
This book is available in English.

Full translation available in

български език,
Deutsch,
Español,
Français,
Ελληνικά,
日本語,
한국어,
Nederlands,
Русский,
Slovenščina,
Tagalog,
Українська
简体中文,

Partial translations available in

Čeština,
Македонски,
Polski,
Српски,
Ўзбекча,
繁體中文,

Translations started for

azərbaycan dili,

Беларуская,
فارسی,
Indonesian,
Italiano,
Bahasa Melayu,
Português (Brasil),
Português (Portugal),
Svenska,
Türkçe.

The source of this book is hosted on GitHub.
Patches, suggestions and comments are welcome.

Chapters ▾

1. **1. 시작하기**
    1. 1.1 버전 관리란?
    2. 1.2 짧게 보는 Git의 역사
    3. 1.3 Git 기초
    4. 1.4 CLI
    5. 1.5 Git 설치
    6. 1.6 Git 최초 설정
    7. 1.7 도움말 보기
    8. 1.8 요약
2. **2. Git의 기초**
    1. 2.1 Git 저장소 만들기
    2. 2.2 수정하고 저장소에 저장하기
    3. 2.3 커밋 히스토리 조회하기
    4. 2.4 되돌리기
    5. 2.5 리모트 저장소
    6. 2.6 태그
    7. 2.7 Git Alias
    8. 2.8 요약
3. **3. Git 브랜치**
    1. 3.1 브랜치란 무엇인가
    2. 3.2 브랜치와 Merge 의 기초
    3. 3.3 브랜치 관리
    4. 3.4 브랜치 워크플로
    5. 3.5 리모트 브랜치
    6. 3.6 Rebase 하기
    7. 3.7 요약
4. **4. Git 서버**
    1. 4.1 프로토콜
    2. 4.2 서버에 Git 설치하기
    3. 4.3 SSH 공개키 만들기

4. 4.4 서버 설정하기
5. 4.5 Git 데몬
6. 4.6 스마트 HTTP
7. 4.7 GitWeb
8. 4.8 GitLab
9. 4.9 또 다른 선택지, 호스팅
10. 4.10 요약

5. **5. 분산 환경에서의 Git**

1. 5.1 분산 환경에서의 워크플로
2. 5.2 프로젝트에 기여하기
3. 5.3 프로젝트 관리하기
4. 5.4 요약

1. **6. GitHub**

1. 6.1 계정 만들고 설정하기
2. 6.2 GitHub 프로젝트에 기여하기
3. 6.3 GitHub 프로젝트 관리하기
4. 6.4 Organization 관리하기
5. 6.5 GitHub 스크립팅
6. 6.6 요약

2. **7. Git 도구**

1. 7.1 리비전 조회하기
2. 7.2 대화형 명령
3. 7.3 Stashing과 Cleaning
4. 7.4 내 작업에 서명하기
5. 7.5 검색
6. 7.6 히스토리 단장하기
7. 7.7 Reset 명확히 알고 가기
8. 7.8 고급 Merge
9. 7.9 Rerere
10. 7.10 Git으로 버그 찾기
11. 7.11 서브모듈
12. 7.12 Bundle
13. 7.13 Replace
14. 7.14 Credential 저장소
15. 7.15 요약

3. **8. Git맞춤**

1. 8.1 Git 설정하기
2. 8.2 Git Attributes
3. 8.3 Git Hooks
4. 8.4 정책 구현하기
5. 8.5 요약

4. **9. Git과 여타 버전 관리 시스템**

1. 9.1 Git: 범용 Client

2. 9.2 Git으로 옮기기
3. 9.3 요약

5. **10. Git의 내부**

1. 10.1 Plumbing 명령과 Porcelain 명령
2. 10.2 Git 개체
3. 10.3 Git Refs
4. 10.4 Packfile
5. 10.5 Refspec
6. 10.6 데이터 전송 프로토콜
7. 10.7 운영 및 데이터 복구
8. 10.8 환경변수
9. 10.9 요약

1. **A1. Appendix A: 다양한 환경에서 Git 사용하기**

1. A1.1 GUI
2. A1.2 Visual Studio
3. A1.3 Eclipse
4. A1.4 Bash
5. A1.5 Zsh
6. A1.6 Git in Powershell
7. A1.7 요약

2. **A2. Appendix B: 애플리케이션에 Git 넣기**

1. A2.1 Git 명령어
2. A2.2 Libgit2
3. A2.3 JGit
4. A2.4 go-git

3. **A3. Appendix C: Git 명령어**

1. A3.1 설치와 설정
2. A3.2 프로젝트 가져오기와 생성하기
3. A3.3 스냅샷 다루기
4. A3.4 Branch와 Merge
5. A3.5 공유하고 업데이트하기
6. A3.6 보기와 비교
7. A3.7 Debugging
8. A3.8 Patch 하기
9. A3.9 Email
10. A3.10 다른 버전 관리 시스템
11. A3.11 관리
12. A3.12 Plumbing 명령어

2nd Edition

3.1 Git 브랜치 - 브랜치란 무엇인가

모든 버전 관리 시스템은 브랜치를 지원한다. 개발을 하다 보면 코드를 여러 개로 복사해야 하는 일이 자주 생긴다. 코드를 통째로 복사하고 나서 원래 코드와는 상관없이 독립적으로 개발을 진행할 수 있는데, 이렇게 독립

적으로 개발하는 것이 브랜치다.

사람들은 브랜치 모델이 Git의 최고의 장점이라고, Git이 다른 것들과 구분되는 특징이라고 말한다. 당최 어떤 점이 그렇게 특별한 것일까. Git의 브랜치는 매우 가볍다. 순식간에 브랜치를 새로 만들고 브랜치 사이를 이동할 수 있다. 다른 버전 관리 시스템과는 달리 Git은 브랜치를 만들어 작업하고 나중에 Merge 하는 방법을 권장한다. 심지어 하루에 수십 번씩해도 괜찮다. Git 브랜치에 능숙해지면 개발 방식이 완전히 바뀌고 다른 도구를 사용할 수 없게 된다.

브랜치란 무엇인가

Git이 브랜치를 다루는 과정을 이해하려면 우선 Git이 데이터를 어떻게 저장하는지 알아야 한다.

Git은 데이터를 Change Set이나 변경사항(Diff)으로 기록하지 않고 일련의 스냅샷으로 기록한다는 것을 시작하기 에서 보여줬다.

커밋하면 Git은 현 Staging Area에 있는 데이터의 스냅샷에 대한 포인터, 저자나 커밋 메시지 같은 메타데이터, 이전 커밋에 대한 포인터 등을 포함하는 커밋 개체(커밋 Object)를 저장한다. 이전 커밋 포인터가 있어서 현재 커밋이 무엇을 기준으로 바뀌었는지를 알 수 있다. 최초 커밋을 제외한 나머지 커밋은 이전 커밋 포인터가 적어도 하나씩 있고 브랜치를 합친 Merge 커밋 같은 경우에는 이전 커밋 포인터가 여러 개 있다.

파일이 3개 있는 디렉토리가 하나 있고 이 파일을 Staging Area에 저장하고 커밋하는 예제를 살펴 보자. 파일을 Stage 하면 Git 저장소에 파일을 저장하고(Git은 이것을 Blob이라고 부른다) Staging Area에 해당 파일의 체크섬을 저장한다(시작하기 에서 살펴본 SHA-1을 사용한다).

```
$ git add README test.rb LICENSE
$ git commit -m 'The initial commit of my project'
```

`git commit` 으로 커밋하면 먼저 루트 디렉토리와 각 하위 디렉토리의 트리 개체를 체크섬과 함께 저장소에 저장한다. 그다음에 커밋 개체를 만들고 메타데이터와 루트 디렉토리 트리 개체를 가리키는 포인터 정보를 커밋 개체에 넣어 저장한다. 그래서 필요하면 언제든지 스냅샷을 다시 만들 수 있다.

이 작업을 마치고 나면 Git 저장소에는 다섯 개의 데이터 개체가 생긴다. 각 파일에 대한 Blob 세 개, 파일과 디렉토리 구조가 들어 있는 트리 개체 하나, 메타데이터와 루트 트리를 가리키는 포인터가 담긴 커밋 개체 하나이다.

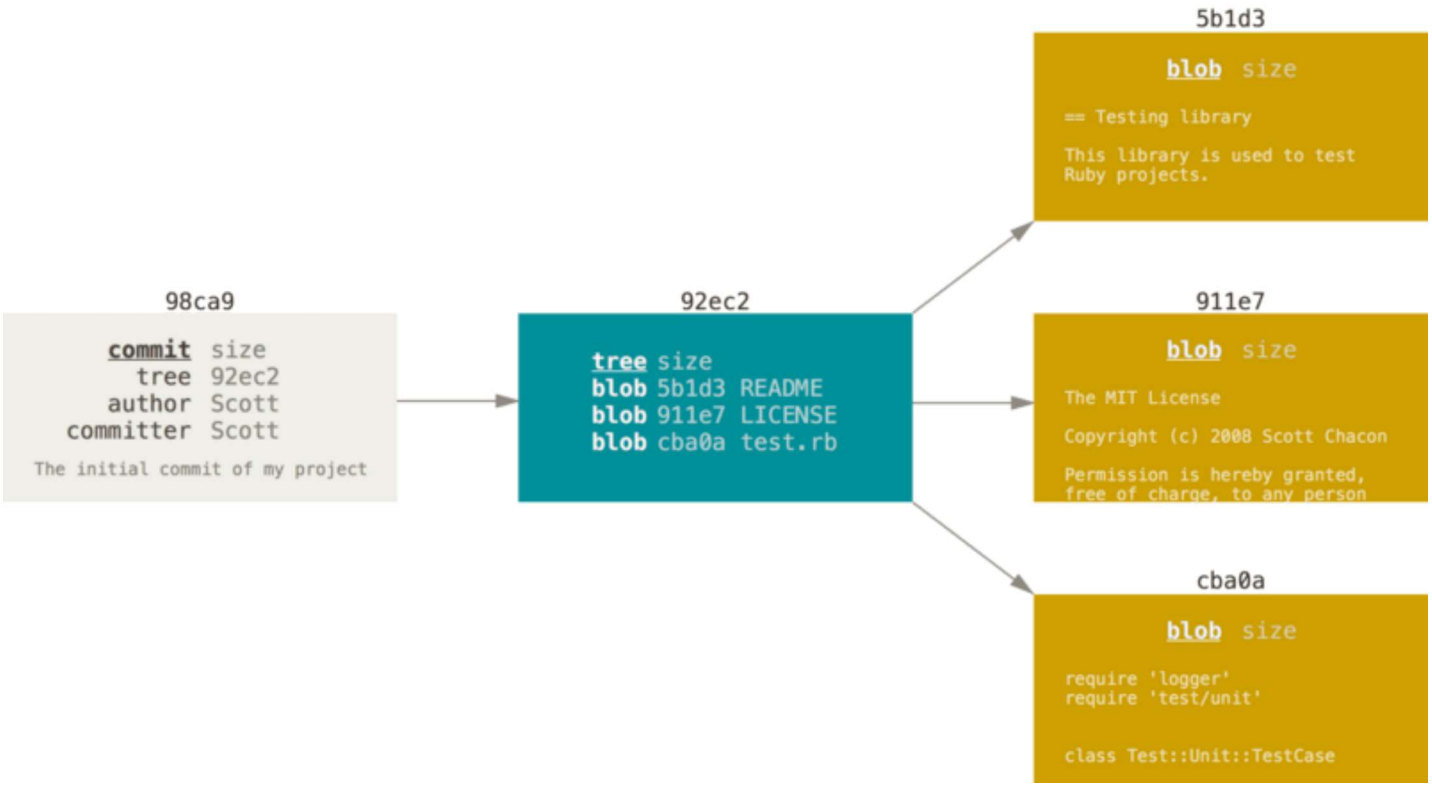

Figure 9. 커밋과 트리 데이터

다시 파일을 수정하고 커밋하면 이전 커밋이 무엇인지도 저장한다.

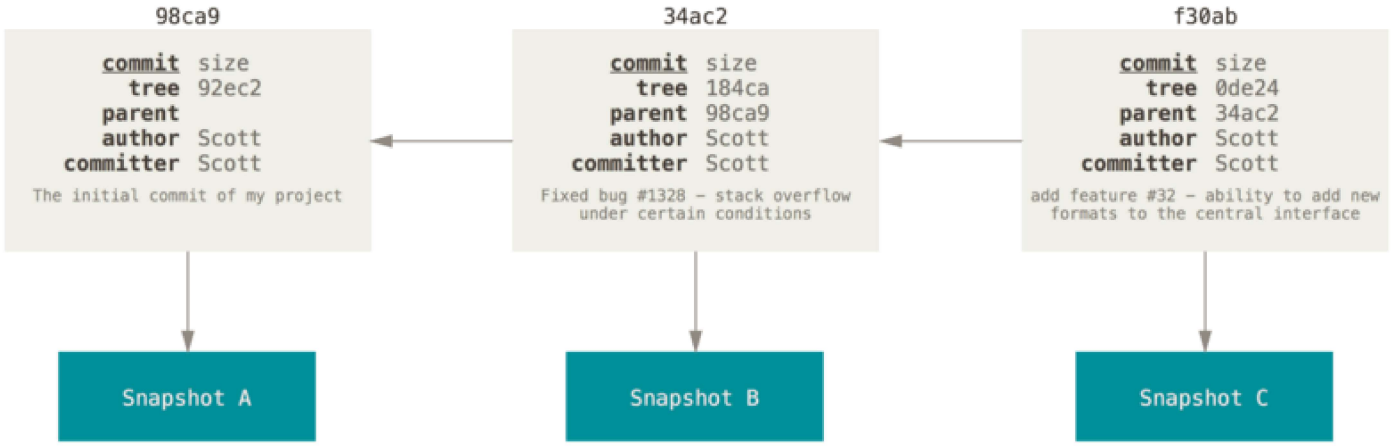

Figure 10. 커밋과 이전 커밋

Git의 브랜치는 커밋 사이를 가볍게 이동할 수 있는 어떤 포인터 같은 것이다. 기본적으로 Git은 `master` 브랜치를 만든다. 처음 커밋하면 이 `master` 브랜치가 생성된 커밋을 가리킨다. 이후 커밋을 만들면 `master` 브랜치는 자동으로 가장 마지막 커밋을 가리킨다.

Note Git 버전 관리 시스템에서 "master" 브랜치는 특별하지 않다. 다른 브랜치와 다른 것이 없다. 다만 모든 저장소에서 "master" 브랜치가 존재하는 이유는 `git init` 명령으로 초기화할 때 자동으로 만들어진 이 브랜치를 애써 다른 이름으로 변경하지 않기 때문이다.

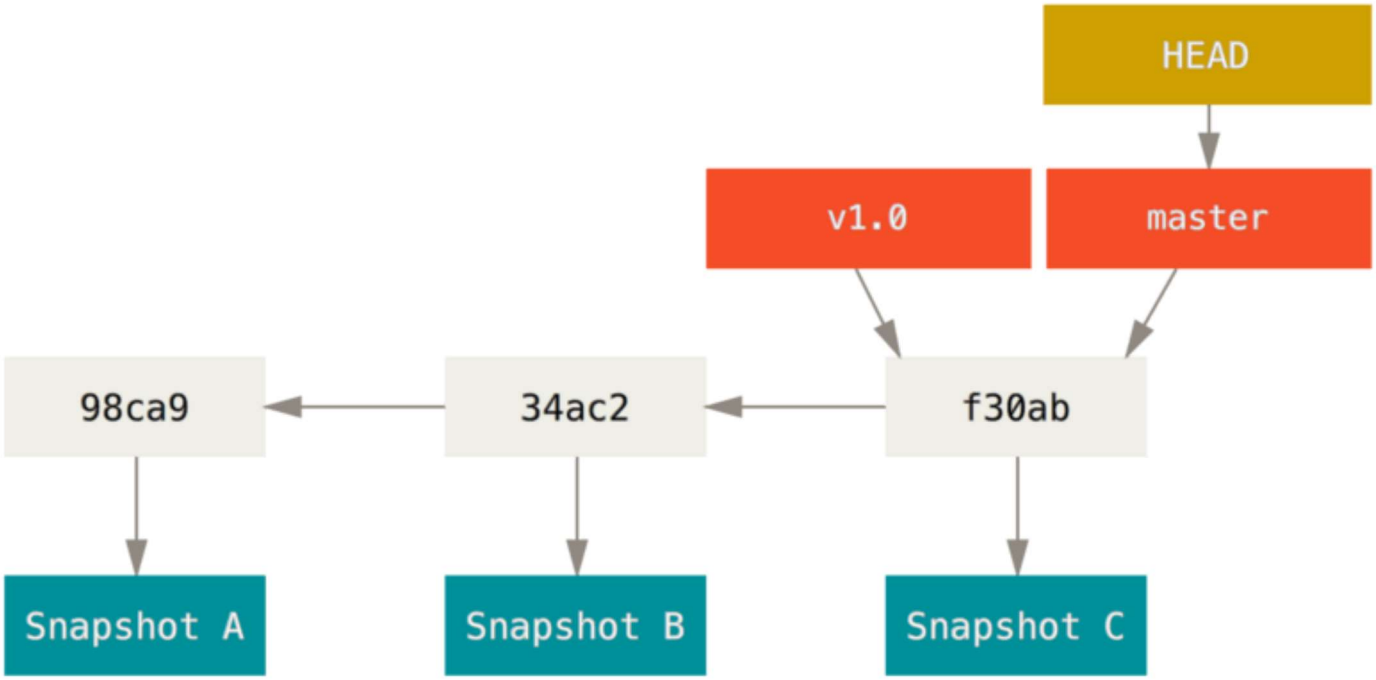

Figure 11. 브랜치와 커밋 히스토리

새 브랜치 생성하기

브랜치를 하나 새로 만들면 어떨까. 브랜치를 하나 만들어서 놀자. 아래와 같이 `git branch` 명령으로 `testing` 브랜치를 만든다.

```
$ git branch testing
```

새로 만든 브랜치도 지금 작업하고 있던 마지막 커밋을 가리킨다.

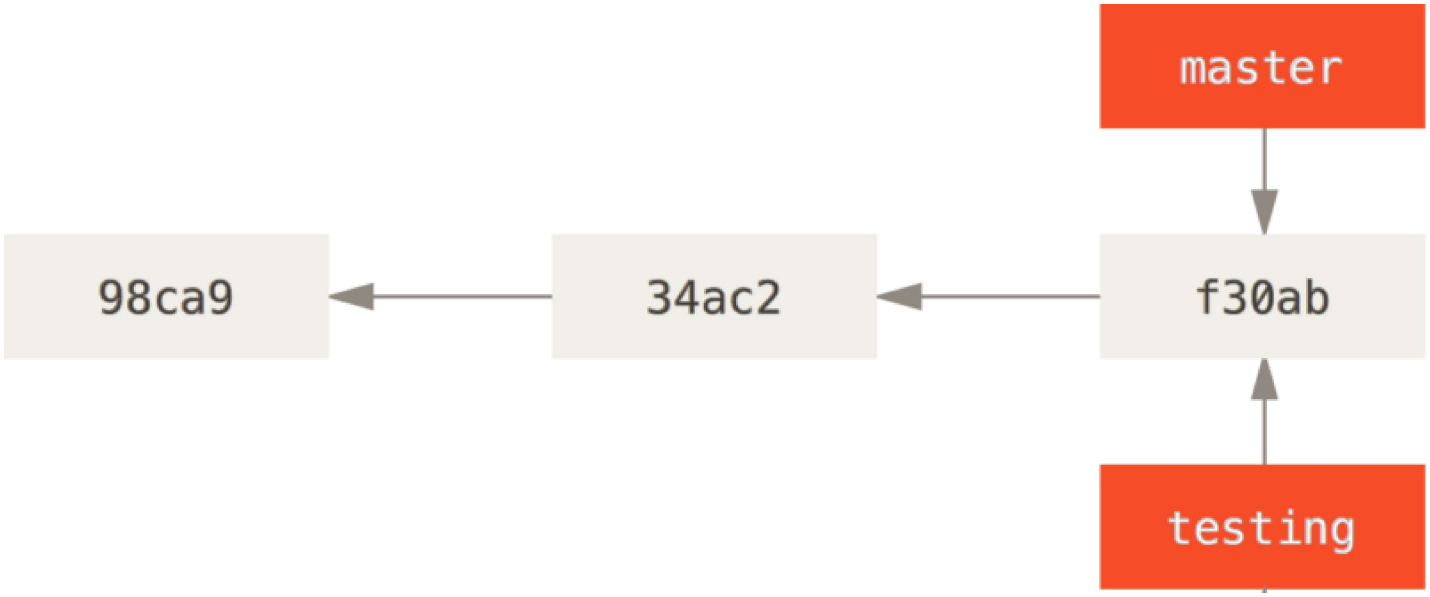

Figure 12. 한 커밋 히스토리를 가리키는 두 브랜치

지금 작업 중인 브랜치가 무엇인지 Git은 어떻게 파악할까. 다른 버전 관리 시스템과는 달리 Git은 'HEAD'라는 특수한 포인터가 있다. 이 포인터는 지금 작업하는 로컬 브랜치를 가리킨다. 브랜치를 새로 만들었지만, Git은 아직 `master` 브랜치를 가리키고 있다. `git branch` 명령은 브랜치를 만들기만 하고 브랜치를 옮기지 않는다.

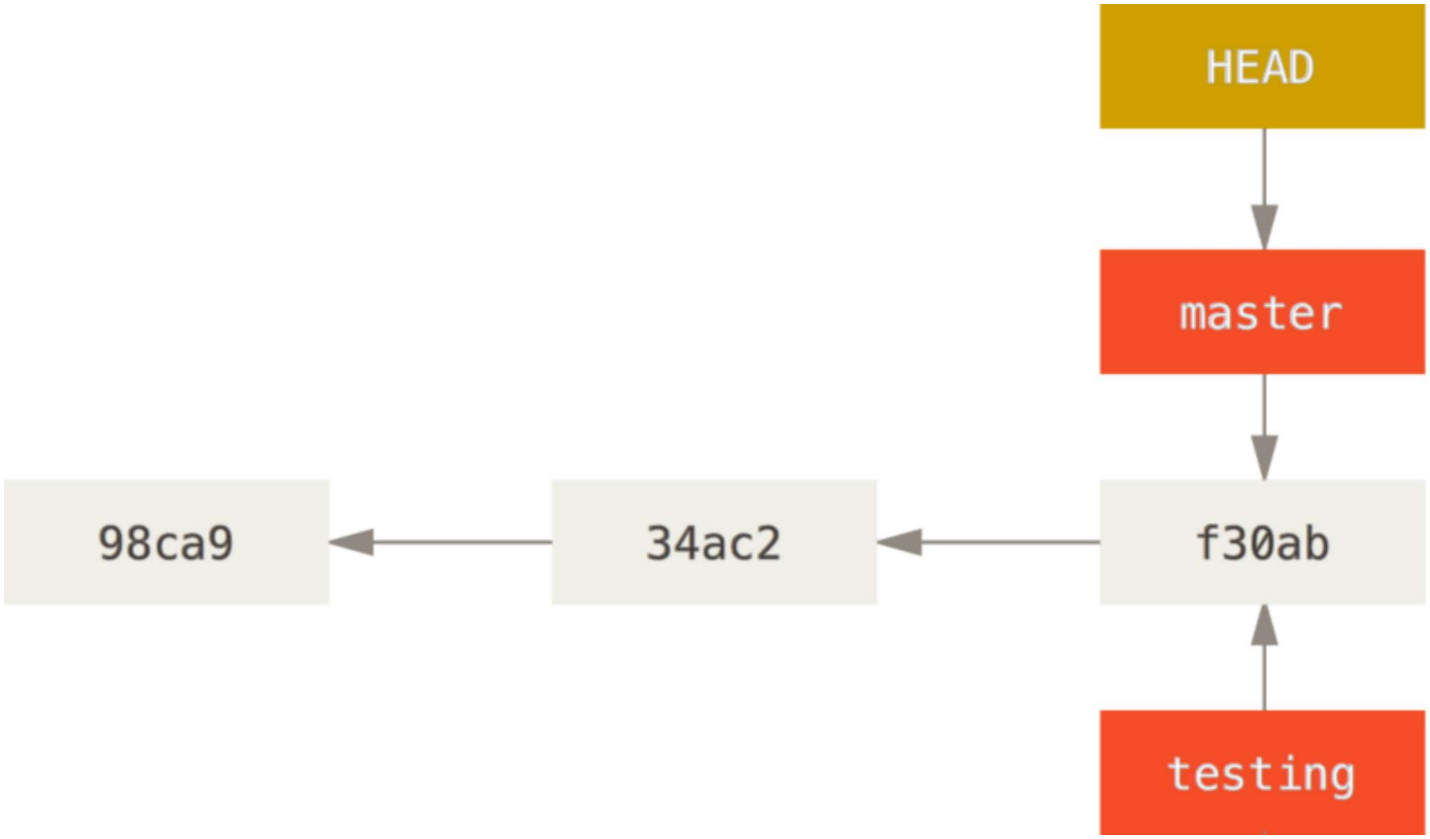

Figure 13. 현재 작업 중인 브랜치를 가리키는 HEAD

`git log` 명령에 `--decorate` 옵션을 사용하면 쉽게 브랜치가 어떤 커밋을 가리키는지도 확인할 수 있다.

```
$ git log --oneline --decorate
f30ab (HEAD -> master, testing) add feature #32 - ability to add new formats to the central interface
34ac2 Fixed bug #1328 - stack overflow under certain conditions
98ca9 The initial commit of my project
```

"master" 와 "testing" 이라는 브랜치가 `f30ab` 커밋 옆에 위치하여 이런식으로 브랜치가 가리키는 커밋을 확인할 수 있다.

브랜치 이동하기

`git checkout` 명령으로 다른 브랜치로 이동할 수 있다. 한번 `testing` 브랜치로 바꿔보자.

```
$ git checkout testing
```

이렇게 하면 HEAD는 testing 브랜치를 가리킨다.

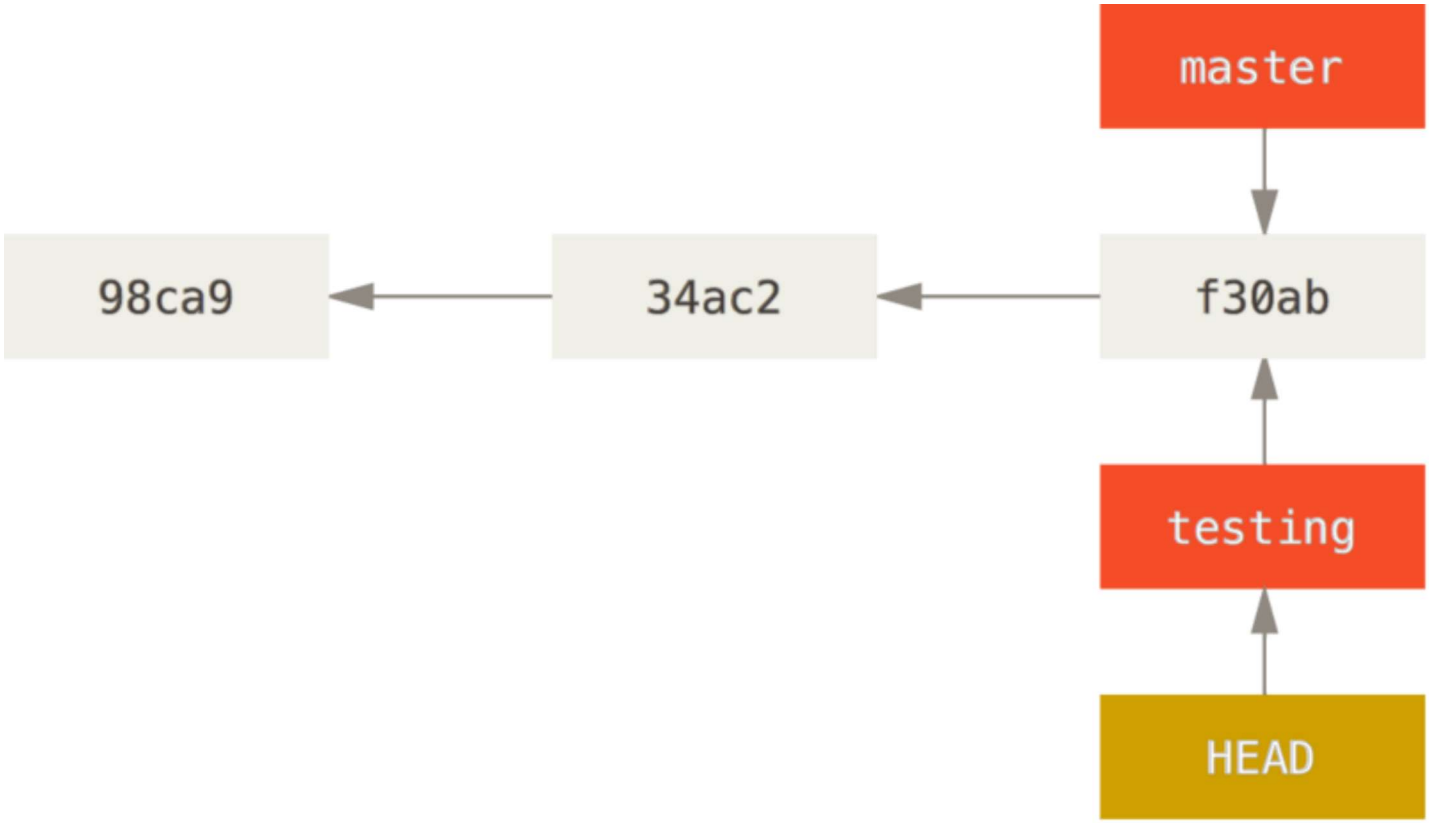

Figure 14. HEAD는 testing 브랜치를 가리킴

자, 이제 핵심이 보일 거다! 커밋을 새로 한 번 해보자.

```
$ vim test.rb
$ git commit -a -m 'made a change'
```

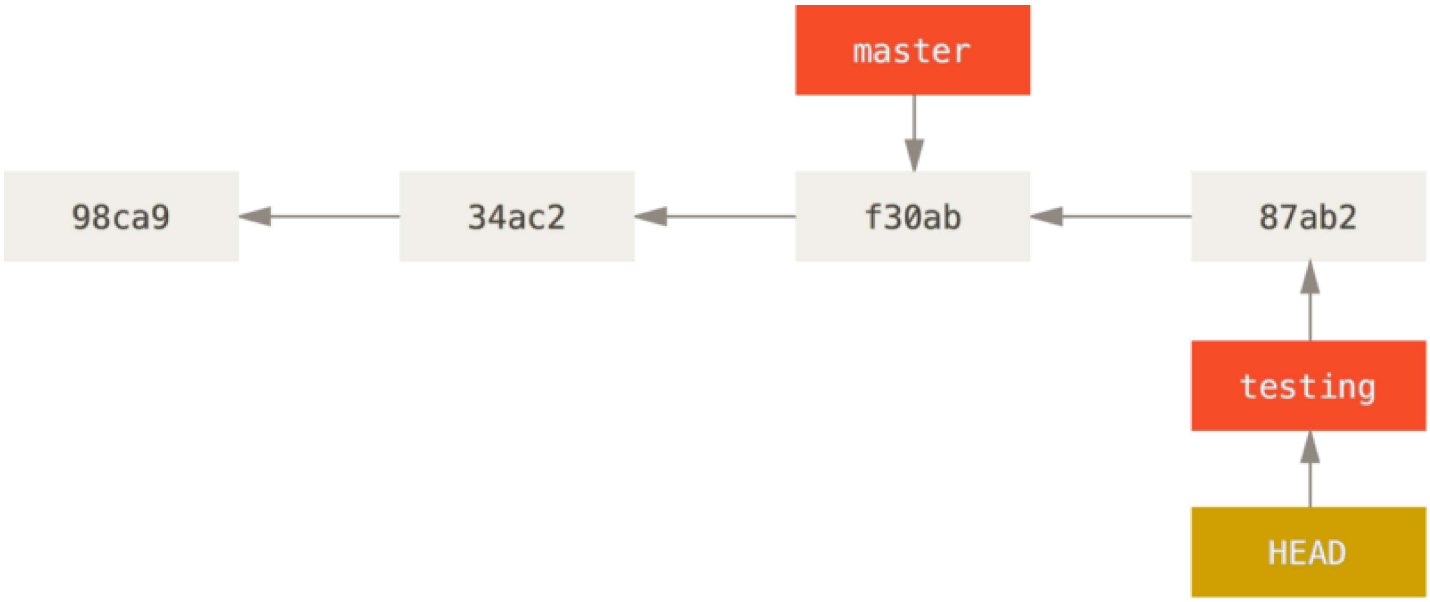

Figure 15. HEAD가 가리키는 testing 브랜치가 새 커밋을 가리킴

이 부분이 흥미롭다. 새로 커밋해서 `testing` 브랜치는 앞으로 이동했다. 하지만, `master` 브랜치는 여전히 이전 커밋을 가리킨다. `master` 브랜치로 되돌아가보자.

```
$ git checkout master
```

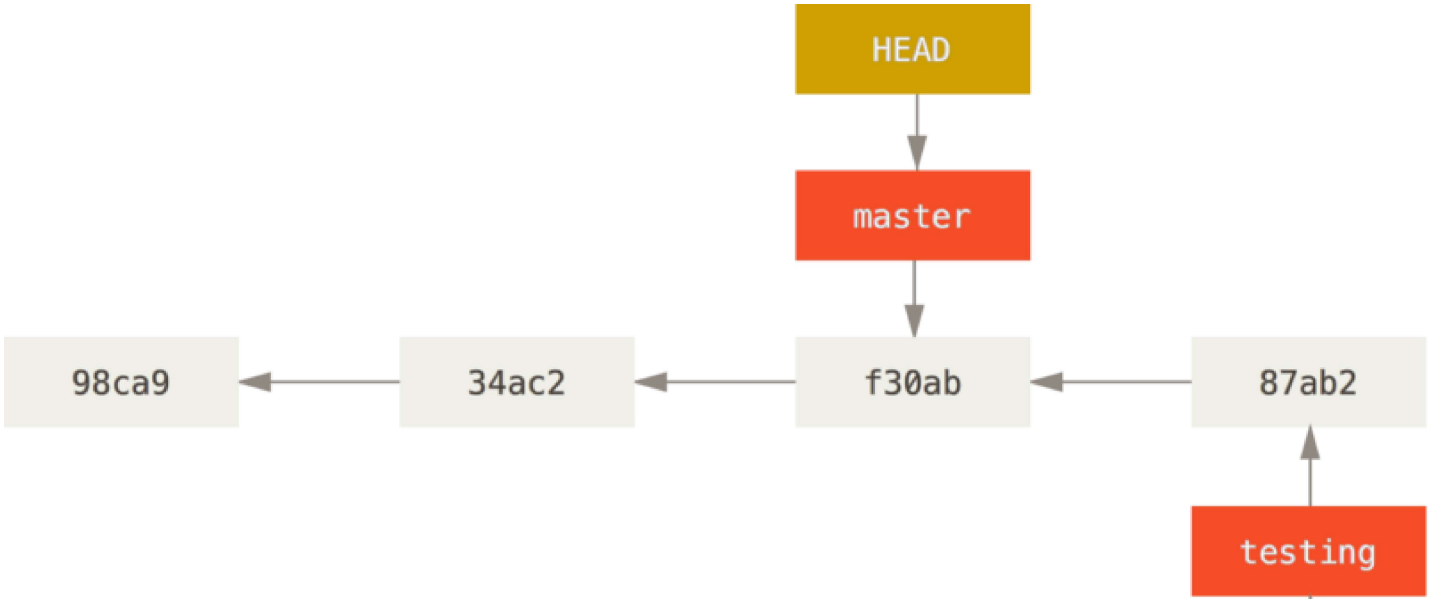

Figure 16. HEAD가 Checkout 한 브랜치로 이동함

방금 실행한 명령이 한 일은 두 가지다. `master` 브랜치가 가리키는 커밋을 HEAD가 가리키게 하고 워킹 디렉토리의 파일도 그 시점으로 되돌려 놓았다. 앞으로 커밋을 하면 다른 브랜치의 작업들과 별개로 진행되기 때문에 `testing` 브랜치에서 임시로 작업하고 원래 `master` 브랜치로 돌아와서 하던 일을 계속할 수 있다.

브랜치를 이동하면 워킹 디렉토리의 파일이 변경된다

Note 브랜치를 이동하면 워킹 디렉토리의 파일이 변경된다는 점을 기억해두어야 한다. 이전에 작업했던 브랜치로 이동하면 워킹 디렉토리의 파일은 그 브랜치에서 가장 마지막으로 했던 작업 내용으로 변경된다. 파일 변경시 문제가 있어 브랜치를 이동시키는게 불가능한 경우 Git은 브랜치 이동 명령을 수행하지 않는다.

파일을 수정하고 다시 커밋을 해보자.

```
$ vim test.rb
$ git commit -a -m 'made other changes'
```

프로젝트 히스토리는 분리돼 진행한다(갈라지는 브랜치). 우리는 브랜치를 하나 만들어 그 브랜치에서 일을 좀 하고, 다시 원래 브랜치로 되돌아와서 다른 일을 했다. 두 작업 내용은 서로 독립적으로 각 브랜치에 존재한다. 커밋 사이를 자유롭게 이동하다가 때가 되면 두 브랜치를 Merge 한다. 간단히 `branch`, `checkout`, `commit` 명령을 써서 말이다.

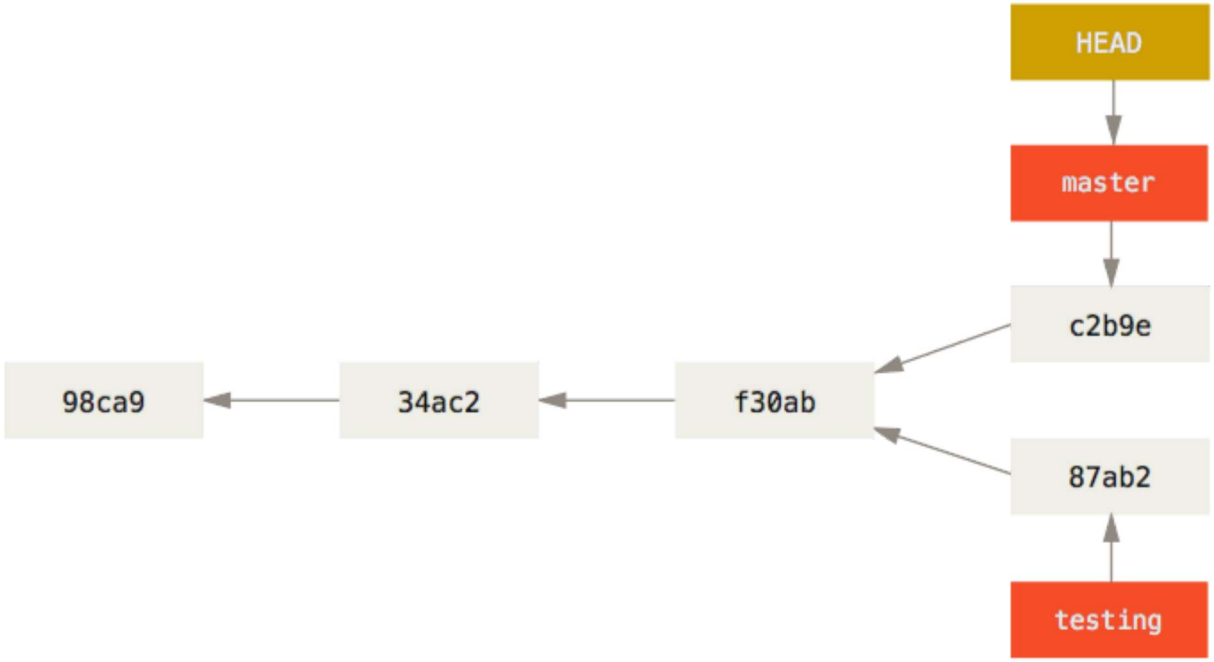

Figure 17. 갈라지는 브랜치

`git log` 명령으로 쉽게 확인할 수 있다. 현재 브랜치가 가리키고 있는 히스토리가 무엇이고 어떻게 갈라져 나왔는지 보여준다. `git log --oneline --decorate --graph --all` 이라고 실행하면 히스토리를 출력한다.

```
$ git log --oneline --decorate --graph --all
* c2b9e (HEAD, master) made other changes
| * 87ab2 (testing) made a change
|/
* f30ab add feature #32 - ability to add new formats to the
* 34ac2 fixed bug #1328 - stack overflow under certain conditions
* 98ca9 initial commit of my project
```

실제로 Git의 브랜치는 어떤 한 커밋을 가리키는 40글자의 SHA-1 체크섬 파일에 불과하기 때문에 만들기도 쉽고 지우기도 쉽다. 새로 브랜치를 하나 만드는 것은 41바이트 크기의 파일을(40자와 줄 바꿈 문자) 하나 만드는 것에 불과하다.

브랜치가 필요할 때 프로젝트를 통째로 복사해야 하는 다른 버전 관리 도구와 Git의 차이는 극명하다. 통째로 복사하는 작업은 프로젝트 크기에 따라 다르겠지만 수십 초에서 수십 분까지 걸린다. 그에 비해 Git은 순식간이다. 게다가 커밋을 할 때마다 이전 커밋의 정보를 저장하기 때문에 Merge 할 때 어디서부터(Merge Base) 합쳐야 하는지 안다. 이런 특징은 개발자들이 수시로 브랜치를 만들어 사용하게 한다.

이제 왜 그렇게 브랜치를 수시로 만들고 사용해야 하는지 알아보자.

prev | next
About this site
Patches, suggestions, and comments are welcome.
Git is a member of Software Freedom Conservancy