1-6 Git Stash & 변경사항 취소

7.3 Git 도구 - Stashing과 Cleaning

Stashing과 Cleaning

당신이 어떤 프로젝트에서 한 부분을 담당하고 있다고 하자. 그리고 여기에서 뭔가 작업하던 일이 있고 다른 요청이 들어와서 잠시 브랜치를 변경해야 할 일이 생겼다고 치자. 그런데 이런 상황에서 아직 완료하지 않은 일을 커밋하는 것이 껄끄럽다는 것이 문제다. 커밋하지 않고 나중에 다시 돌아와서 작업을 다시 하고 싶을 것이다. 이 문제는 `git stash` 라는 명령으로 해결할 수 있다.

Stash 명령을 사용하면 워킹 디렉토리에서 수정한 파일들만 저장한다. Stash는 Modified이면서 Tracked 상태인 파일과 Staging Area에 있는 파일들을 보관해두는 장소다. 아직 끝내지 않은 수정사항을 스택에 잠시 저장했다가 나중에 다시 적용할 수 있다(브랜치가 달라져도 말이다).

노트	`git stash push` 로의 이동 2017년 10월 말 Git 메일링 리스트에는 엄청난 논의가 있었습니다. 논의는 `git stash save` 명령을 은퇴시키고 `git stash push` 로 대체하는 내용에 대한 것이었습니다. `git stash push` 명령의 경우 *pathspec* 으로 선택하여 Stash하는 옵션이 추가되었는데 `git stash save` 명령이 지원하지 못하는 것이었습니다. `git stash save` 명령이 곧바로 삭제되는 것은 아니기에 아직 이 명령을 쓰는 것에 대해 걱정할 필요는 없지만 `git stash push` 명령으로 대체하는 것에 대해 생각해볼 필요가 있습니다.

하던 일을 Stash 하기

예제 프로젝트를 하나 살펴보자. 파일을 두 개 수정하고 그 중 하나는 Staging Area에 추가한다. 그리고 `git status` 명령을 실행하면 아래와 같은 결과를 볼 수 있다.

```
$ git status
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

  modified:   index.html

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

  modified:   lib/simplegit.rb
```

이제 브랜치를 변경해 보자. 아직 작업 중인 파일은 커밋할 게 아니라서 모두 Stash 한다. `git stash` 나 `git stash save` 를 실행하면 스택에 새로운 Stash가 만들어진다.

```
$ git stash
Saved working directory and index state \
  "WIP on main: 049d078 added the index file"
HEAD is now at 049d078 added the index file
(To restore them type "git stash apply")
```

대신 워킹 디렉토리는 깨끗해졌다.

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

이제 아무 브랜치나 골라서 쉽게 바꿀 수 있다. 수정하던 것을 스택에 저장했다. 아래와 같이 `git stash list` 를 사용하여 저장한 Stash를 확인한다.

```
$ git stash list
stash@{0}: WIP on main: 049d078 added the index file
stash@{1}: WIP on main: c264051 Revert "added file_size"
stash@{2}: WIP on main: 21d80a5 added number to log
```

Stash 두 개는 원래 있었다. 그래서 현재 총 세 개의 Stash를 사용할 수 있다. 이제 `git stash apply` 를 사용하여 Stash를 다시 적용할 수 있다. `git stash` 명령을 실행하면 Stash를 다시 적용하는 방법도 알려줘서 편리하다. `git stash apply stash@{2}` 처럼 Stash 이름을 입력하면 골라서 적용할 수 있다. 이름이 없으면 Git은 가장 최근의 Stash를 적용한다.

```
$ git stash apply
On branch main
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working
directory)

  modified:   index.html
  modified:   lib/simplegit.rb

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

Git은 Stash에 저장할 때 수정했던 파일들을 복원해준다. 복원할 때의 워킹 디렉토리는 Stash 할 때의 그 브랜치이고 워킹 디렉토리도 깨끗한 상태였다. 하지만 꼭 깨끗한 워킹 디렉토리나 Stash 할 때와 같은 브랜치에 적용해야 하는 것은 아니다. 어떤 브랜치에서 Stash 하고 다른 브랜치로 옮기고서 거기에 Stash를 복원할 수 있다. 그리고 꼭 워킹 디렉토리가 깨끗한 상태일 필요도 없다. 워킹 디렉토리에 수정하고 커밋하지 않은 파일들이 있을 때도 Stash를 적용할 수 있다. 만약 충돌이 있으면 알려준다.

Git은 Stash를 적용할 때 Staged 상태였던 파일을 자동으로 다시 Staged 상태로 만들어 주지 않는다. 그래서 `git stash apply` 명령을 실행할 때 `--index` 옵션을 주어 Staged 상태까지 적용한다. 그래야 원래 작업하던 상태로 돌아올 수 있다.

```
$ git stash apply --index
On branch main
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

  modified:   index.html

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

  modified:   lib/simplegit.rb
```

`apply` 옵션은 단순히 Stash를 적용하는 것뿐이다. Stash는 여전히 스택에 남아 있다. `git stash drop` 명령을 사용하여 해당 Stash를 제거한다.

```
$ git stash list
stash@{0}: WIP on main: 049d078 added the index file
stash@{1}: WIP on main: c264051 Revert "added file_size"
stash@{2}: WIP on main: 21d80a5 added number to log
$ git stash drop stash@{0}
Dropped stash@{0} (364e91f3f268f0900bc3ee613f9f733e82aaed43)
```

그리고 `git stash pop` 이라는 명령도 있는데 이 명령은 Stash를 적용하고 나서 바로 스택에서 제거해준다.

Stash를 만드는 새로운 방법

Stash를 만드는 방법은 여러 가지다. 주로 사용하는 옵션으로 `stash save` 명령과 같이 쓰는 `--keep-index` 이다. 이 옵션을 이용하면 이미 Staging Area에 들어 있는 파일을 Stash 하지 않는다.

```
$ git status -s
M  index.html
 M lib/simplegit.rb

$ git stash --keep-index
Saved working directory and index state WIP on main: 1b65b17 added the
index file
HEAD is now at 1b65b17 added the index file

$ git status -s
M  index.html
```

추적하지 않는 파일과 추적 중인 파일을 같이 Stash 하는 일도 꽤 빈번하다. 기본적으로 `git stash` 는 추적 중인 파일만 저장한다. 추적 중이지 않은 파일을 같이 저장하려면 Stash 명령을 사용할 때 `--include-untracked` 나 `-u` 옵션을 붙여준다.

```
$ git status -s
M  index.html
 M lib/simplegit.rb
?? new-file.txt

$ git stash -u
Saved working directory and index state WIP on main: 1b65b17 added the
index file
HEAD is now at 1b65b17 added the index file

$ git status -s
$
```

끝으로 `--patch` 옵션을 붙이면 Git은 수정된 모든 사항을 저장하지 않는다. 대신 대화형 프롬프트가 뜨며 변경된 데이터 중 저장할 것과 저장하지 않을 것을 지정할 수 있다.

```
$ git stash --patch
diff --git a/lib/simplegit.rb b/lib/simplegit.rb
index 66d332e..8bb5674 100644
--- a/lib/simplegit.rb
+++ b/lib/simplegit.rb
@@ -16,6 +16,10 @@ class SimpleGit
         return `#{git_cmd} 2>&1`.chomp
       end
     end
+
+    def show(treeish = 'main')
+      command("git show #{treeish}")
+    end

 end
 test
Stash this hunk [y,n,q,a,d,/,e,?]? y

Saved working directory and index state WIP on main: 1b65b17 added the
index file
```

Stash를 적용한 브랜치 만들기

보통 Stash에 저장하면 한동안 그대로 유지한 채로 그 브랜치에서 계속 새로운 일을 한다. 그러면 이제 저장한 Stash를 적용하는 것이 문제가 된다. 수정한 파일에 Stash를 적용하면 충돌이 일어날 수도 있고 그러면 또 충돌을 해결해야 한다. 필요한 것은 Stash 한 것을 쉽게 다시 테스트하는 것이다. `git stash branch <>` 명령을 실행하면 Stash 할 당시의 커밋을 Checkout 한 후 새로운 브랜치를 만들고 여기에 적용한다. 이 모든 것이 성공하면 Stash를 삭제한다.

```
$ git stash branch testchanges
M index.html
M lib/simplegit.rb
Switched to a new branch 'testchanges'
On branch testchanges
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

  modified:   index.html

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

  modified:   lib/simplegit.rb

Dropped refs/stash@{0} (29d385a81d163dfd45a452a2ce816487a6b8b014)
```

이 명령은 브랜치를 새로 만들고 Stash를 복원해주는 매우 편리한 도구다.

워킹 디렉토리 청소하기

작업하고 있던 파일을 Stash 하지 않고 단순히 그 파일들을 치워버리고 싶을 때가 있다. `git clean` 명령이 그 일을 한다.

보통은 Merge나 외부 도구가 만들어낸 파일을 지우거나 이전 빌드 작업으로 생성된 각종 파일을 지우는 데 필요하다.

이 명령을 사용할 때는 신중해야 한다. 이 명령을 사용하면 워킹 디렉토리 안의 추적하고 있지 않은 모든 파일이 지워지기 때문이다. 명령을 실행하고 나서 후회해도 소용없다. 지워진 파일은 돌아오지 않는다. `git stash –all` 명령을 이용하면 지우는 건 똑같지만, 먼저 모든 파일을 Stash 하므로 좀 더 안전하다.

워킹 디렉토리의 불필요한 파일들을 전부 지우려면 `git clean` 을 사용한다. 추적 중이지 않은 모든 정보를 워킹 디렉토리에서 지우고 싶다면 `git clean -f -d` 명령을 사용하자. 이 명령은 하위 디렉토리까지 모두 지워버린다. `-f` 옵션은 강제(force)의 의미이며 "진짜로 그냥 해라"라는 뜻이다.

이 명령을 실행했을 때 어떤 일이 일어날지 미리 보고 싶다면 `-n` 옵션을 사용한다. `-n` 옵션은 “가상으로 실행해보고 어떤 파일들이 지워질지 알려달라” 라는 뜻이다.

```
$ git clean -d -n
Would remove test.o
Would remove tmp/
```

`git clean` 명령은 추적 중이지 않은 파일만 지우는 게 기본 동작이다. `.gitignore` 에 명시했거나 해서 무시되는 파일은 지우지 않는다. 무시된 파일까지 함께 지우려면 `-x` 옵션이 필요하다. 그래서 `.o` 파일 같은 빌드 파일까지도 지울 수 있다.

```
$ git status -s
 M lib/simplegit.rb
?? build.TMP
?? tmp/

$ git clean -n -d
Would remove build.TMP
Would remove tmp/

$ git clean -n -d -x
Would remove build.TMP
Would remove test.o
Would remove tmp/
```

`git clean` 이 무슨 짓을 할지 확신이 안들 때는 항상 `-n` 옵션을 붙여서 먼저 실행해보자. clean 명령을 대화형으로 실행하려면 `-i` 옵션을 붙이면 된다.

대화형으로 실행한 clean 명령의 모습은 아래와 같다.

```
$ git clean -x -i
Would remove the following items:
  build.TMP  test.o
*** Commands ***
    1: clean                2: filter by pattern    3: select by
numbers    4: ask each             5: quit
    6: help
What now>
```

대화형으로 실행하면 파일마다 지우지 말지 결정하거나 특정 패턴으로 걸러서 지울 수도 있다.

노트	작업하던 저장소가 완전 지저분해져서 Git에게 진짜로 강제로 정리하도록 해야 하는 경우가 생길 수 있다. 예를 들어 Git 버전관리 데이터가 포함된 디렉토리를 복사해왔거나 서브모듈 디렉토리에 문제가 생겼거나 하는 경우 `git clean -fd` 옵션으로 실행한 명령이라도 디렉토리 삭제가 되지 않는 경우가 있다. 이런 경우에는 `-f` 옵션을 한번 더 사용하여 강제성을 추가로 주어야 한다.

7.7 Git 도구 - Reset 명확히 알고 가기

Reset 명확히 알고 가기

Git의 다른 특별한 도구를 더 살펴보기 보기 전에 `reset` 과 `checkout` 에 대해 이야기를 해보자. 이 두 명령은 Git을 처음 사용하는 사람을 가장 헷갈리게 하는 부분이다. 제대로 이해하고 사용할 수 없을 것으로 보일 정도로 많은 기능을 지녔다. 이해하기 쉽게 간단한 비유를 들어 설명해보자.

세 개의 트리

Git을 서로 다른 세 트리를 관리하는 컨텐츠 관리자로 생각하면 `reset` 과 `checkout` 을 좀 더 쉽게 이해할 수 있다. 여기서 “트리” 란 실제로는 “파일의 묶음” 이다. 자료구조의 트리가 아니다 세 트리 중 Index는 트리도 아니지만, 이해를 쉽게 하려고 일단 트리라고 한다.

Git은 일반적으로 세 가지 트리를 관리하는 시스템이다.

트리	역할

HEAD	마지막 커밋 스냅샷, 다음 커밋의 부모 커밋
Index	다음에 커밋할 스냅샷
워킹 디렉토리	샌드박스

HEAD

HEAD는 현재 브랜치를 가리키는 포인터이며, 브랜치는 브랜치에 담긴 커밋 중 가장 마지막 커밋을 가리킨다. 지금의 HEAD가 가리키는 커밋은 바로 다음 커밋의 부모가 된다. 단순하게 생각하면 HEAD는 *현재 브랜치 마지막 커밋의 스냅샷*이다.

HEAD가 가리키는 스냅샷을 살펴보기는 쉽다. 아래는 HEAD 스냅샷의 디렉토리 리스팅과 각 파일의 SHA-1 체크섬을 보여주는 예제다.

```
$ git cat-file -p HEAD
tree cfda3bf379e4f8dba8717dee55aab78aef7f4daf
author Scott Chacon  1301511835 -0700
committer Scott Chacon  1301511835 -0700

initial commit

$ git ls-tree -r HEAD
100644 blob a906cb2a4a904a152...   README
100644 blob 8f94139338f9404f2...   Rakefile
040000 tree 99f1a6d12cb4b6f19...   lib
```

`cat-file` 와 `ls-tree` 명령은 일상적으로는 잘 사용하지 않는 저수준 명령이다. 이런 저수준 명령을 "plumbing" 명령이라고 한다. Git이 실제로 무슨 일을 하는지 볼 때 유용하다.

Index

Index는 **바로 다음에 커밋할** 것들이다. 이미 앞에서 우리는 이런 개념을 "Staging Area" 라고 배운 바 있다. "Staging Area" 는 사용자가 `git commit` 명령을 실행했을 때 Git이 처리할 것들이 있는 곳이다.

먼저 Index는 워킹 디렉토리에서 마지막으로 Checkout 한 브랜치의 파일 목록과 파일 내용으로 채워진다. 이후 파일 변경작업을 하고 변경한 내용으로 Index를 업데이트 할 수 있다. 이렇게 업데이트 하고 `git commit` 명령을 실행하면 Index는 새 커밋으로 변환된다.

```
$ git ls-files -s
100644 a906cb2a4a904a152e80877d4088654daad0c859 0       README
100644 8f94139338f9404f26296befa88755fc2598c289 0       Rakefile
100644 47c6340d6459e05787f644c2447d2595f5d3a54b 0       lib/simplegit.
rb
```

또 다른 저수준 `git ls-files` 명령은 훨씬 더 장막 뒤에 가려져 있는 명령으로 이를 실행하면 현재 Index가 어떤 상태인지를 확인할 수 있다.

Index는 엄밀히 말해 트리구조는 아니다. 사실 Index는 평평한 구조로 구현되어 있다. 여기에서는 쉽게 이해할 수 있도록 그냥 트리라고 설명한다.

워킹 디렉토리

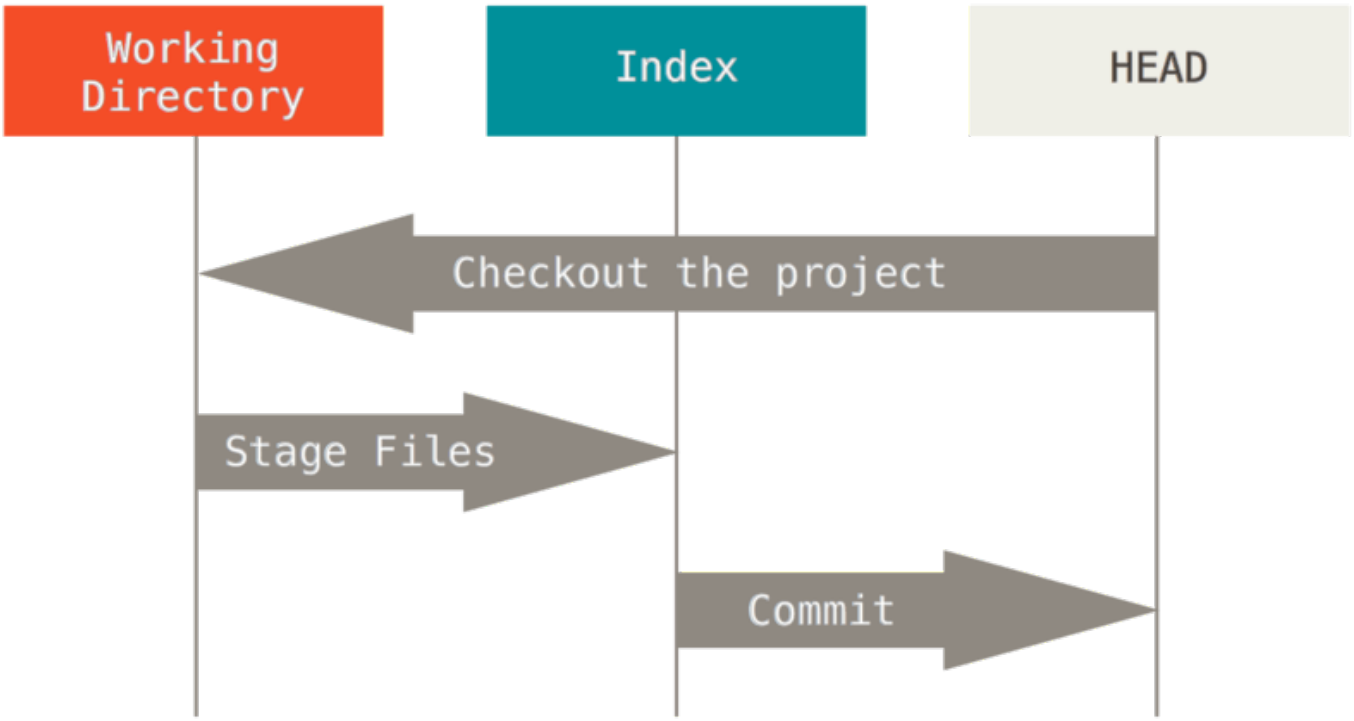
마지막으로 워킹 디렉토리를 살펴보자. 위의 두 트리는 파일과 그 내용을 효율적인 형태로 `.git` 디렉토리에 저장한다. 하지만, 사람이 알아보기 어렵다. 워킹 디렉토리는 실제 파일로 존재한다. 바로 눈에 보이기 때문에 사용자가 편집하기 수월하다. 워킹 디렉토리는 **샌드박스**로 생각하자. 커밋하기 전에는 Index(Staging Area)에 올려놓고 얼마든지 변경할 수 있다.

```
$ tree
.
 README
 Rakefile
 lib
     simplegit.rb

1 directory, 3 files
```

워크플로

Git의 주목적은 프로젝트의 스냅샷을 지속적으로 저장하는 것이다. 이 트리 세 개를 사용해 더 나은 상태로 관리한다.

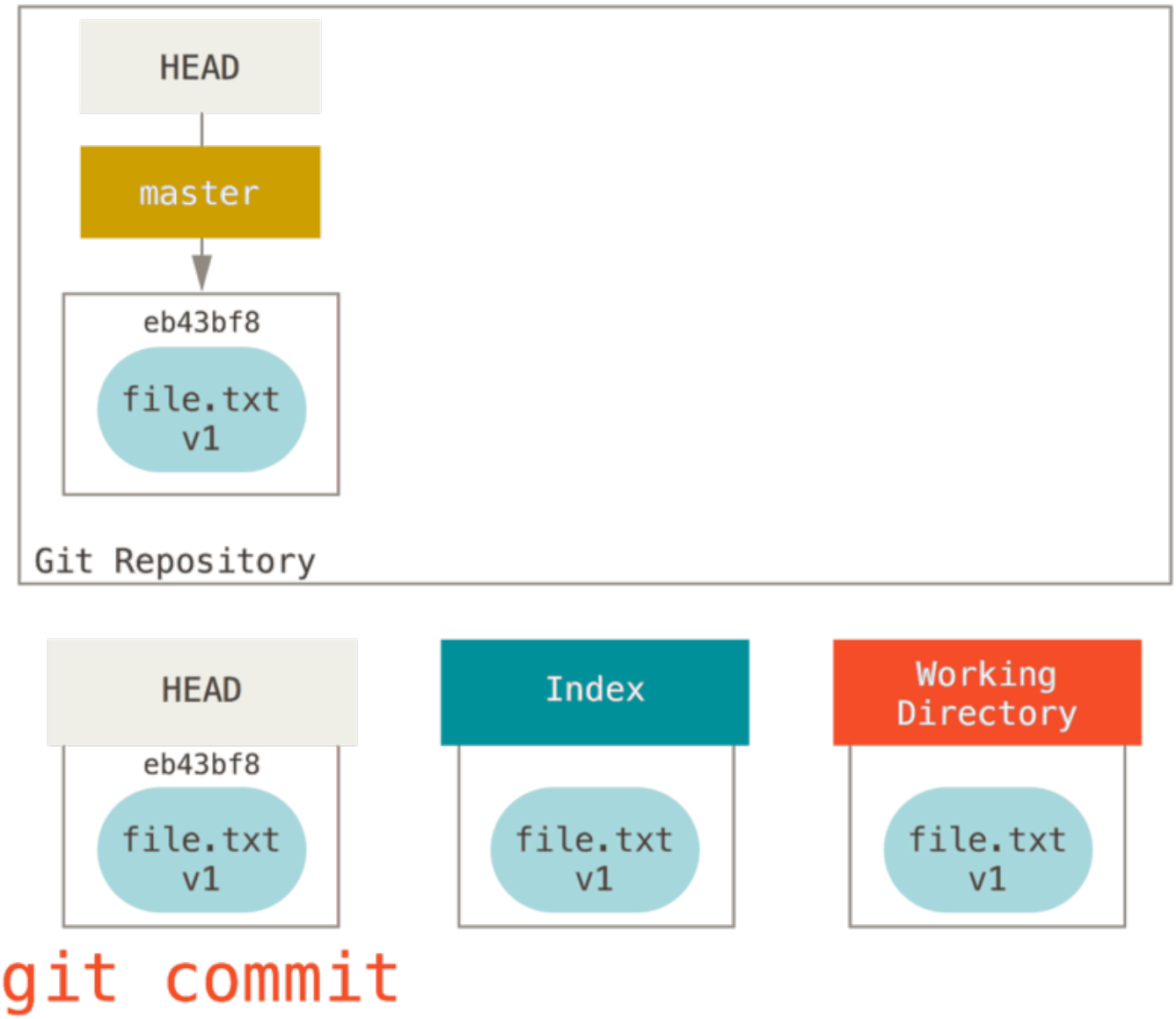


이 과정을 시각화해보자. 파일이 하나 있는 디렉토리로 이동한다. 이걸 파일의 **v1**이라고 하고 파란색으로 표시한다. `git init` 명령을 실행하면 Git 저장소가 생기고 HEAD는 아직 없는 브랜치를 가리킨다(`main` 는 아직 없다).

이 시점에서는 워킹 디렉토리 트리에만 데이터가 있다.

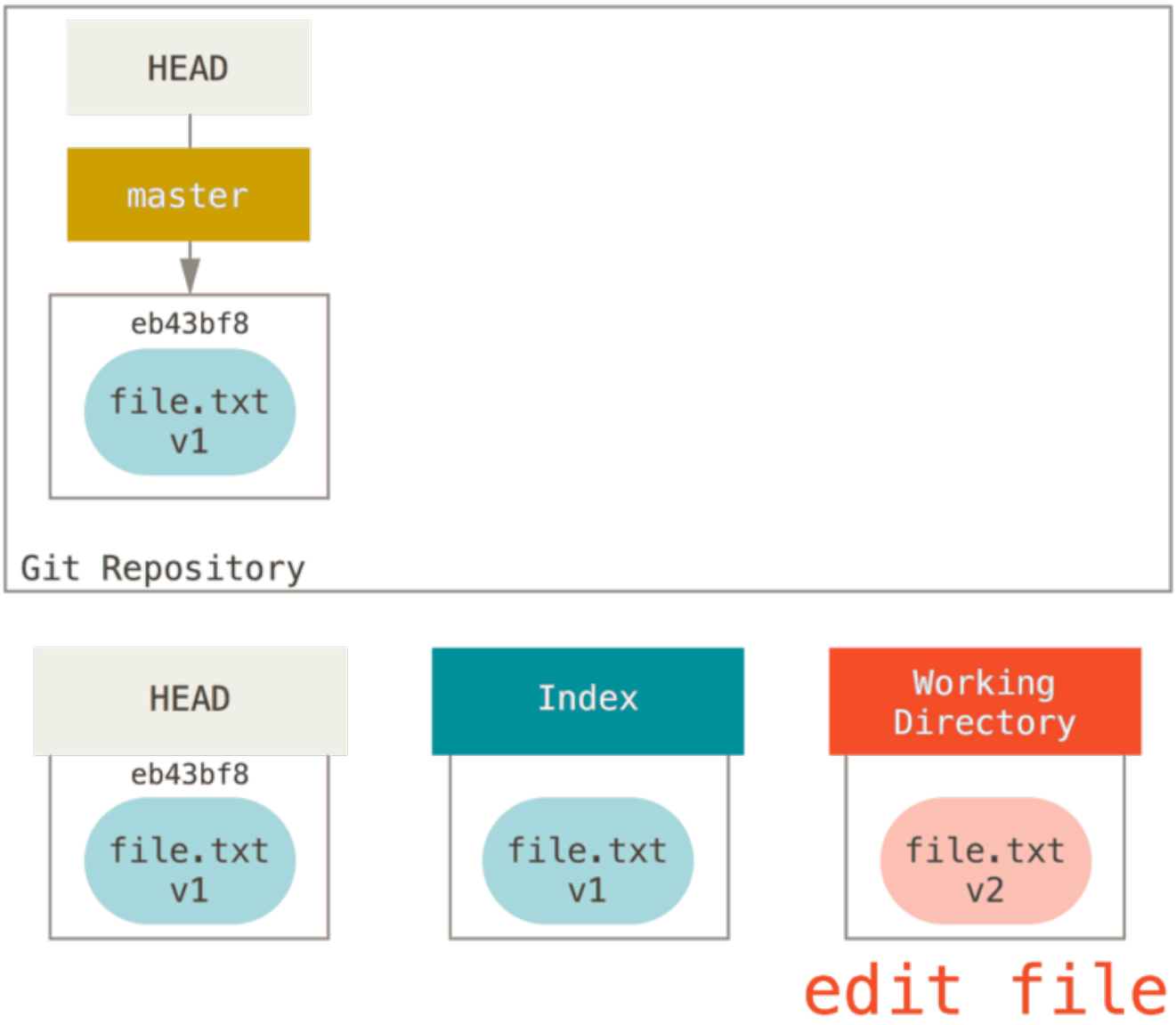
이제 파일을 커밋해보자. `git add` 명령으로 워킹 디렉토리의 내용을 Index로 복사한다.

그리고 `git commit` 명령을 실행한다. 그러면 Index의 내용을 스냅샷으로 영구히 저장하고 그 스냅샷을 가리키는 커밋 객체를 만든다. 그리고는 `main`가 그 커밋 객체를 가리키도록 한다.

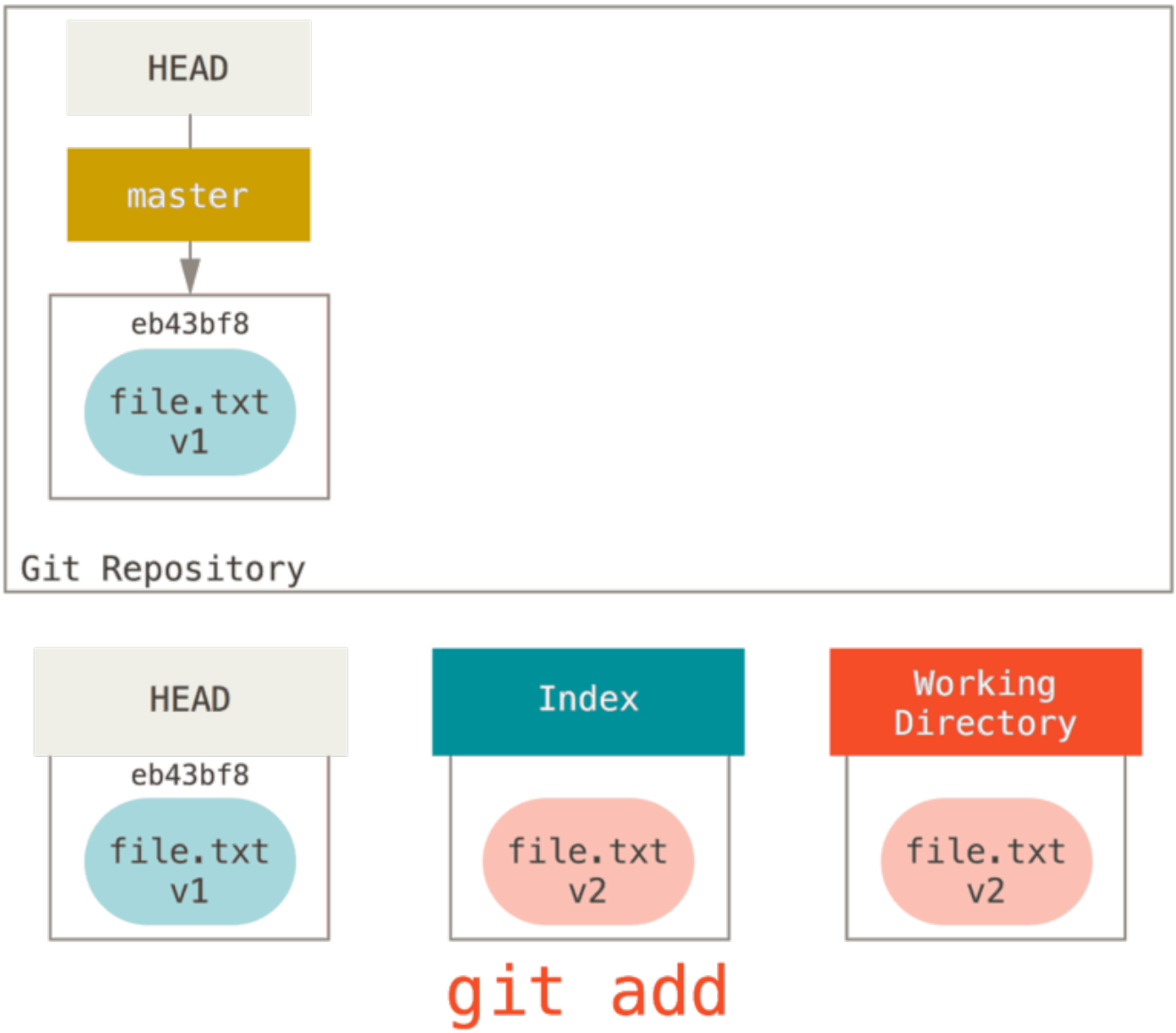


이때 `git status` 명령을 실행하면 아무런 변경 사항이 없다고 나온다. 세 트리 모두가 같기 때문이다.

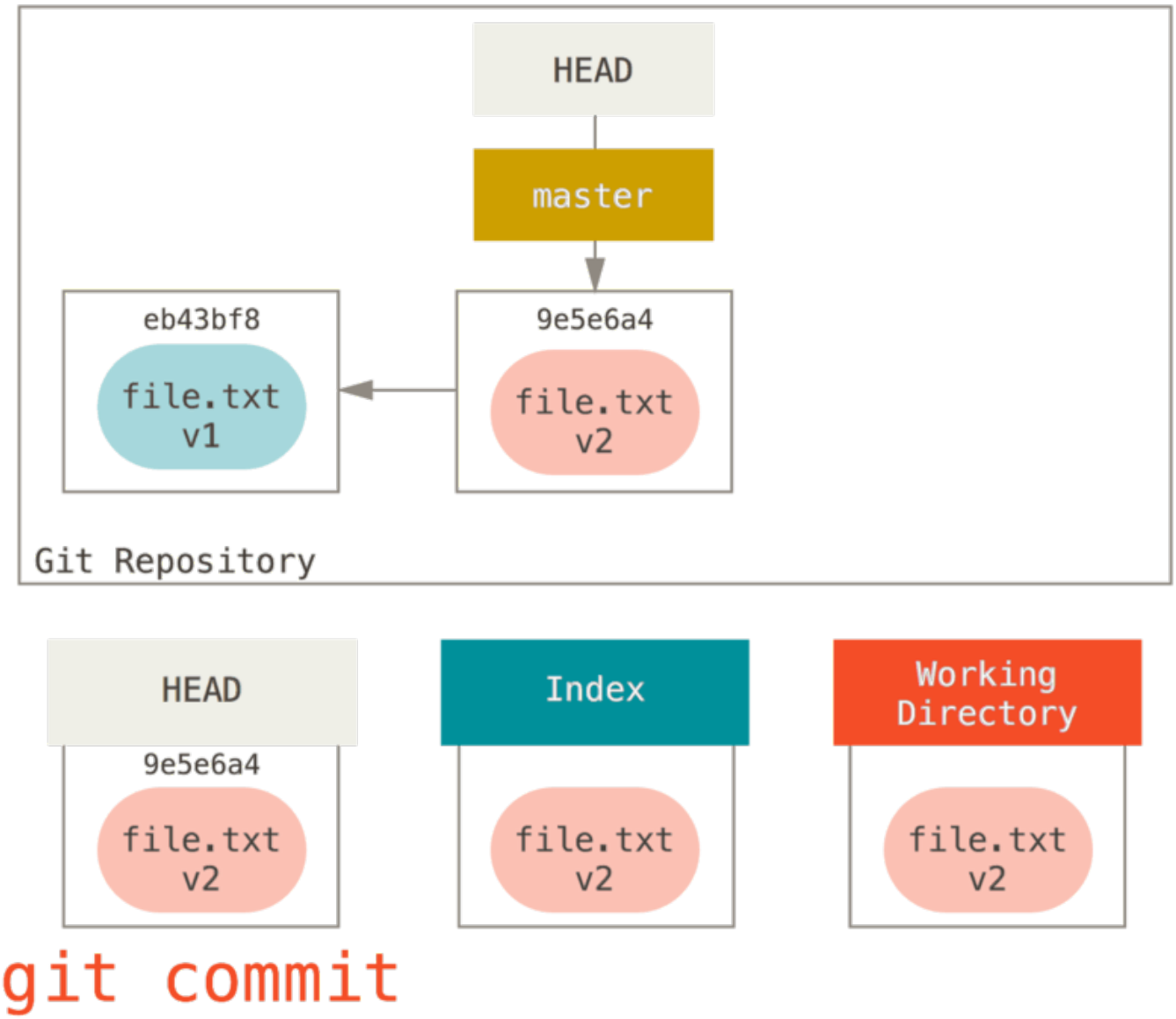
다시 파일 내용을 바꾸고 커밋해보자. 위에서 했던 것과 과정은 비슷하다. 먼저 워킹 디렉토리의 파일을 고친다. 이를 이 파일의 **v2**라고 하자. 이건 빨간색으로 표시한다.



`git status` 명령을 바로 실행하면 “Changes not staged for commit,” 아래에 빨간색으로 된 파일을 볼 수 있다. Index와 워킹 디렉토리가 다른 내용을 담고 있기 때문에 그렇다. `git add` 명령을 실행해서 변경 사항을 Index에 올려주자.



이 시점에서 `git status` 명령을 실행하면 “Changes to be committed” 아래에 파일 이름이 녹색으로 변한다. Index와 HEAD의 다른 파일들이 여기에 표시된다. 즉 다음 커밋할 것과 지금 마지막 커밋이 다르다는 말이다. 마지막으로 `git commit` 명령을 실행해 커밋한다.



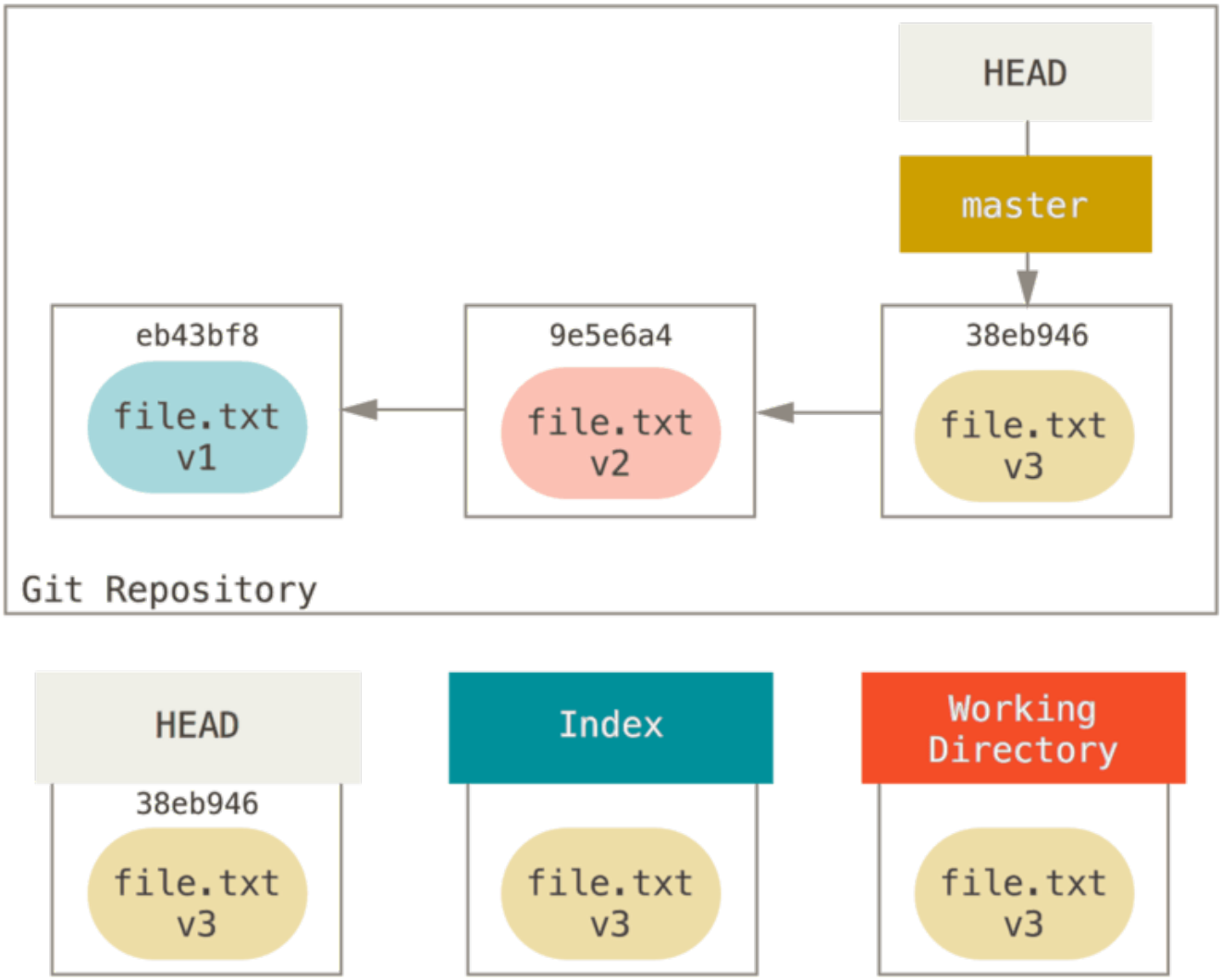
이제 `git status` 명령을 실행하면 아무것도 출력하지 않는다. 세 개의 트리의 내용이 다시 같아졌기 때문이다.

브랜치를 바꾸거나 Clone 명령도 내부에서는 비슷한 절차를 밟는다. 브랜치를 Checkout 하면, **HEAD**가 새로운 브랜치를 가리키도록 바뀌고, 새로운 커밋의 스냅샷을 **Index**에 놓는다. 그리고 Index의 내용을 **워킹 디렉토리**로 복사한다.

Reset 의 역할

위의 트리 세 개를 이해하면 `reset` 명령이 어떻게 동작하는지 쉽게 알 수 있다.

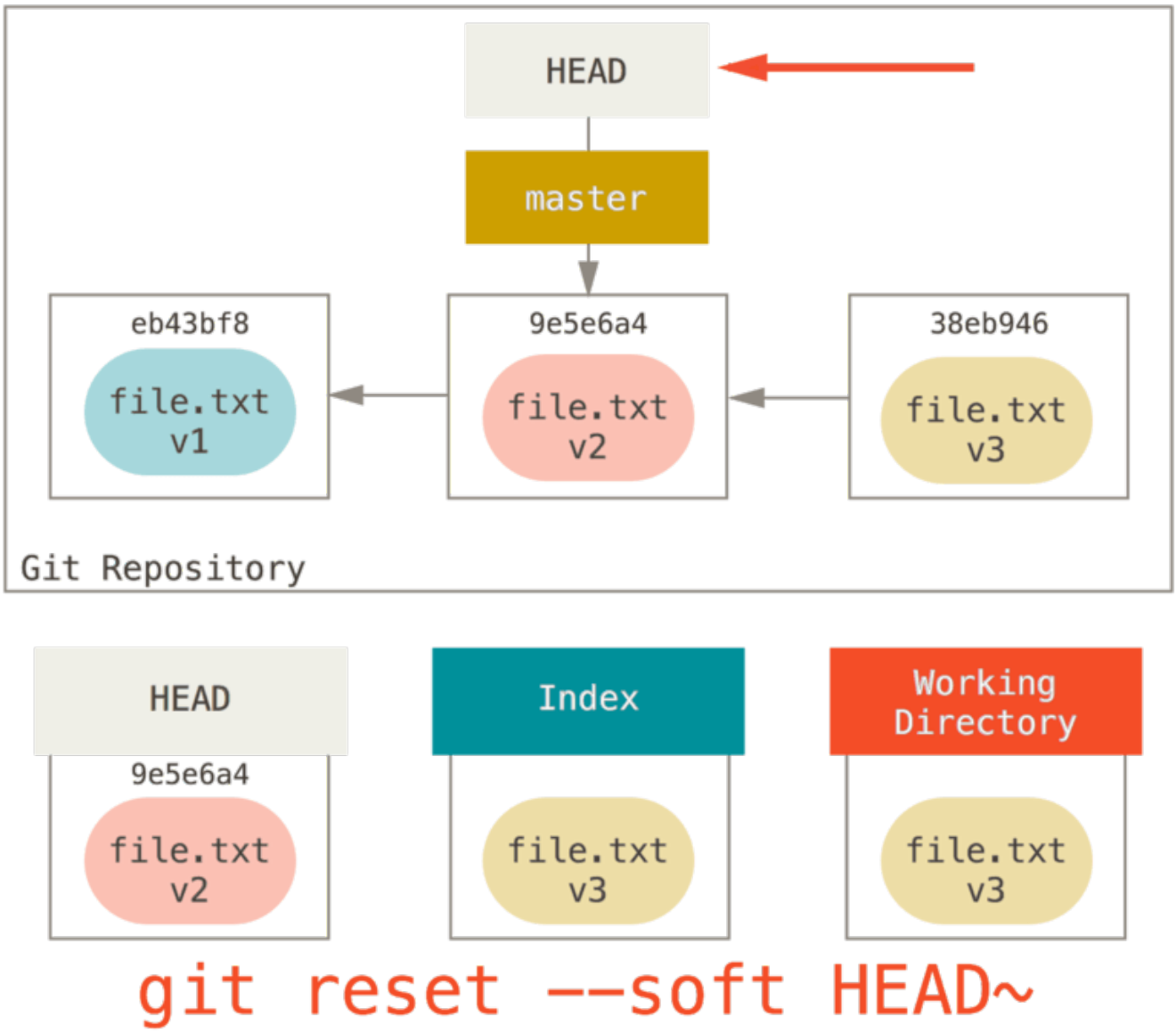
예로 들어 `file.txt` 파일 하나를 수정하고 커밋한다. 이것을 세 번 반복한다. 그러면 히스토리는 아래와 같이 된다.



자 이제 `reset` 명령이 정확히 어떤 일을 하는지 낱낱이 파헤쳐보자. `reset` 명령은 이 세 트리를 간단하고 예측 가능한 방법으로 조작한다. 트리를 조작하는 동작은 세 단계 이하로 이루어진다.

1 단계: HEAD 이동

`reset` 명령이 하는 첫 번째 일은 HEAD 브랜치를 이동시킨다. `checkout` 명령처럼 HEAD가 가리키는 브랜치를 바꾸지는 않는다. HEAD는 계속 현재 브랜치를 가리키고 있고, 현재 브랜치가 가리키는 커밋을 바꾼다. HEAD가 `main` 브랜치를 가리키고 있다면(즉 `main` 브랜치를 Checkout 하고 작업하고 있다면) `git reset 9e5e6a4` 명령은 `main` 브랜치가 `9e5e6a4` 를 가리키게 한다.



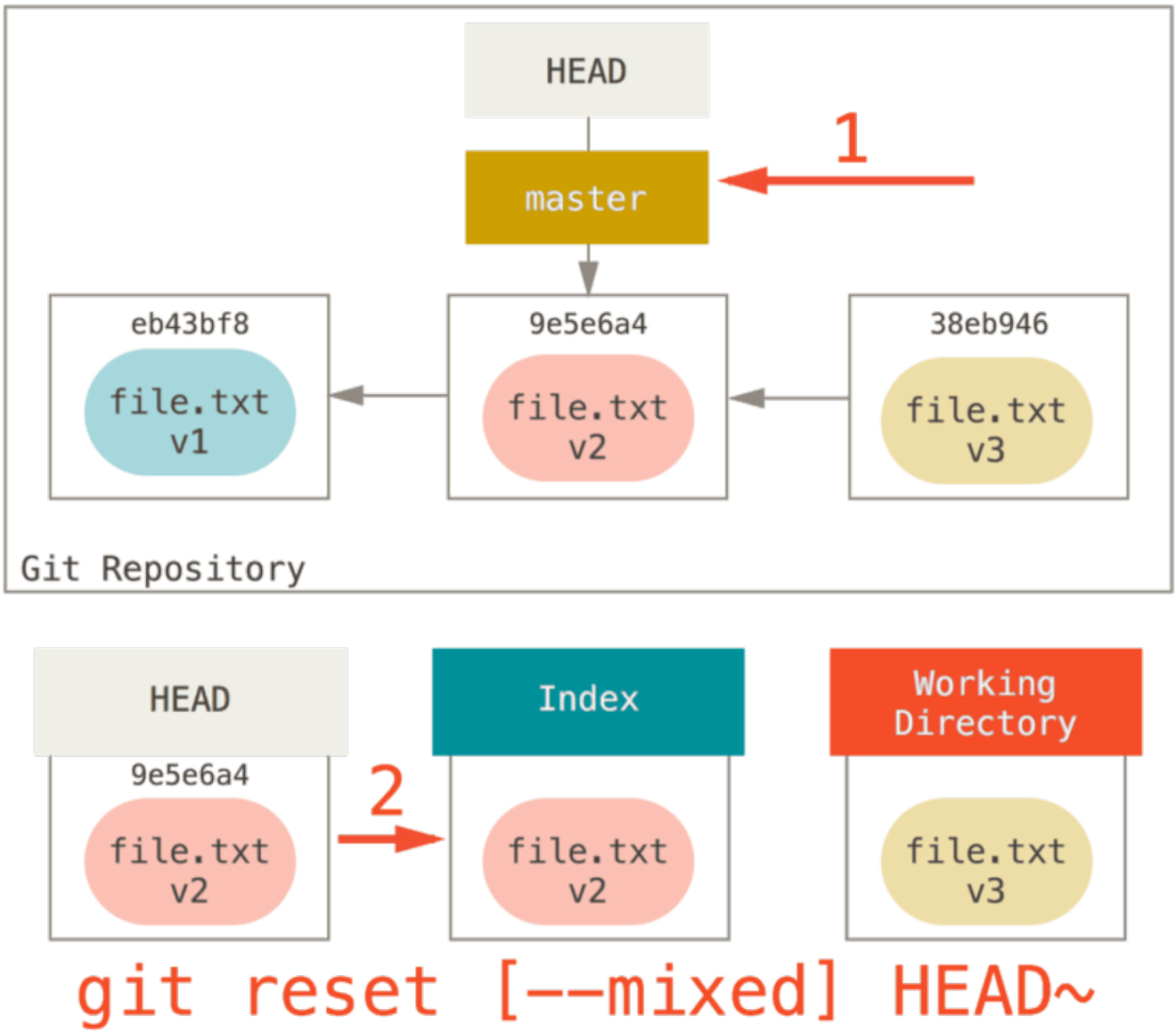
`reset` 명령에 커밋을 넘기고 실행하면 언제나 이런 작업을 수행한다. `reset --soft` 옵션을 사용하면 딱 여기까지 진행하고 동작을 멈춘다.

이제 위의 다이어그램을 보고 어떤 일이 일어난 것인지 생각해보자. `reset` 명령은 가장 최근의 `git commit` 명령을 되돌린다. `git commit` 명령을 실행하면 Git은 새로운 커밋을 생성하고 HEAD가 가리키는 브랜치가 새로운 커밋을 가리키도록 업데이트한다. `reset` 명령 뒤에 `HEAD~` (HEAD의 부모 커밋)를 주면 Index나 워킹 디렉토리는 그대로 놔두고 브랜치가 가리키는 커밋만 이전으로 되돌린다. Index를 업데이트한 다음에 `git commit` 명령를 실행하면 `git commit --amend` 명령의 결과와 같아진다(마지막 커밋을 수정하기를 참조).

2 단계: Index 업데이트 (--mixed)

여기서 `git status` 명령을 실행하면 Index와 `reset` 명령으로 이동시킨 HEAD의 다른 점이 녹색으로 출력된다.

`reset` 명령은 여기서 한 발짝 더 나아가 Index를 현재 HEAD가 가리키는 스냅샷으로 업데이트할 수 있다.

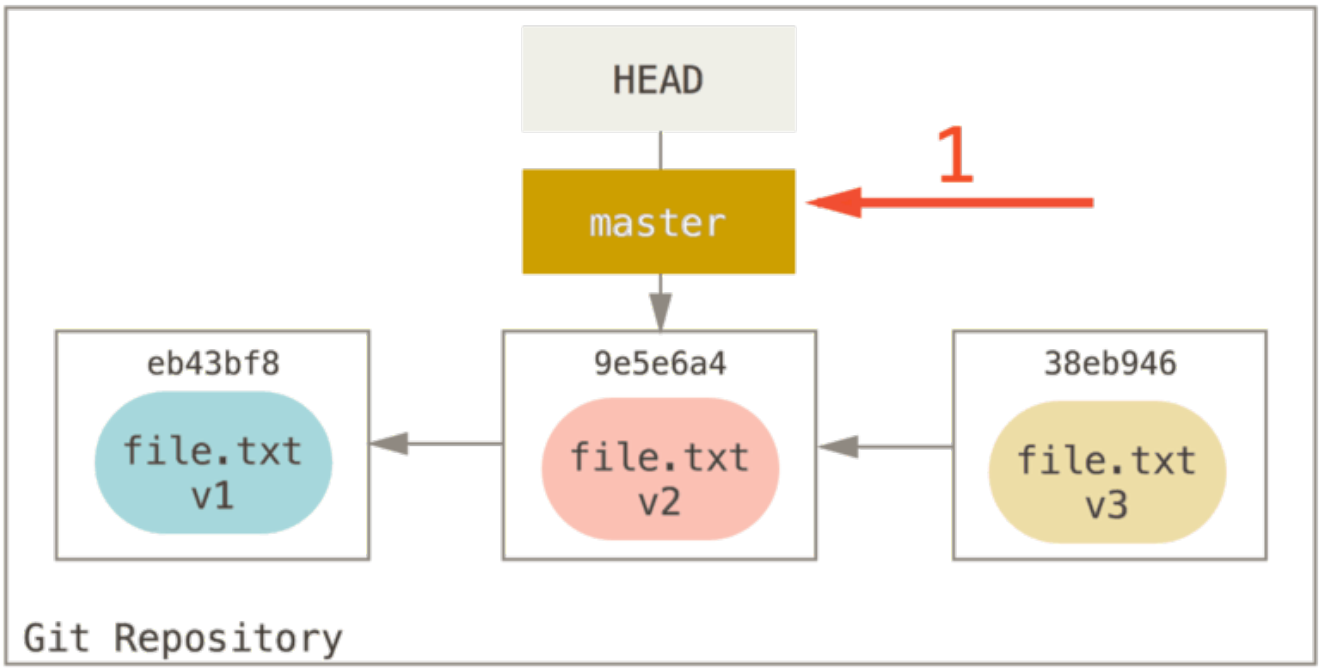


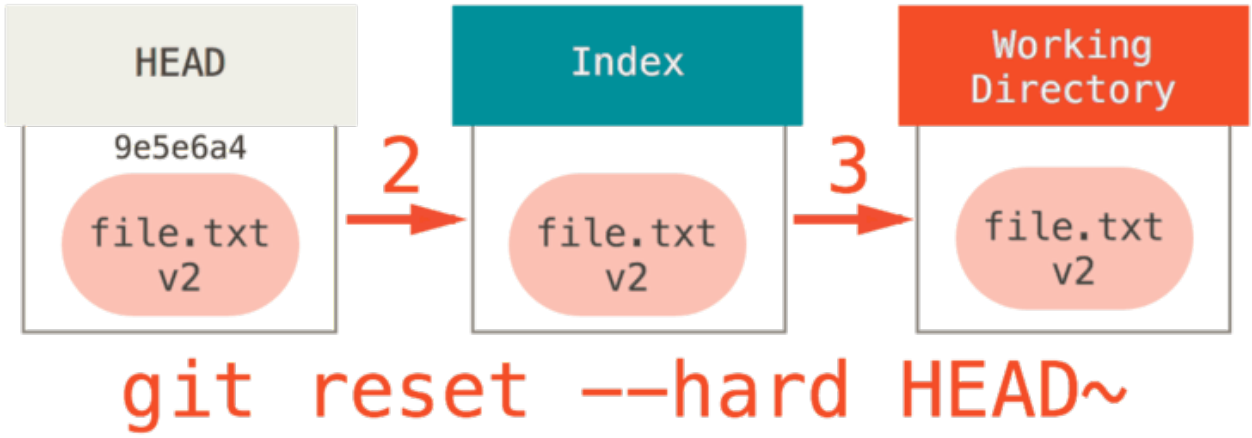
`--mixed` 옵션을 주고 실행하면 `reset` 명령은 여기까지 하고 멈춘다. `reset` 명령을 실행할 때 아무 옵션도 주지 않으면 기본적으로 `--mixed` 옵션으로 동작한다(예제와 같이 `git reset HEAD~` 처럼 명령을 실행하는 경우).

위의 다이어그램을 보고 어떤 일이 일어날지 한 번 더 생각해보자. 가리키는 대상을 가장 최근의 으로 되돌리는 것은 같다. 그러고 나서 *Staging Area* 를 비우기까지 한다. `git commit` 명령도 되돌리고 `git add` 명령까지 되돌리는 것이다.

3 단계: 워킹 디렉토리 업데이트 (--hard)

`reset` 명령은 세 번째로 워킹 디렉토리까지 업데이트한다. `--hard` 옵션을 사용하면 `reset` 명령은 이 단계까지 수행한다.





이 과정은 어떻게 동작하는지 가늠해보자. `reset` 명령을 통해 `git add` 와 `git commit` 명령으로 생성한 마지막 커밋을 되돌린다. **그리고** 워킹 디렉토리의 내용까지도 되돌린다.

이 `--hard` 옵션은 매우 매우 중요하다. `reset` 명령을 위험하게 만드는 유일한 옵션이다. Git에는 데이터를 실제로 삭제하는 방법이 별로 없다. 이 삭제하는 방법은 그 중 하나다. `reset` 명령을 어떻게 사용하더라도 간단히 결과를 되돌릴 수 있다. 하지만 `--hard` 옵션은 되돌리는 것이 불가능하다. 이 옵션을 사용하면 워킹 디렉토리의 파일까지 강제로 덮어쓴다. 이 예제는 파일의 **v3**버전을 아직 Git이 커밋으로 보관하고 있기 때문에 `reflog` 를 이용해서 다시 복원할 수 있다. 만약 커밋한 적 없다면 Git이 덮어쓴 데이터는 복원할 수 없다.

복습

`reset` 명령은 정해진 순서대로 세 개의 트리를 덮어써 나가다가 옵션에 따라 지정한 곳에서 멈춘다.

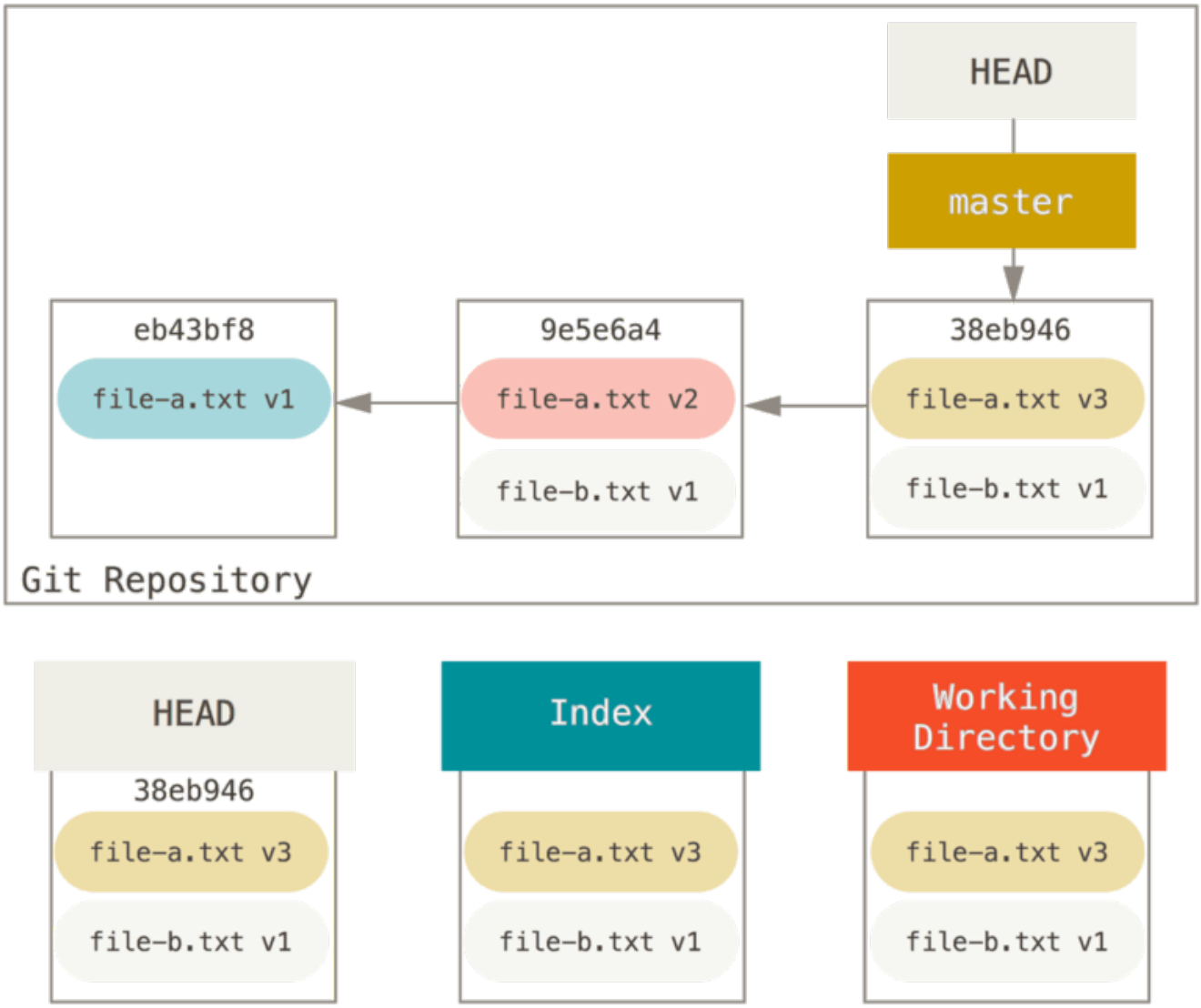
1. HEAD가 가리키는 브랜치를 옮긴다. *(`--soft` 옵션이 붙으면 여기까지)*
2. Index를 HEAD가 가리키는 상태로 만든다. *(`--hard` 옵션이 붙지 않았으면 여기까지)*
3. 워킹 디렉토리를 Index의 상태로 만든다.

합치기(Squash)

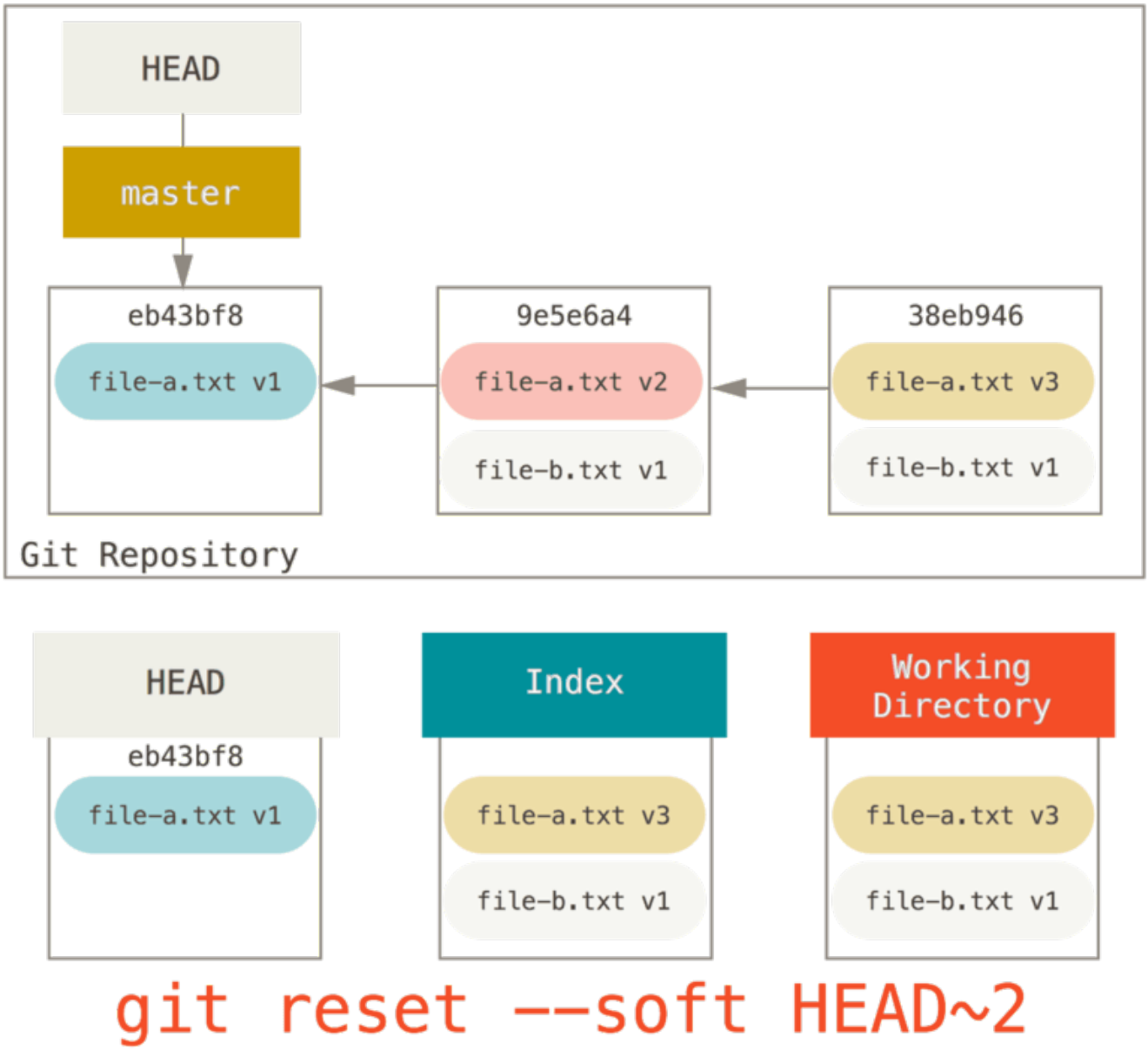
여러 커밋을 커밋 하나로 합치는 재밌는 도구를 알아보자.

"oops." 나 "WIP", "forgot this file" 같은 깃털같이 가벼운 커밋들이 있다고 해보자. 이럴 때는 `reset` 명령으로 커밋들을 하나로 합쳐서 남들에게 똑똑한 척할 수 있다. (커밋 합치기를 하는 명령어가 따로 있지만, 여기서는 `reset` 명령을 쓰는 것이 더 간단할 때도 있다는 것을 보여준다.)

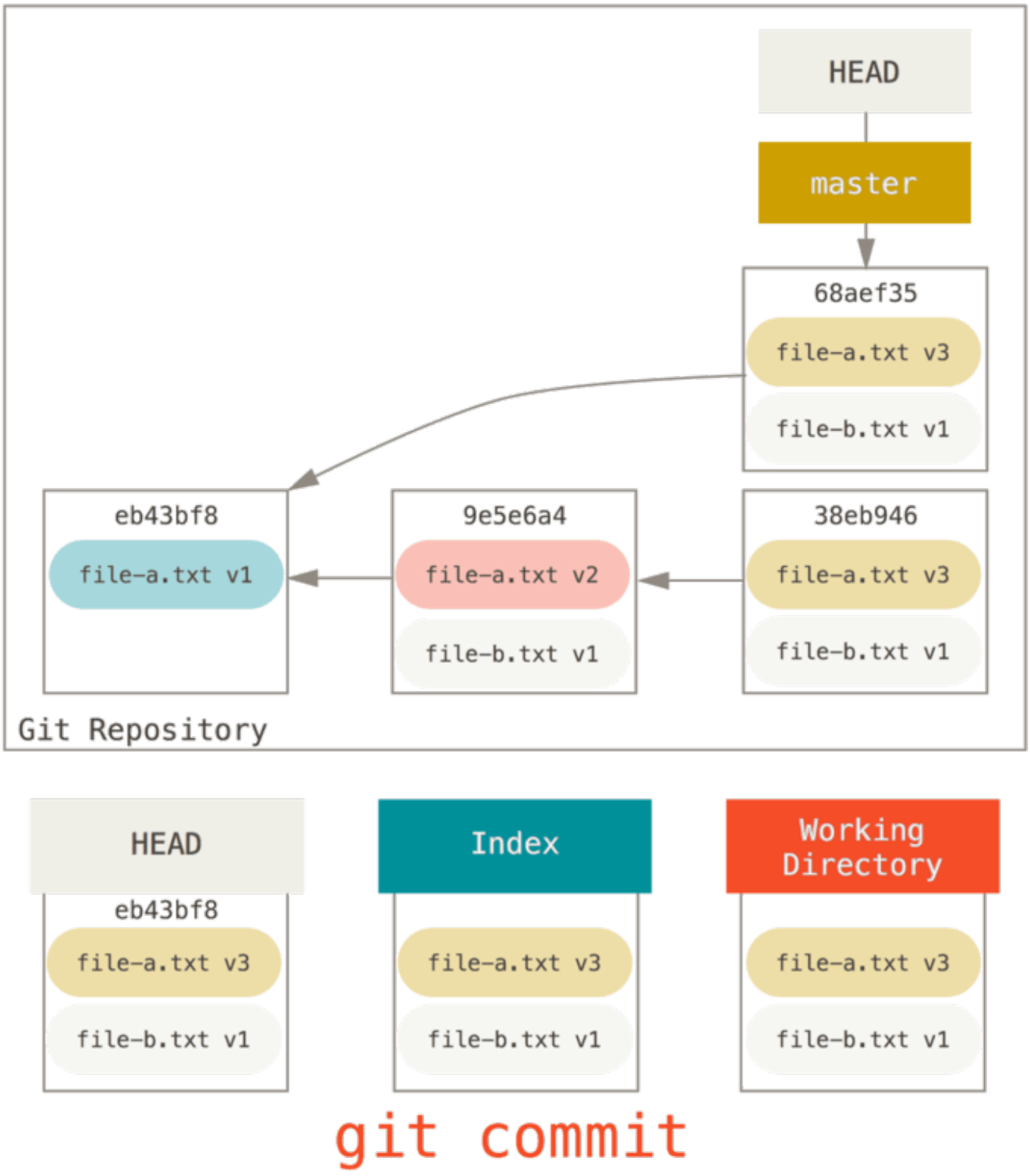
이런 프로젝트가 있다고 생각해보자. 첫 번째 커밋은 파일 하나를 추가했고, 두 번째 커밋은 기존 파일을 수정하고 새로운 파일도 추가했다. 세 번째 커밋은 첫 번째 파일을 다시 수정했다. 두 번째 커밋은 아직 작업 중인 커밋으로 이 커밋을 세 번째 커밋과 합치고 싶은 상황이다.



`git reset --soft HEAD~2` 명령을 실행하여 HEAD 포인터를 이전 커밋으로 되돌릴 수 있다(히스토리에서 그대로 유지할 처음 커밋 말이다).



이 상황에서 `git commit` 명령을 실행한다.



이제 사람들에게 공개할만한 히스토리가 만들어졌다. `file-a.txt` 파일이 있는 v1 커밋이 하나 그대로 있고, 두 번째 커밋에는 v3버전의 `file-a.txt` 파일과 새로 추가된 `file-b.txt` 파일이 있다. v2 버전은 더는 히스토리에 없다.