기초 Git&Github 활용법

Lee Jun Hyeok (wnsx0000@gmail.com)

October 2, 2025

목차

1. 기초 Git&Github 활용법

1.1. 서론

1.1.1. 오픈소스

> *오픈소스(Open Source)는 코드를 제약 없이 확인, 수정, 배포가 가능한 소프트웨어 또는 그런 소프트웨어의 개발 방식임. 오픈소스는 특히 자신의 개발 실력이나 지식을 public하게 공개하거나, 프로젝트 경험을 증명하기 위해 주로 사용됨.*

예를 들어, Linux, GCC, GDB, GNOME 등을 포함하는 GNU 프로젝트, Apache(웹서버), Android, TensorFlow, Pytorch 등과 같은 오픈소스 생태계들이 존재함. 이런 오픈소스 생태계에는 문서화, 번역, 기능 추가 및 개선, 프로젝트 공개 등의 방식으로 기여가 가능함.

1.1.2. wsl 활용

윈도우 환경에서는 wsl을 설치해 리눅스 및 git&github를 사용하는 것이 편리함. 우선 'Windows 기능 켜기/끄기' 옵션을 검색해 들어가서 'Linux용 Windows 하위 시스템'을 체크하고, Hyper-V, Virtual Machine Platform, Windows 하이퍼바이저 등이 존재하면 적절한 것을 체크해야 함.

이후 powershell에서 CLI로 wsl을 설치하거나, 또는 microsoft store에서 Ubuntu를 다운받으면 됨. 이후 wsl 사용이 가능하고, 파일 탐색기에도 wsl(linux) 디렉토리가 생김.

1.2. Git&Github 기초

1.2.1. Git&Github

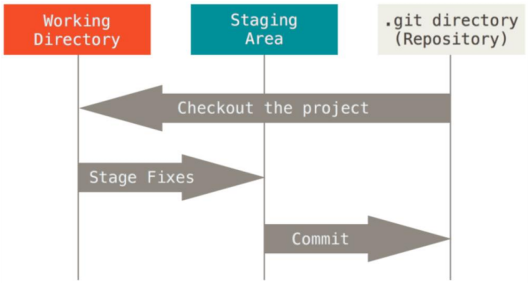
> *Git은 파일 및 디렉토리 변경 사항을 추적하는 버전 관리 시스템, Github는 git으로 관리하는 프로젝트를 저장할 수 있는 git 호스팅 서비스임. git&github를 사용하면 여러 사용자들끼리 협업하며 안전하게 버전 관리가 가능함.*

윈도우에서 git을 사용하는 경우 wsl을 설치해 linux 환경에서 git을 깔아 사용할 수도 있고, 윈도우용 git 어플리케이션을 설치해 사용할수도 있음.

참고로, git에서는 각 파일과 commit 등 데이터를 대상별로 유일한 checksum으로 구분해 관리함.

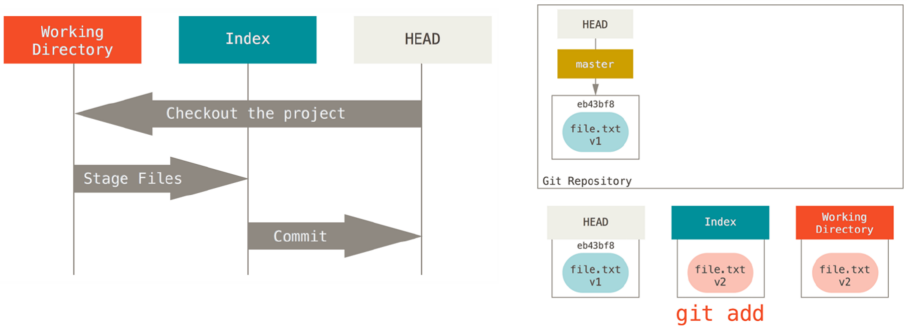
1.2.2. Git의 로컬 저장소와 파일 상태

> ***1. 로컬 저장소***
> *git의 로컬 저장소는 아래와 같이 3가지로 구분됨.*
>
> *1) Working Directory : git에 의한 버전 관리가 수행되는 파일들이 존재하는 디렉토리.*
> *2) Staging Area : stage에 올라온 파일들에 대한 정보가 저장되는 디렉토리.*
> *3) .git Repository : commit 정보가 저장되는 디렉토리.*



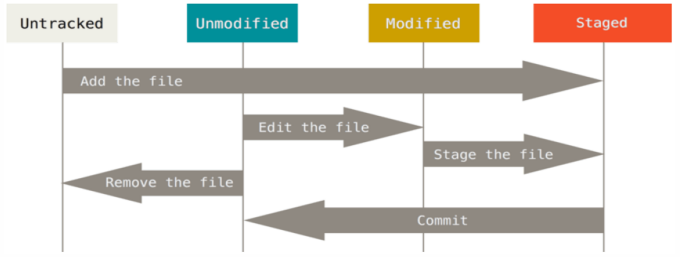
> ***2. 특정 시점의 로컬 저장소 구성***
> *특정 시점의 로컬 저장소는 실제로 3가지 파일 묶음으로 관리됨. 이런 식의 분류는 특히 reset에서 중요함.*

1) Working Directory : 작업 중인 실제 파일.
2) Index : 다음에 commit할 스냅샷.
3) HEAD : 현재 commit의 스냅샷. 즉, HEAD가 가리키는 commit의 스냅샷임.



3. 파일 상태
git에서는 파일의 상태를 아래와 같이 4가지로 구분해 관리함. 관리 대상인 상태는 Tracked이고, unmodified, modified, staged로 구분됨. git에서 파일은 umodified일 때 수정하면 modified가 되고, staging을 통해 staged가 되고, commit으로 다시 unmodified가 되는 lifecycle을 가짐.

1) Untracked : working directory에 존재하지만 파일이 버전 관리 대상이 아닌 상태. staging을 한 번도 하지 않은 파일은 untracked이고, staging을 한 번 하면 tracked가 됨.
2) Unmodified(Commited) : 파일이 로컬 저장소에 commit으로 안전하게 저장된 상태.
3) Modified : 파일이 수정되었지만 commit하지 않아 로컬 저장소에 저장되지 않은 상태.
4) Staged : 수정한 파일을 commit하기 위해 표시(staging)한 상태. 즉, stage에 올린 상태.



1.2.3. Git 특수 파일

git의 특수 파일들로는 아래와 같은 것들이 있음.

1. README.md
README.md에 작성한 내용은 github의 repository 페이지에 뜸.

2. .gitignore
.gitignore에 파일명 또는 디렉토링명을 지정하여 해당 파일 또는 디렉토리를 untracked로 지정할 수 있음.

*특히, [abc](해당 문자 중 하나.), [0-9](해당 범위의 수 중 하나.), a/**/z(중간에 임의의 경로 들어갈 수 있음.), * 등과 같은 표현을 사용할 수 있음.*

1.2.4. Git 기초 명령어들

git 기초 명령어들로는 아래와 같은 것들이 있음.

1. Config
git 사용 환경을 구성했으면 git config로 사용자의 name과 email을 지정해야 함.

```
git config --global user.name "<사용자명>"
git config --global user.email "<사용자 email>"
```

2. Git 저장소 지정

현재 디렉토리에서 git에 의한 버전 관리를 수행하도록 하려면 git init으로 git을 시작하면 됨.

```
git init
```

3. Staging

modified인 파일에 대해 commit을 생성하려면 git add로 stage에 올려야(staging) 함.

*이때 파일명에 * 등을 입력해 전체 파일을 staging하도록 하는 경우, .으로 시작하는 숨김 파일들은 stage에 올라가지 않음.*

```
git add <파일명>
```

4. Commit

git commit으로 staged 파일들에 대한 commit을 생성할 수 있음.

git commit만 입력한 경우 commit message를 입력하는 창이 나오는데, 대신 -m 옵션으로 commit message를 바로 지정할 수도 있음. -a 옵션을 넣으면 tracked 파일들에 대해서 add를 수행하고 commit 함(staging을 한 번이라도 했었어야 가능함.). –amed 옵션(amend는 '개정하다'라는 뜻임.)을 넣으면 해당 commit이 마지막 commit을 덮어씀(파일을 까먹고 포함시키지 않은 경우 등에 사용함.).

```
git commit
git commit -m "<commit message>"
git commit -a -m "<commit message>"
git commit --amend
```

5. 파일 제거/제외 및 파일명 변경

git rm을 사용하면 해당 파일을 실제로 삭제하고, 삭제되었다는 것을 git에 알림. 반면 git rm –cached를 사용하면 untracked로 지정해여 staging area에서 해당 파일을 제거함. 해당 명령어 사용 이후에 commit해야 로컬 저장소에 실제로 반영됨.

git restore –staged로 해당 파일의 staged 상태를 취소하고 tracked로 변경할 수 있음. git rm –cached는 아예 untracked로 지정하므로 차이가 있음.

git mv로 파일명을 변경하고, 이를 git에 알릴 수 있음. 만약 mv로 변경한 경우 기존 이름에 대해 git rm, 새로운 이름에 대해 git add를 해줘야 함.

```
git rm <파일명>
git rm --cached <파일명>

git restore --staged <파일명>

git mv <기존 이름> <새로운 이름>
```

6. 상태 출력

git status로 working directory의 상태를 출력할 수 있음.

git log로 commit 로그를 출력할 수 있음. 이때 -4와 같이 숫자를 지정해 출력할 commit 개수를, –oneline을 지정해 각 commit 정보를 한 줄로, –graph를 지정해 그래프로 출력할 수 있음.

git diff로는 working directory의 내용과 staging area 또는 commit의 내용을 비교하여 출력함. 이때 +, - 등으로 변경된 부분이 출력되는데, 변경된 부분은 @@ -1,5 +1,6 @@과 같이 출력됨(기존 파일에서 1~5번째 줄 삭제, 새로운 파일에서 1~6번째 줄 추가.).

```
git status

git log
git log -4 --oneline
git log --graph

git diff
```

7. git 명령어 도움말 출력
git help로 git 명령어 도움말을 출력할 수 있음.

```
git help
```

8. Tag
git tag로 특정 commit에 tag를 추가할 수 있음. tag를 붙이면 수많은 commit들 중 해당 commit을 해당 tag 이름으로 다룰 수 있어, 소프트웨어를 배포할 때 주로 사용한다고 함(tag 이름을 v1.5.3 등으로 지정할 수 있음.).

git tag로 tag 목록을 출력할 수 있고, -l 옵션을 줘서 해당 이름과 일치하는 tag(등 사용 가능)를 출력할 수 있음.*

tag는 tag 이름만을 포함하는 Lighweight Tag와, 그 외에도 이름, 이메일, 시간, tag 메시지를 포함하는 Annotated Tag로 구분됨. lightwieght tag는 단순히 tag 이름을 지정하여 생성할 수 있고, annotated tag는 -a 옵션과 tag 이름을 지정하여 생성할 수 있음. 또한 tag 메시지를 지정해야 하는데, git commit과 동일하게 -m으로 메시지를 바로 지정할 수 있음.

tag는 기본적으로 현재 commit에 생성하는데, 이전 commit 중 하나를 지정하여 생성하려면 특정 commit의 checksum 값을 tag 이름 뒤에 지정하면 됨. 이때 checksum 값은 겹치지 않는 선에서 일부만 지정해도 됨. 참고로 checksum 값은 git log –pretty=oneline으로 쉽게 확인할 수 있음.

git show로 해당 tag를 가지는 commit의 정보를 출력할 수 있음.

```
git tag
git tag -l "<tag 이름>"
git tag <tag 이름>
git tag -a <tag 이름> -m "<tag 메시지>"

git log --pretty=oneline
git tag -a <tag 이름> <checksum 값>

git show <tag 이름>
```

1.3. Git Branch&Remote

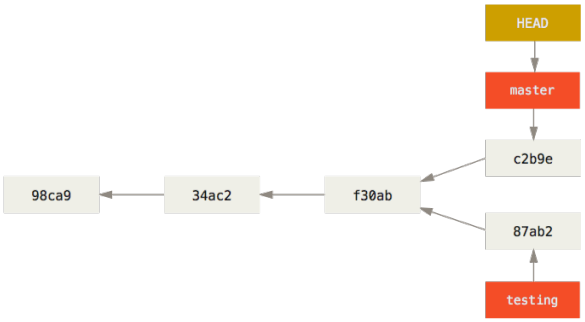
1.3.1. Git Branch

1. Git Branch
git에서 Branch는 독립된 작업 영역 또는 특정 commit을 가리키는 포인터로, 여러 개발자의 협업을 관리할 수 있도록 하는 기능임. 각 개발자들은 기능 구현이나 디버깅에 따라 개별적으로 branch를 파서 commit을 올리며 작업하다가, branch끼리 merge하여 코드를 합칠 수 있음.

git에서 자동 생성되는 default branch의 이름은 main임. 과거에는 master였어서, 일부 원격 저장소를 보면 branch명이 master일 수 있는데 이 경우 main으로 바꿔 작업하는 것이 좋음. default branch 이름은 github 설정에서 변경할 수도 있고, 로컬에서는 git config –global init.defaultBranch main 명령으로 변경할 수도 있음.

HEAD 포인터는 현재 작업중인 branch를 나타내는 포인터임. checkout 명령어를 사용하면 이 포인터가

가리키는 대상이 바뀜.

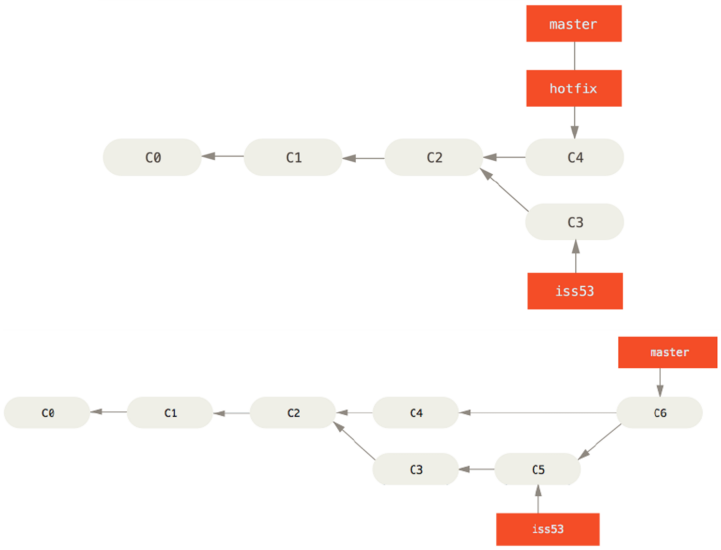


2. Git Merge

branch를 나누어 작업했으면, 이후 그 내용을 합치는 merge를 수행하게 됨. merge는 현재의 branch(HEAD)에 명령어에 지정한 branch의 내용에 추가되는 것으로, 이때 명령어에 지정한 branch에는 변화가 없음. 만약 merge의 결과가 지정한 branch와 달라진다면, 현재 branch에 merge 결과를 가진 새로운 commit이 생성됨.

git에서 merge는 3-Way Merge에 의해 수행됨. 3-way merge에서는 두 branch가 갈라지기 시작한 지점의 commit, 그리고 두 branch 각각의 최신 commit을 사용하여, 한쪽 branch에서 변경된 내용은 merge 결과에 반영하고, 변경되지 않은 내용은 merge 결과에 반영하지 않음. 이때 각 branch의 최신 commit에서 동일한 부분을 서로 다르게 변경한 경우, Conflict(충돌)가 발생했다고 함.

conflict가 발생했으면 새로운 commit이 생성되거나 merge가 완료되는 대신, ««««<, ========, »»»»>로 conflict가 발생한 부분이 파일에 표시됨. 해당 부분을 지우고 수정을 완료한 뒤, 다시 staging하고 commit을 생성하면 merge가 완료됨.



branch를 파서 작업하는 기간이 길어질수록 merge 시에 충돌 발생 가능성이 커지므로, 기능 및 이슈별로 branch를 파고, 코드 작성을 완료하면 즉시 main과 merge하는 것이 좋음.

1.3.2. Git Remote

1. 원격 저장소

git에서는 협업 및 백업을 위해 로컬 저장소의 정보를 원격(Remote) 저장소에 올려 저장하거나, 원격 저장소에 저장된 내용을 내려받아 로컬 저장소를 갱신함.

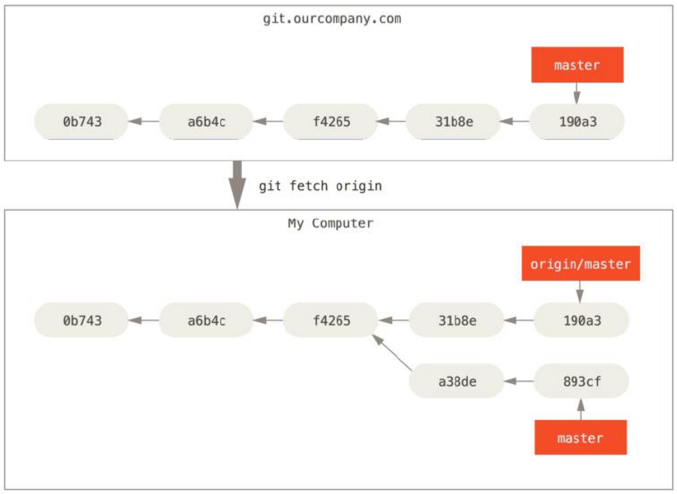
로컬 git에서 원격 저장소의 이름은 origin이 default임. 즉, clone을 하면 저장소명이 origin으로 지정됨.

하나의 프로젝트에서 여러 개의 원격 저장소를 활용할 수 있음.

2. 원격 Branch

원격(Remote) Branch는 원격 저장소의 branch이고, 원격 트래킹(Remote Tracking) Branch는 원격 branch를 추적하는 일종의 포인터임. 원격 트래킹 branch는 원격 저장소에 연결될 때마다 업데이트되며 자동으로 움직여서 branch처럼 보이지만, 임의로 움직일 수 없으며 실제로 사용하는 로컬 branch와는 다름.

원격 트래킹 branch의 이름은 origin/main과 같이 <원격 저장소 이름>/<원격 branch 이름> 형식임.



1.3.3. Git Branch&Remote 관련 명령어들

git branch 및 remote 관련 명령어들로는 아래와 같은 것들이 있음.

1. Branch

git branch 명령어로 branch를 생성 및 관리할 수 있음.

*git branch로 branch들을 출력할 수 있고(출력 결과에서 *이 붙은 것이 현재 branch임.), -a 옵션으로 원격 트래킹 branch들까지 출력할 수 있음. -v 옵션을 주면 각 branch가 가진 최신 commit의 checksum과 commit 메시지가 함께 간략히 출력됨. –merged 옵션을 주면 이미 merge된 branch들이 출력되고(삭제하는 게 좋을 수 있음.), –no-merged 옵션을 주면 merge되지 않은 branch들이 출력됨(merge되지 않은 것은 기본적으로 삭제되지 않음.).*

git branch에 branch 이름을 지정하면 해당 이름의 branch를 생성할 수 있고, -m 옵션을 줘서 branch 명을 변경할 수 있음. -d 옵션을 주면 해당 branch를 삭제할 수 있음. 이때 해당 branch가 아직 merge되지 않은 상태이면 삭제되지 않는데, -D 옵션을 주면 강제로 삭제할 수 있음.

```
git branch
git branch -a
git branch -v
git branch --merged
git branch --no-merged

git branch <branch 이름>
git branch -m <기존 이름> <새로운 이름>
git branch -d <branch 이름>
git branch -D <branch 이름>
```

default branch 이름은 github repository 설정에서 변경할 수도 있고, 로컬에서는 git config –global init.defaultBranch main 명령으로 변경할 수도 있음.

2. Checkout

branch 변경은 git checkout 명령어로 수행할 수 있음. -b 옵션을 주면 해당 이름의 branch를 생성한 이후, 바로 그 branch로 변경함. 또는 -b 사용 시에 추가로 기준 branch를 지정해 해당 branch로부터 새로운 branch를 생성할 수도 있음.

```
git checkout <branch 이름>
git checkout -b <branch 이름>
git checkout -b <branch 이름> <기준 branch 이름>
```

3. Merge

git merge 명령어를 사용해 현재 branch에 지정한 이름의 branch를 merge함. merge 이후에도 기존 branch는 여전히 존재하므로, 더 이상 사용하지 않는다면 git branch -d로 삭제하는 것이 좋음.

conflict가 발생한 경우 commit 생성 및 merge가 수행되는 대신 수정할 부분이 파일에 추가되는데, 해당 부분을 merge가 가능하도록 직접 수정한 뒤 staging 및 commit해야 함. 이때 conflict가 발생한 위치는 git status로도 확인할 수 있음.

```
git merge <branch 이름>
```

4. Clone

git clone 명령어로 원격 저장소의 내용을 로컬 저장소에 복사할 수 있음. 또한 -o 옵션으로 해당 원격 저장소의 이름을 origin 대신 지정할 수 있음.

```
git clone <원격 저장소 url>
git clone <원격 저장소 url> -o <원격 저장소 이름>
```

5. Remote

git remote로 원격 저장소와 로컬 저장소의 연결을 관리할 수 있음. git remote로 원격 저장소 목록(이름)을 출력하고, -v 옵션을 줘서 이름, url, 명령어를 포함하는 원격 저장소 목록을 출력할 수 있음. 또한 remote show 명령어로 특정 원격 저장소 정보를 자세히 출력할 수 있음.

remote add로 원격 저장소를 추가, remote remove로 원격 저장소를 삭제, remote rename으로 원격 저장소 이름을 변경할 수 있음.

이전에 정리한 것처럼 원격 저장소의 default 이름은 origin이므로, 연결 시에도 origin으로 지정하는 것이 편리할 수 있음. clone 시에는 origin으로 자동 연결됨.

특정 원격 트래킹 branch로 checkout할 수도 있지만 이는 논리적으로 적절하지 않으므로(문제가 생길 수 있다고 함.), -b 옵션을 사용해 로컬 branch를 생성해 작업하는 것이 좋음. 물론 단순 merge는 가능함.

```
git remote
git remote -v
git remote show <원격 저장소 이름>

git remote add <원격 저장소 이름> <URL>
git remote remove <원격 저장소 이름>
git remote rename <기존 이름> <새로운 이름>

git checkout pb/master
git checkout -b local-pb pb/master
```

6. Fetch

git fetch 명령어로 원격 저장소에서 commit 및 원격 branch와 관련된 최신 정보를 로컬 저장소에 불러올 수 있음. 이때 단순히 불러오기만 하고, 로컬 branch에 해당 내용을 반영하지는 않음.

```
git fetch <원격 저장소 이름>
```

7. Pull/Push

git pull과 git push 명령어로 원격 저장소에서 특정 원격 branch의 커밋을 가져오거나, 원격 저장소에 특정 로컬 branch의 commit을 올릴 수 있음. 이때 pull은 fetch와 merge를 합친 것으로 이해할 수 있음.

원격 저장소의 정보를 마지막으로 pull한 이후 변경 내역이 없어야 push가 가능함. 변경 사항이 존재한다면 pull을 먼저 해야 함.

push에 –delete 옵션으로 특정 원격 branch를 삭제할 수 있음.

참고로 push 시에 password를 입력하라고 하면 github의 developer setting에서 발급받은 token을 입력해야 함. 또는 매번 입력하는 대신 .git 디렉토리의 config 파일의 url 부분을 https://계정명:토큰@github.com/... 와 같이 지정할 수도 있음.

```
git pull
git pull <원격 저장소 이름> <원격 branch 이름>

git push
git push <원격 저장소 이름> <로컬 branch 이름>

git push <원격 저장소 이름> --delete <원격 branch 이름>
```

어떤 branch에서 commit, merge 또는 branch 생성을 수행했는지 잘 아는 것이 버전 관리 흐름을 파악하는 데에 중요함.

git pull은 fetch 이후 merge 또는 rebase로 구성되는데, 두 방식 중 뭘 사용할 것인지는 git config pull.rebase false로 지정이 가능함. git config의 pull.rebase 값은 기본적으로 false로, 이는 merge 방식을 의미함. 이 값이 true이면 이는 rebase 방식으로, 병합을 통한 commit 생성이 아니라 단순히 로컬 commit들을 원격 commit들의 뒤쪽으로 재배치하는 것임.

1.4. Git Reset

1.4.1. Commit 가리키기

∧와 ∼으로 특정 부모 commit을 편리하게 가리킬 수 있음.

아래와 같이 commit 이름 끝에 ∧를 붙이면 해당 commit의 부모를 가리킴. ∧를 여러 개 붙일 수도 있고, 뒤에 숫자를 명시에 여러 개의 부모가 존재하는 경우 몇 번째 부모인지를 명시할 수도 있음. 이때 기본적인 merge만 수행된 경우 부모는 최대 2개인데, 첫 번째 부모는 merge 시의 HEAD가 있던 branch의 commit이고, 두 번째 부모는 merge되어 들어온 branch의 commit임.

```
git show HEAD^
git show HEAD^^^
git show HEAD^2
```

이름 뒤에 ∼과 숫자를 명시해 몇 번 위의 부모(조상)인지를 명시할 수 있음. 숫자를 명시하지 않으면 1로 취급됨. 참고로, 해당 숫자만큼 거슬러 올라갈 때 항상 첫 번째 부모 쪽으로만 감.

```
git show HEAD~
git show HEAD~3
```

```
* 734713b fixed refs handling, added gc auto, updated tests
*   d921970 Merge commit 'phedders/rdocs'
|\
| * 35cfb2b Some rdoc changes
* | 1c002dd added some blame and merge stuff
|/
* 1c36188 ignore *.gem
* 9b29157 add open3_detach to gemspec file list
```

```
$ git show d921970^
commit 1c002dd4b536e7479fe34593e72e6c6c1819e53b
Author: Scott Chacon <schacon@gmail.com>
Date:   Thu Dec 11 14:58:32 2008 -0800

    added some blame and merge stuff
```

d921970의 첫 번째 부모(checkout 했던 브랜치)

```
$ git show d921970^2
commit 35cfb2b795a55793d7cc56a6cc2060b4bb732548
Author: Paul Hedderly <paul+git@mjr.org>
Date:   Wed Dec 10 22:22:03 2008 +0000

    Some rdoc changes
```

d921970의 두 번째 부모(Merge 한 대상 브랜치)

```
* 734713b fixed refs handling, added gc auto, updated tests
*   d921970 Merge commit 'phedders/rdocs'
|\
| * 35cfb2b Some rdoc changes
* | 1c002dd added some blame and merge stuff
|/
* 1c36188 ignore *.gem
* 9b29157 add open3_detach to gemspec file list
```

```
$ git show HEAD~3
commit 1c3618887afb5fbcbea25b7c013f4e2114448b8d
Author: Tom Preston-Werner <tom@mojombo.com>
Date:   Fri Nov 7 13:47:59 2008 -0500

    ignore *.gem
```

~ 뒤의 숫자는 조상의 단계를 표시

1.4.2. Git Reset

git reset은 HEAD는 여전히 기존의 branch를 가리키고, branch가 가리키는 commit을 특정 commit으로 변경하는 명령어임.

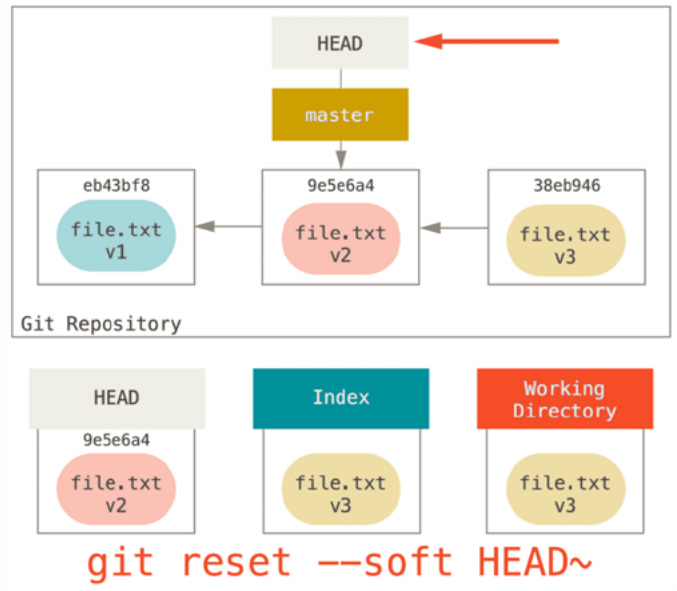
반면 앞에서 본 git checkout은 HEAD가 새로운 branch를 가리키도록 해 branch를 전환하는 명령어로, HEAD, index, working directory 모두가 해당 branch의 마지막 스냅샷을 가지도록 수정됨.

앞에서 다룬 것처럼 로컬 저장소는 HEAD, index, working directory로 구성됨. reset은 이런 파일 구조와 관련하여 동작이 soft, mixed, hard로 구분됨. 이때 commit 이름은 HEAD^, HEAD~, checksum 값 일부 등으로 지정할 수 있음. 또한 당연하게도 원격 트래킹 branch를 지정할 수도 있음.

1. Soft

아래와 같이 git reset에 –soft 옵션을 주는 경우, HEAD만 해당 commit의 스냅샷으로 변경됨. 즉, –soft로 reset하고 다시 staging 및 commit하는 것은 commit –amend와 동작이 동일함(직전 commit을 덮어씀.).

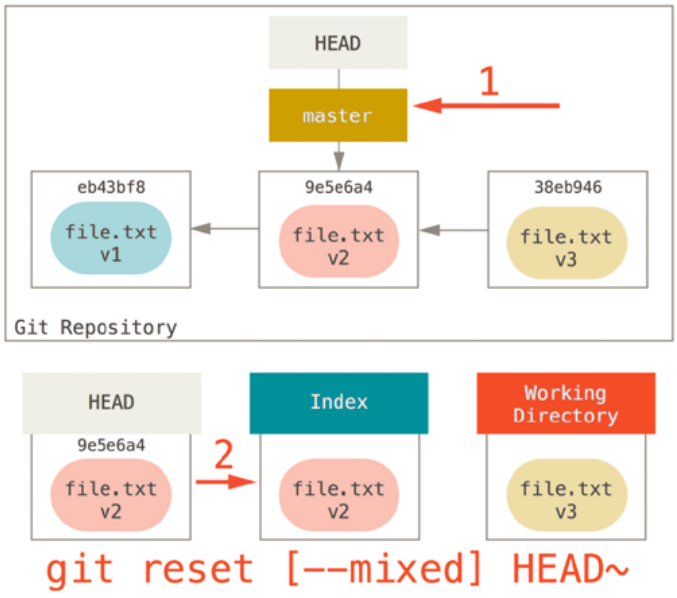
```
git reset --soft <commit 이름>
```



2, Mixed

아래와 같이 git reset에 –mixed 옵션을 주는 경우, HEAD와 index만 해당 commit의 스냅샷으로 변경됨. 옵션을 생략하면 mixed가 default임.

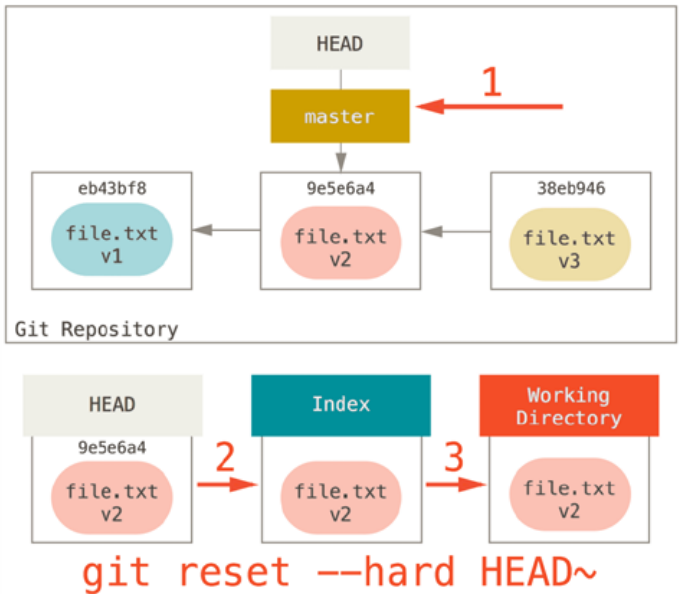
```
git reset --mixed <commit 이름>
```



3. Hard

아래와 같이 git reset에 –hard 옵션을 주는 경우, HEAD, index, working directory 모두 해당 commit의 스냅샷으로 변경됨.

```
git reset --hard <commit 이름>
```



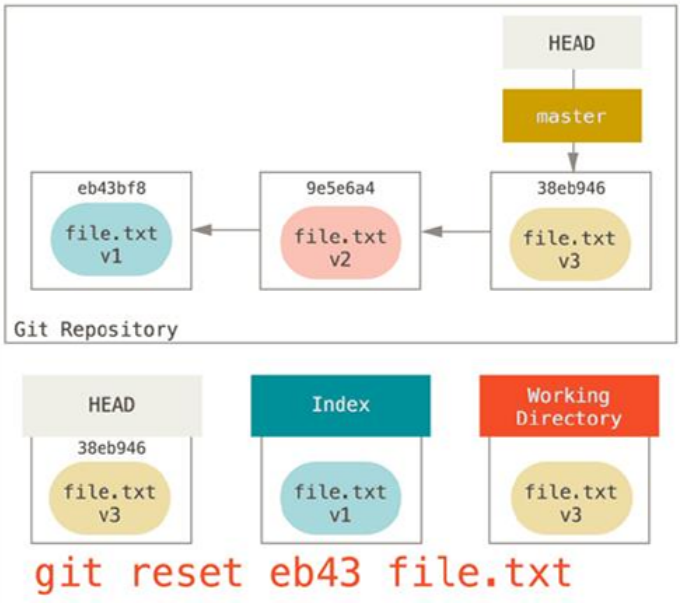
1.4.3. 파일명으로 Reset&Checkout

git reset과 git checkout에는 파일명을 지정할 수도 있는데, 이 경우 동작이 다름.

1. 파일명으로 Reset

아래와 같이 특정 파일명을 지정해 reset하면 해당 파일에 대해 index를 수정할 수 있음. commit 이름을 명시하지 않는 경우 HEAD에 대응되는 commit의 파일 정보가 index에 저장되고(unstage됨.), commit 이름을 명시하는 경우 해당 commit의 파일 정보가 index에 저장됨.

```
git reset <파일명>
git reset <commit 이름> <파일명>
```



2. 파일명으로 Checkout

아래와 같이 특정 파일명을 지정해 checkout하면 해당 파일에 대해 index와 working directory를 수정할 수 있음. branch 또는 commit을 지정한 경우 해당 branch 또는 commit의 파일 정보가 index와 working directory에 저장되고, branch 또는 commit을 지정하지 않은 경우 index의 파일 정보가 working

directory에 저장됨.

```
git checkout <branch 또는 commit 이름> <파일명>
```

1.5. Github를 활용한 기여&협업 방법

1.5.1. Github를 활용한 기여&협업 방법

github를 활용한 기여 또는 협업은 아래와 같은 형태로 이루어질 수 있음.

1. 원격 저장소를 만들어 작업자 별로 branch를 파 작업한 뒤, PR(Pull Request)를 보내 merge함.

2. 원본(upstream) 원격 저장소를 fork해 자신의 저장소에 복사한 뒤, 로컬에 연결해 작업하고, 원본 원격 저장소에 PR(Pull Request)을 보내 merge함.

참고로, fork는 github에서 다른 사람의 원격 저장소를 복사해 자신의 원격 저장소로 생성하는 것을 말함. 이는 github 웹페이지에서 수행할 수 있음. 이때 복사한 것이므로 당연하게도 한쪽에서 단순히 수정하는 경우 다른 쪽에 반영되지 않음.