분기된 새로운 브랜치를 생성하지만, 현재 브랜치는 그대로 유지
git checkout <branch-name>	지정된 브랜치로 이동.	생성된 브랜치 또는 존재하는 브랜치로 이동
git checkout -b <branch-name>	새로운 브랜치를 생성하고 해당 브랜치로 이동	git branch와 git checkout을 한 번에 실행하는 명령어.

분기 (Branching): master 브랜치에서 새로운 브랜치를 만들어 독립적인 작업을 수행

병합 (Merging): 작업이 완료된 브랜치를 master 브랜치에 다시 합쳐 변경 사항을 통합

동시 작업: 여러 개발자가 동시에 다른 작업을 수행

코드 안정성: master 브랜치를 안정적으로 유지하면서 새로운 기능을 개발하거나 버그를 수정

브랜치란?

Git 에서 브랜치(branch)는 프로젝트의 이력 흐름을 분기하여 기록하고 관리하는 기능입니다. 마치 나무의 가지처럼, 하나의 프로젝트에서 여러 개의 독립적인 작업 라인을 만들 수 있습니다.

브랜치의 핵심 개념

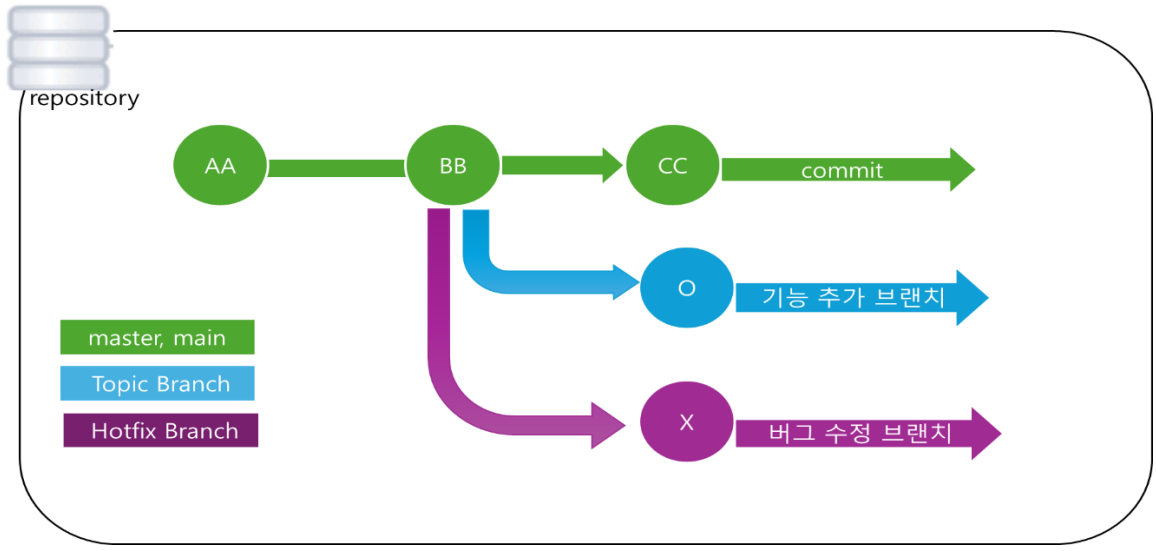
- **이력 분기**
  - 브랜치는 프로젝트의 변경 이력을 분리하여 기록합니다.
  - 이를 통해 특정 시점의 코드 상태를 유지하고, 필요에 따라 과거 버전으로 되돌아갈 수 있습니다.
- **독립적인 작업 공간**
  - 각 브랜치는 다른 브랜치의 영향을 받지 않는 독립적인 작업 공간을 제공합니다.
  - 따라서 여러 개발자가 동시에 다른 작업을 진행하거나, 새로운 기능을 실험적으로 개발할 수 있습니다.
- **동시 작업 및 협업**
  - 브랜치를 통해 여러 개발자가 동시에 다양한 작업을 수행하고, 나중에 변경 사항을 병합하여 하나의 프로젝트로 통합할 수 있습니다.
  - 이는 효율적인 협업을 가능하게 합니다.
- **코드 안정성 유지**
  - 안정적인 버전의 코드를 담고 있는 주요 브랜치(예: master, main)를 유지하면서, 새로운 기능 개발이나 버그 수정 작업을 별도의 브랜치에서 진행할 수 있습니다.
  - 이를 통해 주요 브랜치의 안정성을 확보하고, 오류 발생 시 영향을 최소화할 수 있습니다.

브랜치의 주요 용도:

- **새로운 기능 개발**: 새로운 기능을 추가할 때 별도의 브랜치를 생성하여 개발하고, 완료 후 주요 브랜치에 병합합니다.
- **버그 수정**: 버그를 수정할 때 별도의 브랜치를 생성하여 수정 작업을 진행하고, 수정 사항을 주요 브랜치에 반영합니다.
- **실험적인 개발**: 새로운 아이디어나 기술을 실험적으로 적용해볼 때 별도의 브랜치를 생성하여 테스트합니다.
- **릴리스 관리**: 특정 버전의 코드를 릴리스할 때 해당 버전의 코드를 별도의 브랜치로 분리하여 관리합니다.

브랜치를 효과적으로 사용하면 코드 관리의 효율성을 높이고, 협업을 원활하게 하며, 코드의 안정성을 확보할 수 있습니다.

아래그림은 분기된 브랜치를 보여주며, 다른 브랜치의 영향을 받지 않으므로 동일한 리포지토리에서 여러 변경을 동시에 진행할 수 있음을 나타냅니다.



리포지토리 (repository): 코드, 파일, 변경 이력 등을 저장하는 공간을 나타냄

AA, BB, CC (녹색 화살표): master 또는 main 브랜치로 안정적인 버전의 코드를 포함하며, 프로젝트의 주요 개발 라인임, 각각 커밋(commit)을 나타내며, 코드 변경 사항의 기록임, master 브랜치를 통합 브랜치로 사용

O (파란색 화살표): 기능 추가 브랜치로 master 브랜치에서 분기되어 새로운 기능을 개발하는 데 사용됨

X **(보라색 화살표):** 버그 수정 브랜치로 master 브랜치에서 분기되어 버그를 수정하는 데 사용됨

브랜치 운영 방법

1) 통합 브랜치 (Integration Branch)

통합 브랜치는 언제든지 릴리스 버전을 생성할 수 있도록 안정적인 상태를 유지하는 브랜치입니다. 또한 토픽 브랜치의 분기 기준이 되므로 안정성이 중요합니다. 코드 변경은 주로 토픽 브랜치를 통해 이루어지며, 통합 브랜치는 Jenkins 와 같은 CI 도구를 사용하여 자동 빌드 및 테스트를 수행합니다.
일반적으로 master 또는 main 브랜치를 통합 브랜치로 사용합니다.

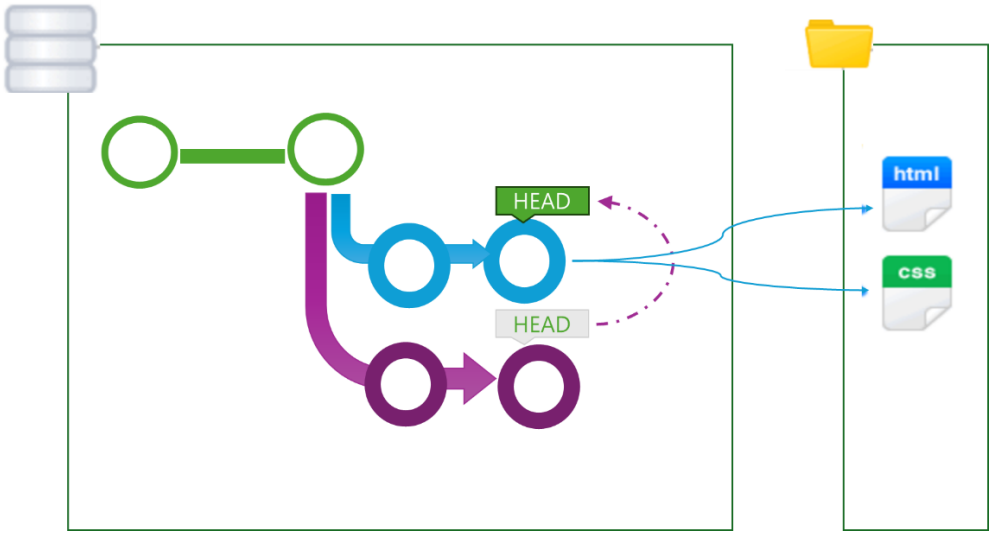
2) 토픽 브랜치 (Topic Branch)

토픽 브랜치는 기능 추가, 버그 수정 등 특정 작업을 수행하기 위해 생성하는 브랜치입니다. 여러 작업을 동시에 진행할 경우, 각 작업마다 토픽 브랜치를 생성합니다. 토픽 브랜치는 통합 브랜치에서 분기하여 생성하고, 작업 완료 후 통합 브랜치에 병합하는 방식으로 사용합니다.

브랜치 전환 및 관리 _ 체크아웃, HEAD, Stash

1. 브랜치 전환 (Checkout)

- **정의**: 작업할 브랜치를 변경하는 과정입니다.
- **작업**:
  - 대상 브랜치의 최신 커밋 내용을 작업 트리에 반영합니다.
  - 이후 커밋은 해당 브랜치에 추가됩니다.

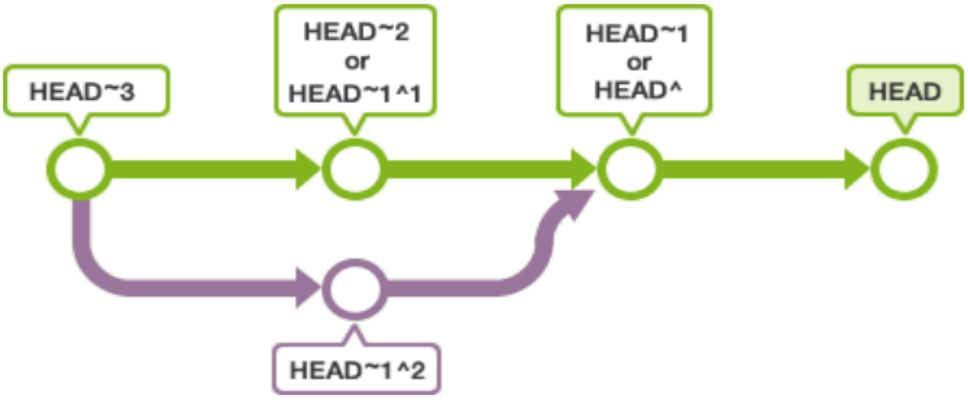


위 그림은 **Git 에서 브랜치를 전환할 때(Checkout)** HEAD 가 가리키는 브랜치에 따라 작업 디렉토리의 파일(html, css 등)이 변경되는 과정을 나타내며, 각 브랜치의 최신 커밋이 반영되어 브랜치 전환 시 파일 시스템이 변경되는 내용을 보여줍니다.

2. HEAD

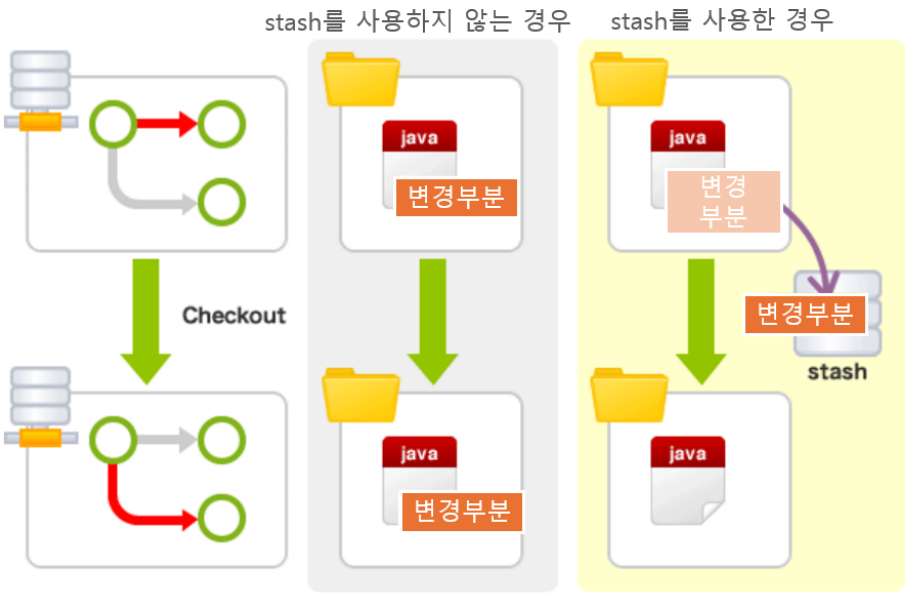
- **정의**: 현재 작업 중인 브랜치를 가리키는 포인터입니다.
- **역할**: HEAD 가 이동하면 작업 브랜치가 변경됩니다.
- **커밋 상대 위치 지정**:

- ~ (틸드): 이전 커밋을 지정합니다. (예: HEAD~2 는 2 세대 전 커밋)
- ^ (캐럿): 병합된 커밋의 부모 커밋을 지정합니다. (예: HEAD^1 은 첫 번째 부모 커밋)



3. Stash

- **정의**: 아직 커밋하지 않은 변경 내용을 임시 저장하는 공간입니다.
- **필요성**:
  - 작업 중인 변경 사항을 커밋하지 않고 다른 브랜치로 이동해야 할 때 사용합니다.
  - 체크아웃 시 충돌이 발생할 경우, 변경 사항을 임시 저장하고 충돌을 해결합니다.
- **사용법**:
  - git stash: 변경 사항을 스태시에 저장합니다.
  - git stash list: 스태시 목록을 확인합니다.
  - git stash apply: 스태시의 변경 사항을 적용합니다.
  - git stash drop: 스태시를 삭제합니다



위 그림에서 왼쪽(stash 미사용)에서는 파일 변경 후 체크아웃 시 변경 사항이 그대로 유지되지만, 오른쪽(stash 사용)에서는 변경 사항을 임시 저장한 후 체크아웃하여 파일을 원래 상태로 되돌리고, 변경 사항은 stash 에 보관되는 것을 확인할 수 있습니다

브랜치 통합

작업이 완료된 주제 브랜치는 결국 통합 브랜치에 통합됩니다. 브랜치 통합에는 merge 를 사용하는 방법과 rebase 를 사용하는 방법의 두 가지 유형이 있습니다. 어느 쪽을 사용할지에 따라 통합 후의 브랜치의 이력이 크게 다릅니다.

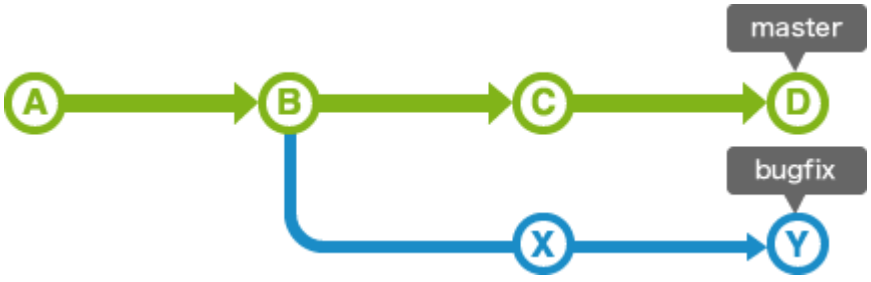
merge

merge 를 사용하면 여러 히스토리의 흐름을 결합할 수 있습니다.

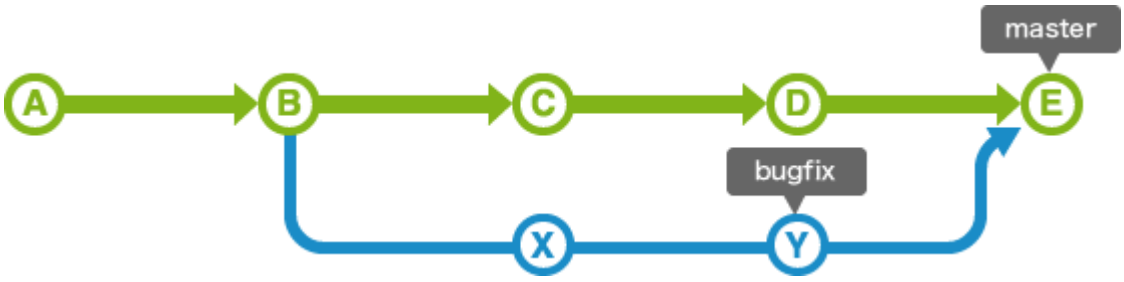
예를 들어 아래 그림과 같이 master 브랜치에서 분기하는 bugfix 라는 브랜치가 있다고 가정합니다.

이 bugfix 브랜치를 master 브랜치에 병합 할 때 master 브랜치의 상태가 이전부터 변경되지 않은 경우 매우 쉽게 병합 할 수 있습니다. bugfix 브랜치의 히스토리에는 master 브랜치의 히스토리가 모두 포함되어 있으므로 master 브랜치는 단순히 이동하기만 하면 bugfix 브랜치의 내용을 캡처할 수 있습니다. 또한 이러한 병합을 fast-forward 라고 합니다.

그러나 master 브랜치의 이력이 bugfix 브랜치를 분기했을 때보다 진행되어 버리는 경우도 있습니다. 이 경우 두 마스터 브랜치의 변경 사항과 bugfix 브랜치의 변경 사항을 하나로 결합해야합니다.

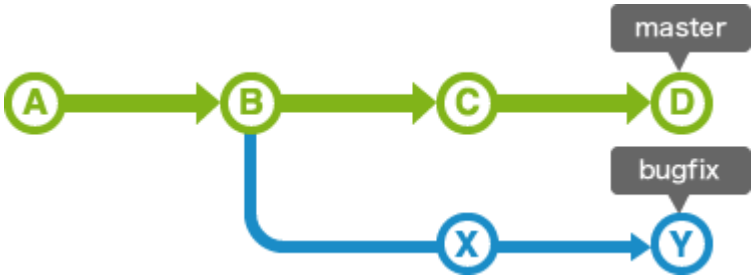


따라서 두 변경 사항을 모두 캡처한 병합 커밋이 만들어집니다. 마스터 브랜치의 시작은 커밋으로 이동합니다.



rebase

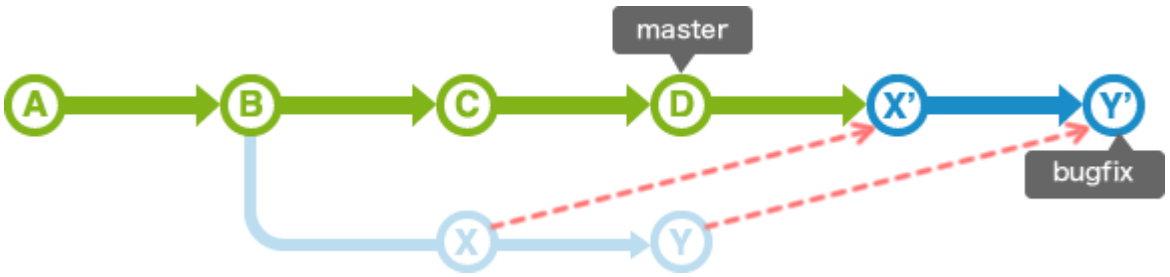
merge 예제와 마찬가지로 아래 그림과 같이 master 브랜치에서 분기하는 bugfix 라는 브랜치가 있다고 가정합니다.



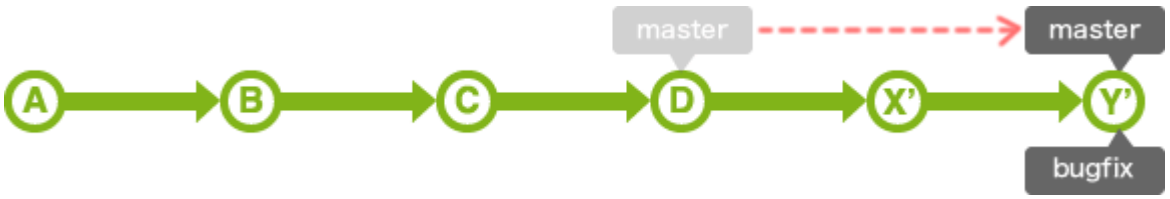
이것에 rebase 를 사용해 브랜치의 통합을 실시했을 경우에는 다음의 그림과 같은 이력이 됩니다. 그러면 어떤 단계로 병합할 것인지에 대해 간략하게 설명합니다.

먼저 bugfix 브랜치를 master 브랜치로 재구성하면 bugfix 브랜치의 히스토리가 master 브랜치 뒤로 바뀝니다. 따라서 그림과 같이 히스토리는 단일화됩니다.

이 때 이동하는 커밋 X 와 Y 에서는 충돌이 발생할 수 있습니다. 그 때는 각각의 커밋에서 발생한 경합 개소를 수정해 나갈 필요가 있습니다.



rebase 한 것만으로 master 의 선두의 위치는 그대로입니다. 따라서 master 브랜치에서 bugfix 브랜치를 병합하여 bugfix 시작 부분으로 이동합니다.



Git Merge와 Rebase: 이력 통합 방식 비교

Git 에서 merge 와 rebase 는 모두 브랜치의 변경 이력을 통합하는 데 사용되지만, 작동 방식과 결과가 다릅니다.

1. Merge

- **특징**:
  - 변경 이력을 그대로 유지합니다.

    - 병합 커밋(merge commit)을 생성하여 두 브랜치의 변경 사항을 통합합니다.
    - 이력이 복잡해질 수 있지만, 원래의 커밋 이력을 보존합니다.
- **사용 시나리오**:
    - 변경 이력을 중요하게 생각하는 경우
    - 협업 과정에서 발생한 모든 변경 사항을 기록하고 싶은 경우
    - 여러 사람이 동시에 작업하는 환경에서 충돌을 최소화하고 싶은 경우

2. Rebase

- **특징**:
    - 변경 이력을 단순하게 만듭니다.
    - 현재 브랜치의 기반을 다른 브랜치의 최신 커밋으로 변경하고, 변경 사항을 순차적으로 적용합니다.
    - 이력이 깔끔해지지만, 원래의 커밋 이력이 변경됩니다.
- **사용 시나리오**:
    - 깔끔하고 선형적인 이력을 유지하고 싶은 경우
    - 개인 브랜치에서 작업하는 경우
    - 코드 리뷰 시 이력을 단순화하여 검토를 용이하게 하고 싶은 경우

<<팀 운영 정책에 따른 선택>>

merge 와 rebase 는 팀의 운영 정책에 따라 구분하여 사용하는 것이 좋습니다.

- **이력 단일화 정책**:
    - 토픽 브랜치에 통합 브랜치의 최신 코드를 반영할 때 rebase 를 사용합니다.
    - 통합 브랜치에 토픽 브랜치를 병합할 때, 먼저 rebase 를 수행한 후 merge 를 사용합니다.
- **이력 보존 정책**:

- 모든 변경 이력을 보존해야 하는 경우 merge 를 사용합니다.
- 코드 리뷰 시 원래의 커밋 이력을 확인해야 하는 경우 merge 를 사용합니다.

<<예시 시나리오>>

- **개인 토픽 브랜치**: 개인적으로 작업하는 토픽 브랜치에서는 rebase 를 사용하여 이력을 깔끔하게 유지할 수 있습니다.
- **공유 토픽 브랜치**: 여러 사람이 협업하는 토픽 브랜치에서는 merge 를 사용하여 충돌을 최소화하고 변경 이력을 보존할 수 있습니다.
- **통합 브랜치 병합**: 토픽 브랜치를 통합 브랜치에 병합할 때, 먼저 rebase 를 수행하여 토픽 브랜치의 이력을 정리하고, merge 를 사용하여 통합 브랜치에 병합합니다.

<< 주의사항>>

- rebase 는 커밋 이력을 변경하므로, 이미 공유된 브랜치에서는 사용하지 않는 것이 좋습니다.
- rebase 를 사용할 때는 충돌 발생 가능성을 고려하고, 충돌 해결 방법을 숙지해야 합니다.

Git 원격 저장소와 협업의 핵심: Pull, Fetch, Push

1. 원격 저장소 (Remote Repository)

Git 을 사용하는 협업의 중심에는 원격 저장소가 있습니다. 이는 인터넷이나 네트워크를 통해 접근할 수 있는 공유 저장소로, 여러 개발자가 함께 작업하고 코드를 공유하는 데 필수적인 공간입니다.

마치 공동 작업실과 같이, 모든 팀원이 최신 코드를 확인하고 변경 사항을 반영할 수 있도록 해줍니다. 대표적인 예로는 GitHub, GitLab, Bitbucket 등이 있습니다.

2. 풀 (Pull): 원격 저장소의 변화를 내 작업 공간에 반영하기

원격 저장소의 최신 변경 사항을 내 로컬 저장소로 가져와 작업 공간을 최신 상태로 유지하는 것은 협업의 기본입니다.

git pull 명령어는 이 과정을 자동화합니다. 먼저 원격 저장소의 최신 커밋과 브랜치 정보를 가져오는 fetch 작업을 수행하고, 가져온 변경 사항을 현재 작업 중인 로컬 브랜치에 병합하는 merge 작업을 연속적으로 수행합니다. 이는 마치 공동 작업실의 최신 자료를 내 책상 위에 정리해두는 것과 같습니다.

git pull origin main 과 같이 사용하면 origin 이라는 원격 저장소의 main 브랜치 변경 사항을 현재 브랜치에 반영합니다.

3. 페치 (Fetch): 원격 저장소의 변화를 확인하고 준비하기

git fetch 명령어는 원격 저장소의 최신 정보를 가져오지만, 로컬 브랜치에 자동으로 병합하지는 않습니다. 이는 마치 공동 작업실의 새 자료를 확인만 하고, 정리 여부는 나중에 결정하는 것과 같습니다.

원격 저장소의 변경 사항을 미리 확인하고 병합 여부를 신중하게 결정하고 싶을 때 유용합니다. git fetch origin 과 같이 사용하면 origin 저장소의 최신 정보를 로컬 저장소에 가져옵니다.

4. 푸시 (Push): 내 작업 결과를 원격 저장소에 공유하기

로컬 저장소에서 작업한 내용을 원격 저장소에 공유하는 것은 협업의 완성입니다. git push 명령어는 로컬 브랜치의 커밋을 원격 브랜치로 전송합니다.

만약 원격 브랜치가 존재하지 않으면 자동으로 생성합니다. 이는 마치 내 작업 결과를 공동 작업실에 게시하여 다른 팀원들과 공유하는 것과 같습니다.

git push origin main 과 같이 사용하면 로컬 main 브랜치의 커밋을 origin 저장소의 main 브랜치로 전송합니다. 다만, 원격 브랜치에 다른 사람이 변경한 내용이 있는 경우 충돌이 발생할 수 있으므로 주의해야 합니다.

<< Git 원격 저장소와 명령어 요약>>

1. 원격 저장소 (Remote Repository):

- 협업을 위한 공유 저장소 (GitHub, GitLab 등)

2. 풀 (Pull):

- 원격 저장소 변경 사항을 로컬에 병합 (fetch + merge)
- git pull origin main: origin/main 브랜치 변경 사항 반영

3. 페치 (Fetch):

- 원격 저장소 변경 사항을 로컬에 저장 (병합 X)
- git fetch origin: origin 저장소 최신 정보 가져옴

4. 푸시 (Push):

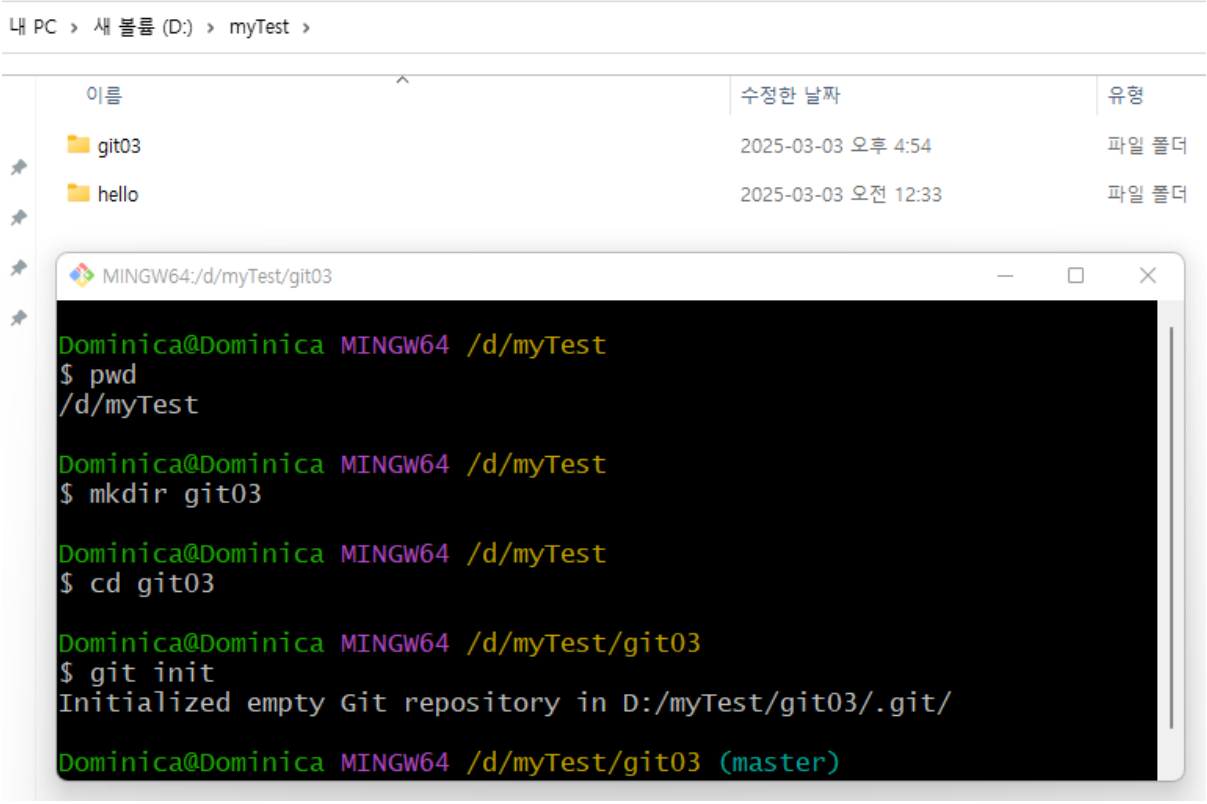
- 로컬 변경 사항을 원격 저장소에 업로드
- git push origin main: 로컬 main 브랜치 커밋을 origin/main 에 전송
- 충돌 주의

실습 : 브랜치를 사용

Q1. 저장소 -> 파일 -> 커밋을 하자

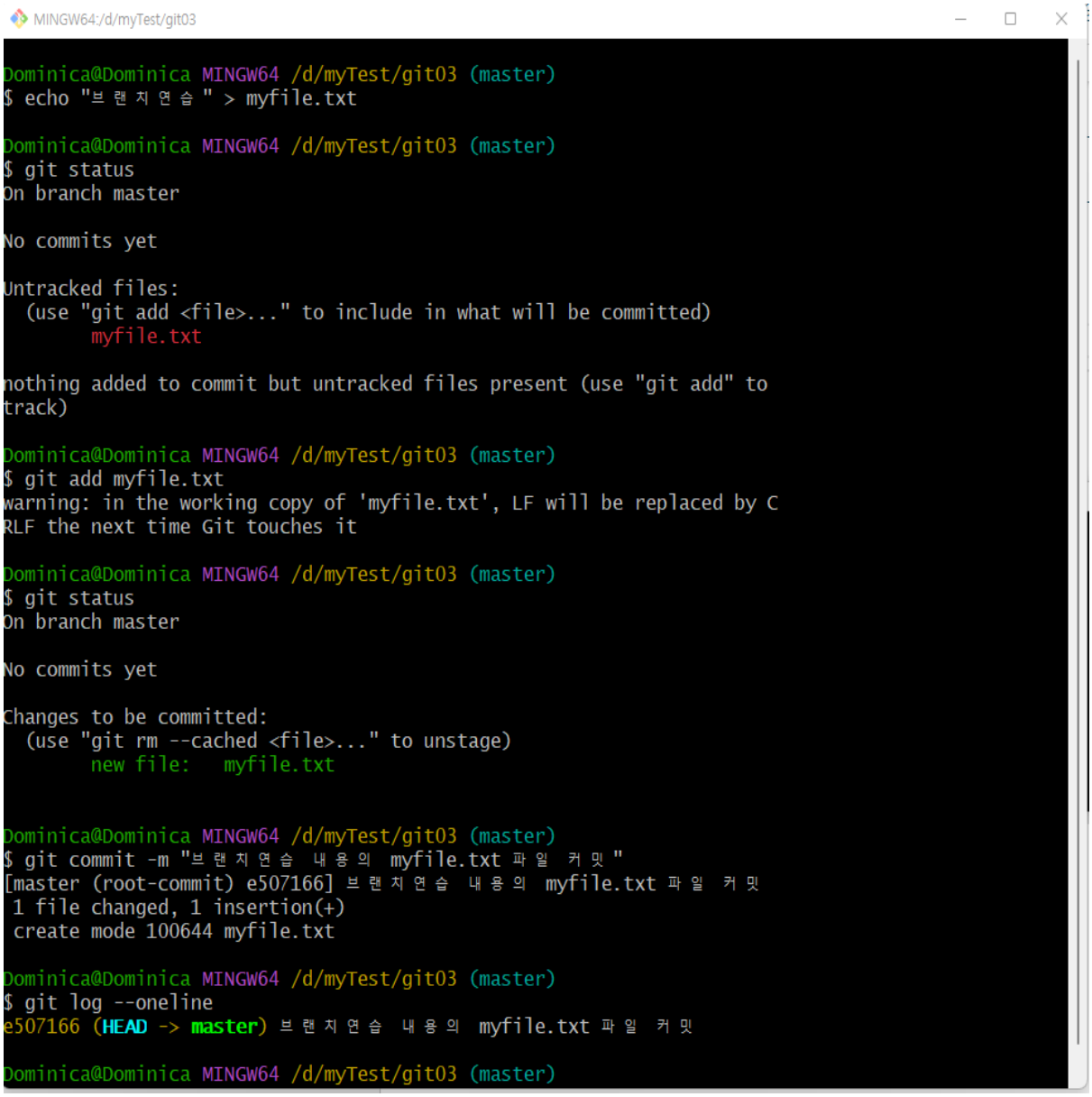
[HEAD가 master 브랜치의 최신 커밋을 가리키는 기본적인 Git 저장소 상태가 생성]

- **저장소 생성**



-myfile.txt 파일을 커밋

```
echo "브랜치연습" > myfile.txt
git status
git add myfile.txt
git status
git commit -m "브랜치연습 내용의 myfile.txt 파일 커밋"
git log --oneline
```

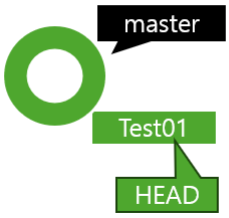

MINGW64:/d/myTest/git03
Dominica@Dominica MINGW64 /d/myTest/git03 (master)
$ echo "브랜치연습" > myfile.txt
Dominica@Dominica MINGW64 /d/myTest/git03 (master)
$ git status
On branch master
No commits yet
Untracked files:
(use "git add <file>..." to include in what will be committed)
myfile.txt
nothing added to commit but untracked files present (use "git add" to
track)
Dominica@Dominica MINGW64 /d/myTest/git03 (master)
$ git add myfile.txt
warning: in the working copy of 'myfile.txt', LF will be replaced by C
RLF the next time Git touches it
Dominica@Dominica MINGW64 /d/myTest/git03 (master)
$ git status
On branch master
No commits yet
Changes to be committed:
(use "git rm --cached <file>..." to unstage)
new file: myfile.txt
Dominica@Dominica MINGW64 /d/myTest/git03 (master)
$ git commit -m "브랜치연습 내용의 myfile.txt 파일 커밋"
[master (root-commit) e507166] 브랜치연습 내용의 myfile.txt 파일 커밋
1 file changed, 1 insertion(+)
create mode 100644 myfile.txt
Dominica@Dominica MINGW64 /d/myTest/git03 (master)
$ git log --oneline
e507166 (HEAD -> master) 브랜치연습 내용의 myfile.txt 파일 커밋
Dominica@Dominica MINGW64 /d/myTest/git03 (master)

Q2. Test01 이라는 브랜치를 생성하자

```
git branch Test01
```

```
Dominica@Dominica MINGW64 /d/myTest/git03 (master)
$ git branch Test01
```

Q3. 브랜치에 커밋을 추가해서 Test01브랜치를 체크아웃하자




```
$ git checkout Test01
```

```
Dominica@Dominica MINGW64 /d/myTest/git03 (master)
$ git checkout Test01
Switched to branch 'Test01'

Dominica@Dominica MINGW64 /d/myTest/git03 (Test01)
$
```

Q3. Test01 브랜치를 체크아웃한 상태에서 커밋을 하면 Test01 브랜치에 히스토리가 기록된다. myfile.txt에 add 명령 설명을 추가한 다음 커밋해 보자

```
$ echo "add 설명 추가" >> myfile.txt
$ git status
$ git add myfile.txt
$ git status git commit -m "myfile.txt 에 add 설명 추가"
$ git log --oneline
```

```
Dominica@Dominica MINGW64 /d/myTest/git03 (Test01)
$ git log --oneline
393da9e (HEAD -> Test01) myfile.txt에 add 설명 추가
e507166 (master) 브랜치연습 내용의 myfile.txt 파일 커밋
```

Test01 브랜치에는 "myfile.txt에 add 설명 추가" 커밋이 있고, master 브랜치에는 "브랜치연습 내용의 myfile.txt 파일 커밋" 커밋이 있다

HEAD는 현재 Test01 브랜치를 가리키고 있으므로, Test01 브랜치에서 작업 중임을 알 수 있다.

Q4. Test01브랜치에 변경한 내용을 master 브랜치에 통합해보자

현재 작업관리자 폴더와 파일을 확인 후 진행 한다.

```
$ git checkout master
$ git merge Test01
$ git log --oneline --graph --all
$ cat myfile.txt
```

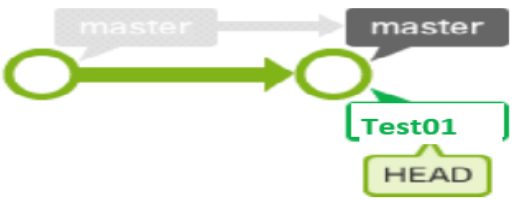
```
Dominica@Dominica MINGW64 /d/myTest/git03 (Test01)
$ git checkout master
Switched to branch 'master'

Dominica@Dominica MINGW64 /d/myTest/git03 (master)
$ git merge Test01
Updating e507166..393da9e
Fast-forward
 myfile.txt | 1 +
 1 file changed, 1 insertion(+)

Dominica@Dominica MINGW64 /d/myTest/git03 (master)
$ git log --oneline --graph --all
* 393da9e (HEAD -> master, Test01) myfile.txt에 add 설명 추가
* e507166 브랜치연습 내용의 myfile.txt 파일 커밋

Dominica@Dominica MINGW64 /d/myTest/git03 (master)
$ cat myfile.txt
브랜치연습
add 설명 추가
```

master 브랜치로 체크아웃

Test01 브랜치 병합

확인

파일내용확인



master 브랜치가 가리키는 커밋이 Test01과 같은 위치로 이동.
fast-forward 병합

Q5. 브랜치 삭제해보자.

```
$ git branch -d Test01
```

삭제 확인

```
$ git branch
 *master
```

```
Dominica@Dominica MINGW64 /d/myTest/git03 (master)
$ git branch -d Test01
Deleted branch Test01 (was 393da9e).

Dominica@Dominica MINGW64 /d/myTest/git03 (master)
$ git branch
* master
```

Q6. Git 브랜치 병렬 작업 연습 (Test02, Test03)을 하자

① Test02 브랜치와 Test03 브랜치를 생성하고, Test02 브랜치로 체크아웃.

```
Dominica@Dominica MINGW64 /d/myTest/git03 (master)
$ git branch Test02

Dominica@Dominica MINGW64 /d/myTest/git03 (master)
$ git branch Test03

Dominica@Dominica MINGW64 /d/myTest/git03 (master)
$ git checkout Test02
Switched to branch 'Test02'

Dominica@Dominica MINGW64 /d/myTest/git03 (Test02)
$ git branch
* Test02
  Test03
  master
```

② Test02 브랜치의 myfile.txt 파일에 commit 명령에 대한 설명을 추가하고 커밋

```
Dominica@Dominica MINGW64 /d/myTest/git03 (Test02)
$ echo "commit 인덱스의 상태를 기록합니다." >> myfile.txt

Dominica@Dominica MINGW64 /d/myTest/git03 (Test02)
$ git add myfile.txt
warning: in the working copy of 'myfile.txt', LF will be replaced by CRLF the next time Git touc
hes it

Dominica@Dominica MINGW64 /d/myTest/git03 (Test02)
$ git commit -m "commit 설명 추가"
[Test02 9053b37] commit 설명 추가
 1 file changed, 1 insertion(+)

Dominica@Dominica MINGW64 /d/myTest/git03 (Test02)
```

③ Test03 브랜치로 체크아웃

```
Dominica@Dominica MINGW64 /d/myTest/git03 (Test02)
$ git checkout Test03
Switched to branch 'Test03'
```

④ Test03 브랜치의 myfile.txt 파일에 pull 명령에 대한 설명을 추가하고 커밋

```
Dominica@Dominica MINGW64 /d/myTest/git03 (Test03)
$ echo "pull 원격 리포지토리의 내용을 가져옵니다." >> myfile.txt

Dominica@Dominica MINGW64 /d/myTest/git03 (Test03)
$ git add myfile.txt
warning: in the working copy of 'myfile.txt', LF will be replaced by CRLF the next time Git touc
hes it

Dominica@Dominica MINGW64 /d/myTest/git03 (Test03)
$ git commit -m "pull 설명 추가"
[Test03 21ea38e] pull 설명 추가
 1 file changed, 1 insertion(+)
```

Q7. Git 브랜치 병합 시 충돌이 발생할 수 있으며, 수동으로 충돌을 해결하고 병합 커밋을 생성해 보자

또한, Fast-forward 병합과 non-fast-forward 병합의 차이점을 생각해 보자.

① Test02 브랜치의 변경 사항과 Test03 브랜치의 변경 사항을 master 브랜치에 통합.

```
Dominica@Dominica MINGW64 /d/myTest/git03 (master)
$ git merge Test02
Updating 393da9e..9053b37
Fast-forward
 myfile.txt | 1 +
 1 file changed, 1 insertion(+)
```

② 먼저 master 브랜치를 체크아웃하고 Test02 브랜치를 병합.

```
Dominica@Dominica MINGW64 /d/myTest/git03 (master)
$ git merge Test03
Auto-merging myfile.txt
CONFLICT (content): Merge conflict in myfile.txt
Automatic merge failed; fix conflicts and then commit the result.
```

③ Test03 브랜치를 병합하고 충돌을 해결

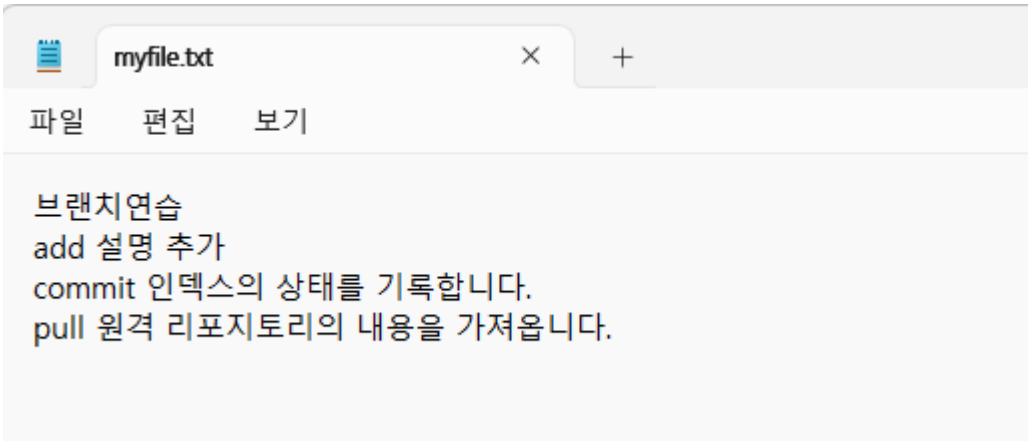
텍스트 편집기 또는 IDE 를 사용하여 myfile.txt 파일을 확인

```
브랜치연습
add 설명 추가
<<<<<<< HEAD
commit 인덱스의 상태를 기록합니다.
=======
pull 원격 리포지토리의 내용을 가져옵니다.
>>>>>>> Test03
```

<<<<<<< HEAD와 ======= 사이에는 현재 브랜치(master)의 내용이,

=======와 >>>>>>> Test03 사이에는 병합하려는 브랜치(Test03)의 내용이 표시

메모장에서 수동으로 수정 후 저장



④ 충돌 해결 후 커밋

```
Dominica@Dominica MINGW64 /d/myTest/git03 (master|MERGING)
$ git add myfile.txt

Dominica@Dominica MINGW64 /d/myTest/git03 (master|MERGING)
$ git commit -m "Test03 브랜치 병합 및 충돌 해결"
[master 45d813b] Test03 브랜치 병합 및 충돌 해결

Dominica@Dominica MINGW64 /d/myTest/git03 (master)
$
```

git log --oneline --graph --all 명령어를 사용하여 커밋 히스토리를 확인하면 병합 커밋이 생성되었음을 확인

```
Dominica@Dominica MINGW64 /d/myTest/git03 (master)
$ git log --oneline --graph --all
*   45d813b (HEAD -> master) Test03 브랜치 병합 및 충돌 해결
|\
| * 21ea38e (Test03) pull 설명 추가
* | 9053b37 (Test02) commit 설명 추가
|/
* 393da9e myfile.txt에 add 설명 추가
* e507166 브랜치연습 내용의 myfile.txt 파일 커밋
```