Stratio, Inc.

Git guide book

Wonhyuk Yang

Version 1.1, 2020-10-05

Table of Contents

Chapter 1. 시작하기

이 장에서 설명하는 것은 Git을 처음 접하는 사람에게 필요한 내용이다. 이 장을 다 읽고 나면 Git에 대한 기초와 Git을 설정하고 사용하는 방법을 터득하게 될 것이다.

1.1. Git 기초

Git의 핵심은 뭘까? 이 질문은 Git을 이해하는데 굉장히 중요하다. Git이 무엇이고 어떻게 동작하는지 이해한다면 쉽게 Git을 효과적으로 사용할 수 있다.

1.1.1. 스냅샷

Git은 프로젝트를 파일 시스템 스냅샷의 연속으로 취급한다. 쉽게 생각하면 프로젝트의 각각의 순간을 저장한다고 생각해도 된다. 스냅샷은 파일에 대한 링크로 구성되고 파일의 내용은 다른 곳에 저장된다. 변화가 없는 파일에 대한 스냅샷은 이전에 파일을 가르키면 된다.

프로젝트는 아래 그림처럼 일련의 **스냅샷의 스트림**처럼 구성되며 Git은 이 스트림을 관리한다.

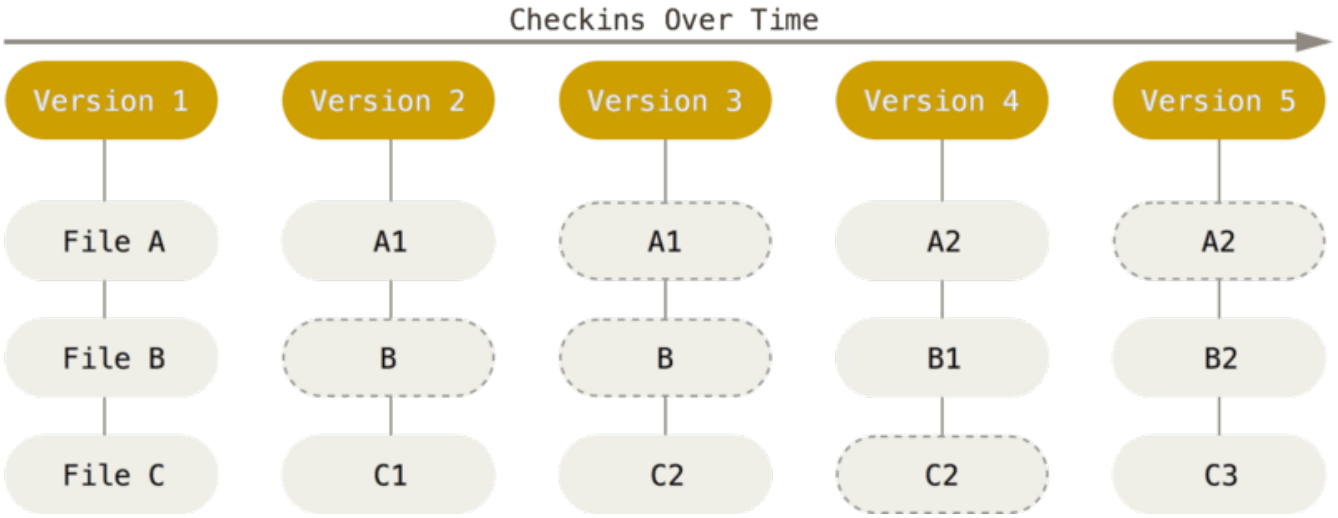


Figure 1. 시간순으로 프로젝트의 스냅샷을 저장.

1.1.2. Git은 데이터를 추가할 뿐

Git으로 무얼 하든 Git 데이터베이스에 데이터가 *추가* 된다. 데이터 베이스에 추가된 것은 되돌리거나 데이터를 삭제하기 어렵다. 하지만 커밋(데이터 베이스에 스냅샷 추가)하지 않으면 변경사항을 잃어버릴 수 있다.

따라서 Git을 사용하면 프로젝트가 심각하게 망가질 걱정 없이 매우 즐겁게 여러 가지 실험을 해 볼 수 있다. 되돌리기을 보면 Git에서 데이터를 복구하는지 알 수 있다.

1.1.3. 세 가지 상태

이 부분은 중요하기에 집중해서 읽어야 한다. Git을 공부하기 위해 반드시 짚고 넘어가야 할 부분이다. Git은 **Git 디렉토리, 워킹 트리, Staging Area** 이렇게 3가지 단계를 이용한다.

1. **Git 디렉토리**는 저장소 관리에 필요한 데이터 베이스로 다음과 같은 일을 수행한다.
    - 스냇샷들을 저장한다.
    - 스냅샷들이 가르키는 파일을 저장한다.
    - 스냅샷 간에 연결 관계를 저장한다.
2. **워킹 트리**는 실제 파일이 존재하는 곳이다. 바로 눈에 보이기 때문에 사용자는 워킹 트리 안의 내용을 편집할 수 있다.
3. **Staging Area**는 곧 커밋할 스냅샷에 대한 정보를 저장한다. Git에서는 기술용어로는 `Index` 라고 하지만, `Staging Area` 라는 용어를 써도 상관 없다.

Git으로 하는 일은 기본적으로 아래와 같다.

1. 워킹 트리에서 파일을 수정한다.
2. 워킹 트리의 수정된 파일을 Staging area에 추가/제거한다.
3. Staging Area에 있는 스냅샷을 Git 디렉토리에 영구적인 스냅샷으로 저장한다.

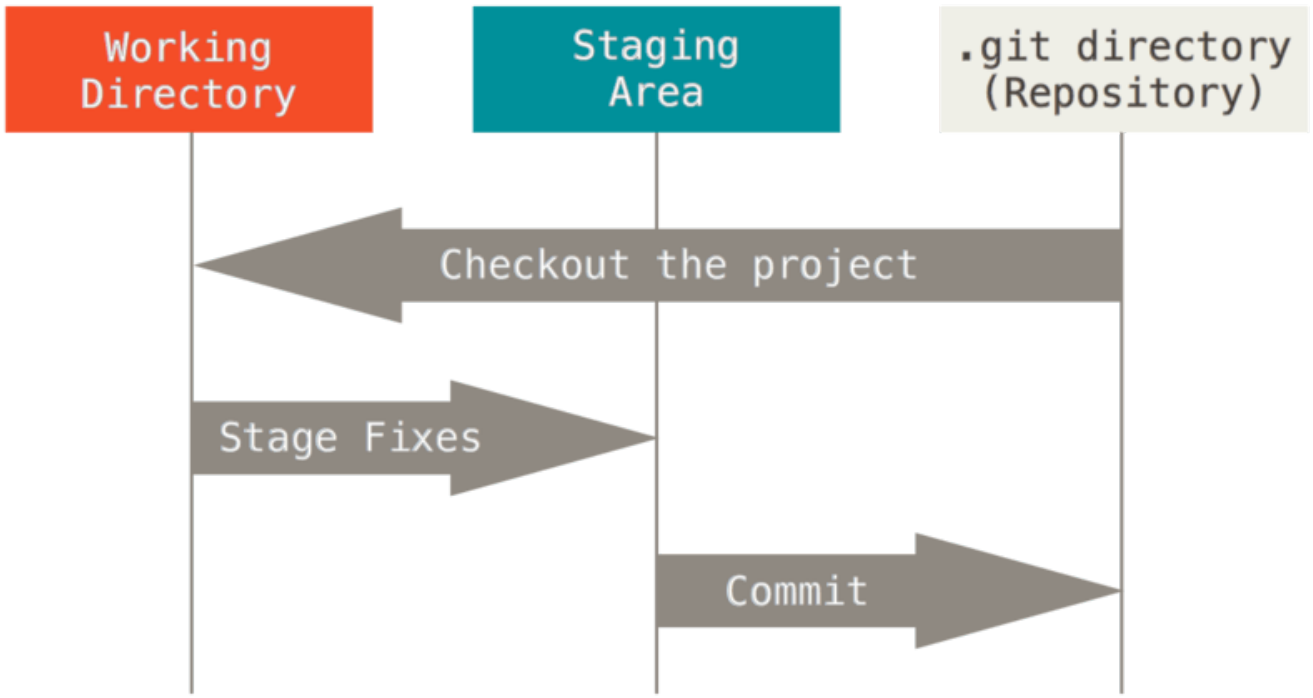


Figure 2. 워킹 트리, Staging Area, Git 디렉토리.

1.2. CLI

Git을 사용하는 방법은 많다. CLI로 사용할 수도 있고 GUI를 사용할 수도 있다. 이 책에서는 Git CLI 사용법을 설명한다. Git의 *모든* 기능을 지원하는 것은 CLI 뿐이다. GUI 프로그램의 대부분은 Git 기능 중 일부만 구현하기 때문에 비교적 단순하다. CLI를 사용할 줄 알면 GUI도 사용할 수 있지만 반대는 성립하지 않는다. GUI를 사용하고 싶더라도 CLI가 기본으로 설치되는 도구이기 때문에 CLI기준으로 설명하겠다.

그래서 Mac의 Terminal이나 Windows의 CMD나 Powershell을 실행시키는 방법은 알고 있을 거라고 가정한다. 만약 이 말이 무슨 말인지 모르겠다면 일단 여기서 멈추고 Terminal이나 Powershell에 대해 알아보기 바란다. 그래야 이 책의 설명을 따라올 수 있다.

1.3. Git 최초 설정

Git을 설치하고 나면 Git의 사용 환경을 적절하게 설정해 주어야 한다. 환경 설정은 한 컴퓨터에서 한 번만 하면 된다. 설정한 내용은 Git을 업그레이드해도 유지된다. 언제든지 다시 바꿀 수 있는 명령어도 있다.

1.3.1. 사용자 정보

Git을 설치하고 나서 가장 먼저 해야 하는 것은 사용자이름과 이메일 주소를 설정하는 것이다. Git은 커밋할 때마다 이 정보를 사용한다. 커밋에 해당 정보가 사용된다.

```
$ git config --global user.name "John Doe"
$ git config --global user.email johndoe@example.com
```

다시 말하자면 `--global` 옵션으로 설정하는 것은 딱 한 번만 하면 된다. 해당 시스템에서 해당 사용자가 사용할 때는 이 정보를 사용한다. 만약 프로젝트마다 다른 이름과 이메일 주소를 사용하고 싶으면 `--global` 옵션을 빼고 명령을

실행한다.

1.3.2. 편집기

사용자 정보를 설정하고 나면 Git에서 사용할 텍스트 편집기를 고른다. 기본적으로 Git은 시스템의 기본 편집기를 사용한다.

하지만, vim 같은 다른 텍스트 편집기를 사용할 수 있고 아래와 같이 실행하면 된다.

```
$ git config --global core.editor vim
```

Vim과 Emacs, Notepad++은 꽤 인기 있는 편집기로 개발자들이 즐겨 사용한다. Mac이나 Linux 같은 Unix 시스템, Windows 시스템에서 사용 가능하다. 여기서 소개하는 편집기들이 불편해서 다른 편집기를 사용하고자 한다면 해당 편집기를 Git 편집기로 설정하는 방법을 찾아봐야 한다.

자신의 편집기를 설정하지 않으면 갑자기 실행된 편집기에 당황할 수 있다. 그땐 당황하지 말고 편집기를 그냥 종료하면 Git 명령을 취소할 수 있다.

1.4. 요약

우리는 간략하게 Git이 무엇인지 배웠다. 다음 장부터는 본격적으로 Git의 사용법을 배운다.

Chapter 2. Git의 기초

Git을 사용하는 방법을 알고 싶은데 한 챕터밖에 읽을 시간이 없다면 이번 챕터를 읽어야 한다. Git에서 자주 사용하는 명령어는 모두 2장에 등장한다.

2장을 다 읽으면 저장소를 만들고 설정하는 방법, 파일을 추적하거나(Track) 추적을 그만두는 방법, 변경 내용을 Stage 하고 커밋하는 방법을 알게 된다. 특정 파일을 무시하도록 Git을 설정하는 방법, 실수를 쉽고 빠르게 만회하는 방법, 프로젝트 히스토리를 조회하고 커밋을 비교하는 하는 방법을 살펴본다.

2.1. Git 저장소 만들기

주로 다음 주 가지 중 한 가지 방법으로 Git 저장소를 쓰기 시작한다.

1. 아직 버전관리를 하지 않는 로컬 디렉토리 하나를 선택해서 Git 저장소를 적용하는 방법
2. 다른 어딘가에서 Git 저장소를 *Clone* 하는 방법

어떤 방법을 사용하든 로컬 디렉토리에 Git 저장소가 준비되면 이제 뭔가 해볼 수 있다.

2.1.1. 기존 디렉토리를 Git 저장소로 만들기

버전관리를 하지 않은 기존 프로젝트를 Git으로 관리하고 싶은 경우 우선 프로젝트의 디렉토리로 이동한다.

```
$ cd /your/project/directory/my_project
```

그리고 아래와 같은 명령을 실행한다:

```
$ git init
```

이 명령은 `.git` 이라는 하위 디렉토리를 만든다. `.git` 디렉토리에는 저장소 관리에 필요한 **Git 디렉토리**와 **Staging Area**가 저장되어 있다.

2.1.2. 기존 저장소를 Clone 하기

다른 프로젝트에 참여하려거나(Contribute) Git 저장소를 복사하고 싶을 때 `git clone` 명령을 사용한다. `git clone` 을 실행하면 프로젝트 히스토리를 전부 받아온다. 실제로 서버의 디스크가 망가져도 클라이언트 저장소 중에서 아무거나 하나 가져다가 복구하면 된다(서버에만 적용했던 설정은 복구하지 못하지만 모든 데이터는 복구된다.

`git clone <url>` 명령으로 저장소를 Clone 한다. `libgit2` 라이브러리 소스코드를 Clone 하려면 아래과 같이 실행한다.

```
$ git clone https://github.com/libgit2/libgit2
```

이 명령은 `libgit2` 라는 디렉토리를 만들고 그 안에 `.git` 디렉토리를 만든다. 그리고 저장소의 데이터를 모두 가져와서 자동으로 가장 최신 버전으로 워킹 트리를 구성해 놓는다.

2.2. 수정하고 저장소에 저장하기

만질 수 있는 Git 저장소를 하나 만들었다. 이제는 파일을 수정하고 파일의 스냅샷을 커밋해 보자. 파일을 수정하다가 저장하고 싶으면 스냅샷을 커밋한다.

2.2.1. 파일의 상태 확인하기

파일의 상태를 확인하려면 보통 `git status` 명령을 사용한다. init 한 후에 바로 이 명령을 실행하면 아래과 같은 메시지를 볼 수 있다.

```
$ git status
On branch master
No commits yet
nothing to commit (create/copy files and use "git add" to track)
```

- **Line 1**: 기본 브랜치가 master이기 때문에 현재 브랜치 이름이 `master`로 나온다.
- **Line 2**: 해당 저장소에 아무런 커밋이 없다는 것을 말해주다.
- **Line 3**: 커밋할 새로운 스냅샷이 없다는 것을 말해준다.

브랜치 관련 내용은 Git 브랜치 에서 자세히 다룬다.

텅 빈 저장소이기에 **워킹 트리, Staging Aread, Git 디렉토리**가 비어있다. 해당 저장소의 상태를 그림으로 표현하면 아래와 같다.

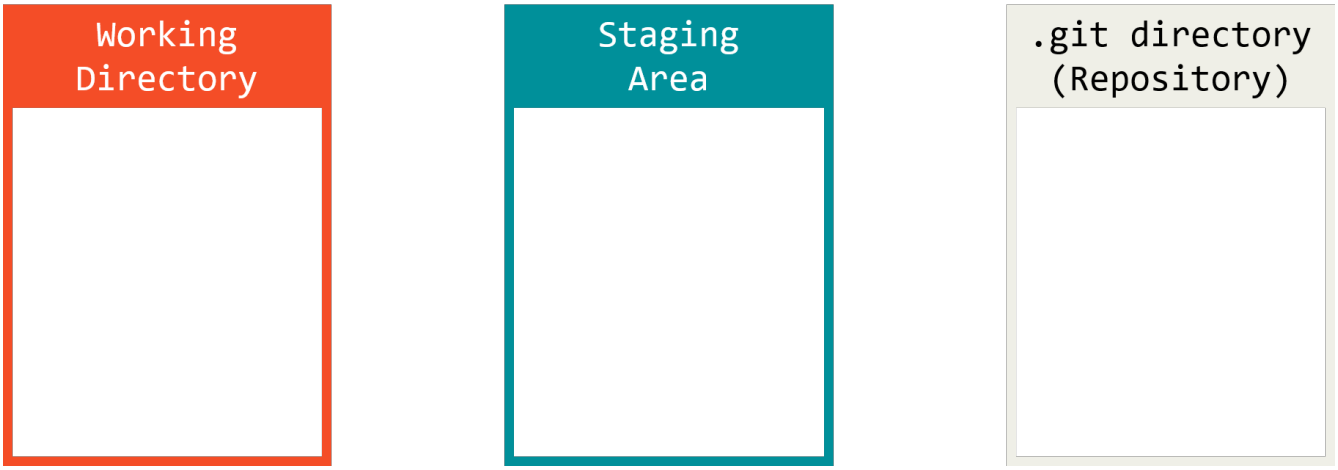


Figure 3. 텅 빈 저장소

이제 프로젝트에 `README` 파일을 만들어보자. `README` 파일은 새로 만든 파일이기 때문에 `git status` 를 실행하면 `Untracked files`에 들어 있다.

```
$ echo 'My Project' > README
$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

    README

nothing added to commit but untracked files present (use "git add" to track)
```

`README` 파일은 `Untracked files` 부분에 속해 있는데 이것은 `README` 파일이 Untracked 상태라는 것을 말한다. Git은 `Untracked files`을 **워킹 트리**에는 존재하고, **Staging Area**에 올라가지 않은 파일이라고 본다.

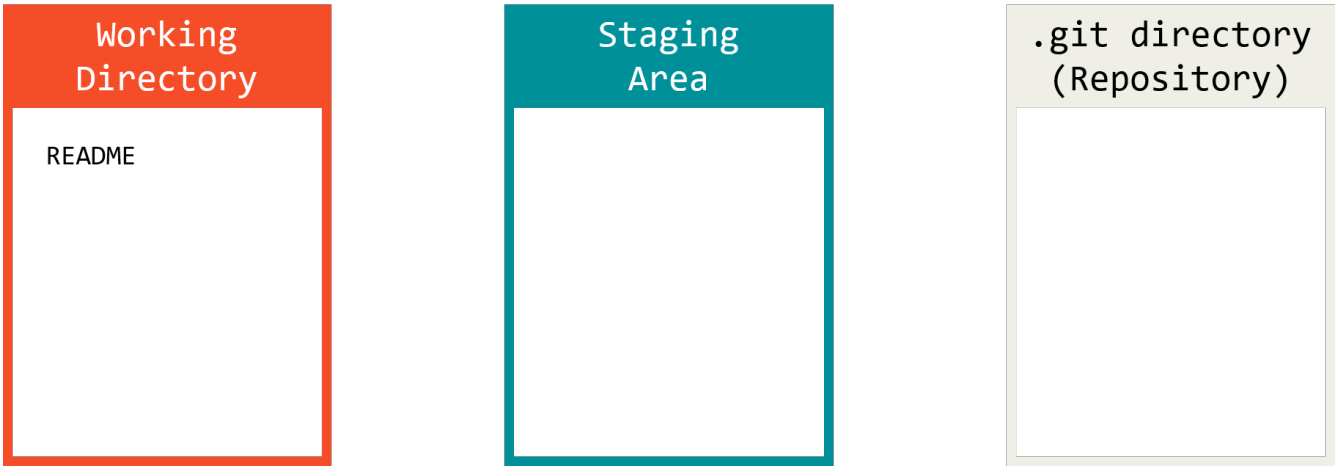


Figure 4. 새로운 파일이 추가된 저장소

파일이 **Staging Area**에 올라가지 전까지는 Git은 절대 그 파일을 커밋하지 않는다.

그래서 일하면서 생성하는 바이너리 파일 같은 것을 커밋하는 실수는 하지 않게 된다. `README` 파일을 **Staging Area**에 올려 보자.

2.2.2. 파일을 새로 추적하기

`git add` 명령으로 파일을 **Staging Area**의 스냅샷에 반영할 수 있다. 아래 명령을 실행하면 Git은 `README` 파일 **Staging Area**에 추가한다.

```
$ git add README
```

`git status` 명령을 다시 실행하면 `README` 파일이 Tracked 상태이면서 커밋에 추가될 `Staged 상태`라는 것을 확인할 수 있다.

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

    new file:   README
```

`Changes to be committed` 에 들어 있는 파일은 **Staged Area**에 추가된 상태라는 것을 의미한다. 커밋하면 **Staging Area**의 스냅샷이 저장소 히스토리에 남는다.

`git add` 명령은 파일 또는 디렉토리의 경로를 아규먼트로 받는다. 디렉토리면 아래에 있는 모든 파일들까지 재귀적으로 추가한다.

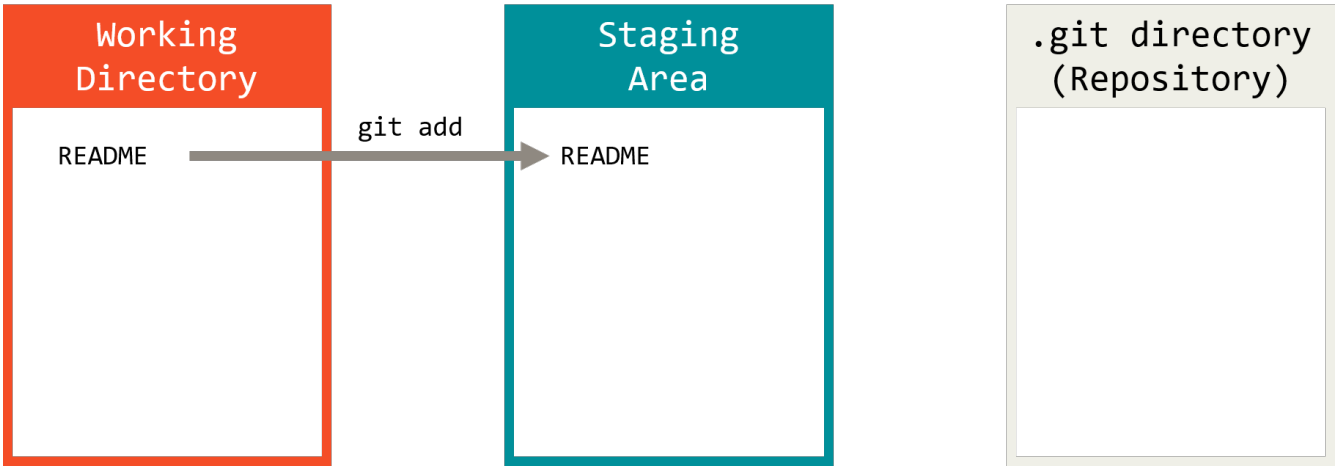


Figure 5. Staging Area가 업데이트된 저장소

2.2.3. 변경사항 커밋하기

수정한 것을 커밋하기 위해 **Staging Area**에 파일을 정리했다. Git은 **워킹 트리**에 생성하거나 수정하고 나서 `git add` 명령으로 **Staging Area**에 반영되지 않은 내용은 커밋하지 않는다.

커밋하기 전에 `git status` 명령으로 모든 것이 `Staged` 상태인지 확인할 수 있다. 그 후에 `git commit` 을 실행하여 커밋한다.

```
$ git commit
```

Git 설정에 지정된 편집기가 실행되고, 아래와 같은 텍스트가 자동으로 포함된다 (아래 예제는 Vim 편집기의 화면이다. 시작하기 에서 설명했듯이 `git config --global core.editor` 명령으로 어떤 편집기를 사용할지 설정할 수 있다).

```
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
#
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   new file:   README
#
```

커밋 메시지를 작성하고 저장하고 종료하면 Git은 입력된 내용(#로 시작하는 내용을 제외한)으로 새 커밋을 하나 완성한다.

```
$ git commit
[master (root-commit) b6005bd] Add README
 1 file changed, 1 insertion(+)
  create mode 100644 README
```

커밋이 성공하면 위와 같이 몇 가지 정보를 출력하는데 위 예제는(`master`)브랜치에 커밋했고 체크섬은 (`b6005bd`)이라고 알려준다. 그리고 수정한 파일은 몇 개이고 삭제됐거나 추가된 라인이 몇 라인인지 알려준다.

Staging Area의 스냅샷이 **Git 저장소**에 저장되었기 때문에 나중에 스냅샷끼리 비교하거나 예전 스냅샷으로 되돌릴 수 있다. 커밋이 완료된 저장소의 모습은 아래와 같다.

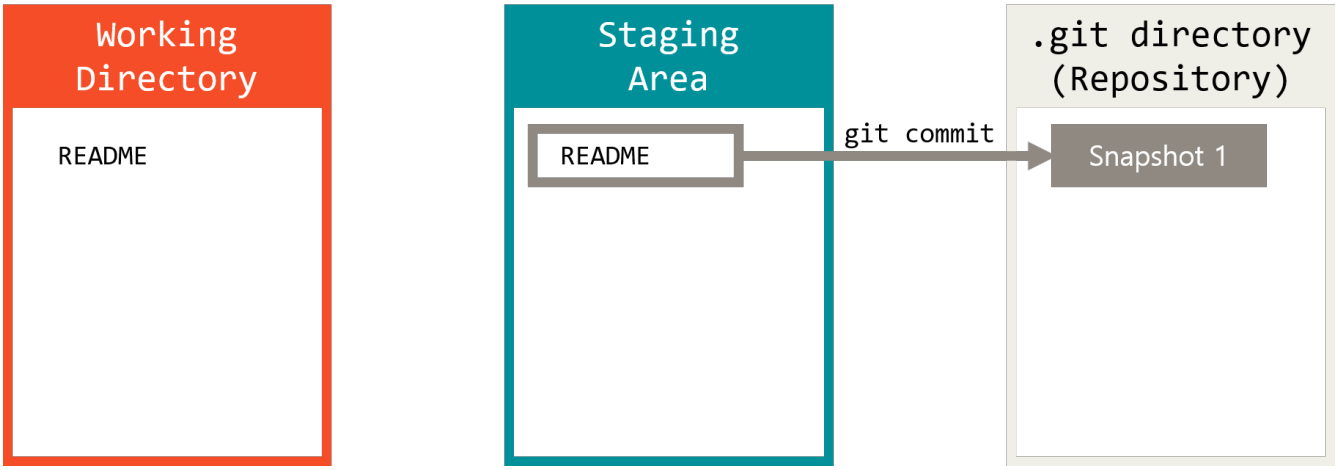


Figure 6. commit이 완료된 저장소

2.2.4. 커밋 가이드라인

다른 것보다 먼저 커밋 메시지에 대한 주의사항을 알아보자. 커밋 메시지를 잘 작성하는 가이드라인을 알아두면 다른 개발자와 함께 일하는 데 도움이 많이 된다.

무엇보다도 먼저 공백문자를 깨끗하게 정리하고 커밋해야 한다. Git은 공백문자를 검사해볼 수 있는 간단한 명령을 제공한다. 커밋을 하기 전에 `git diff --check` 명령으로 공백문자에 대한 오류를 확인할 수 있다.

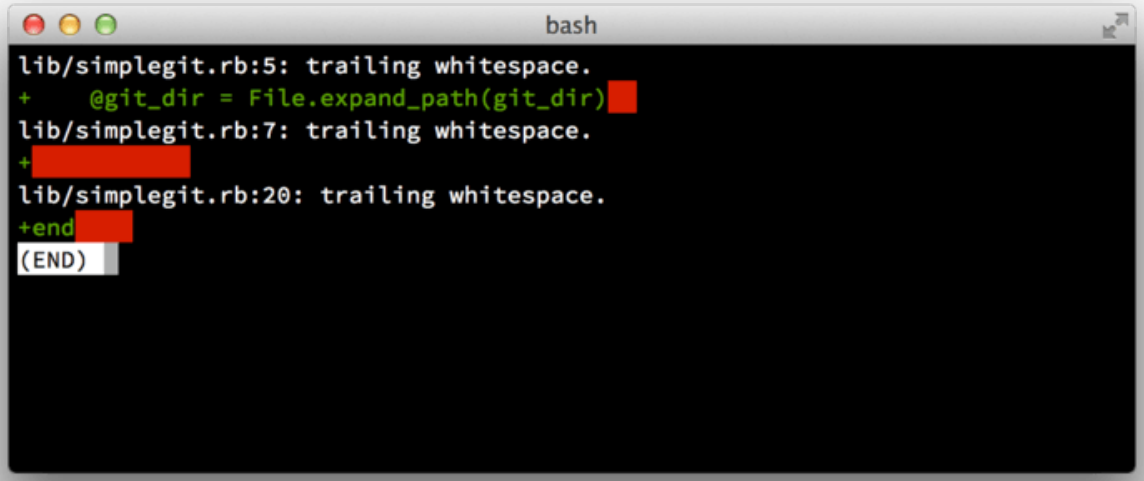


Figure 7. `git diff --check` 의 결과.

커밋을 하기 전에 공백문자에 대해 검사를 하면 공백으로 불필요하게 커밋되는 것을 막고 이런 커밋으로 인해 불필요하게 다른 개발자들이 신경 쓰는 일을 방지할 수 있다.

그리고 각 커밋은 논리적으로 구분되는 Changeset이다. 최대한 수정사항을 한 주제로 요약할 수 있어야 하고 여러 가지 이슈에 대한 수정사항을 하나의 커밋에 담지 않아야 한다. 여러 가지 이슈를 한꺼번에 수정했다고 하더라도 Staging Area를 이용하여 한 커밋에 이슈 하나만 담기도록 한다. 작업 내용을 분할하고, 각 커밋마다 적절한 메시지를 작성한다.

마지막으로 명심해야 할 점은 커밋 메시지 자체다. 좋은 커밋 메시지를 작성하는 습관은 Git을 사용하는 데 도움이 많이 된다. 아래 내용은 Tim Pope가 작성한 커밋 메시지의 템플릿이다.

```
영문 50글자 이하의 간략한 수정 요약

자세한 설명. 영문 72글자 이상이 되면
라인 바꿈을 하고 이어지는 내용을 작성한다.
특정 상황에서는 첫 번째 라인이 이메일
메시지의 제목이 되고 나머지는 메일
내용이 된다. 빈 라인은 본문과 요약을
구별해주기에 중요하다(본문 전체를 생략하지 않는 한).

이어지는 내용도 한 라인 띄우고 쓴다.

  - 목록 표시도 사용할 수 있다.

  - 보통 '-' 나 '*' 표시를 사용해서 목록을 표현하고
    표시 앞에 공백 하나, 각 목록 사이에는 빈 라인
    하나를 넣는데, 이건 상황에 따라 다르다.
```

메시지를 이렇게 작성하면 함께 일하는 사람은 물론이고 자신에게도 매우 유용하다.

2.2.5. Modified 상태의 파일을 Stage 하기

이미 `Tracked 상태`인 파일을 수정하는 법을 알아보자. `README` 파일을 수정하고 나서 `git status` 명령을 다시 실행하면 결과는 아래와 같다.

```
$ git status
On branch master
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:   README
```

이 `README` 파일은 `Changes not staged for commit` 에 있다. 이것은 해당 파일이 **Staging Area**에 있는 내용과 **워킹 트리**에 있는 내용이 다르다는 말이다. 수정된 파일의 저장소는 아래 모습과 같다.

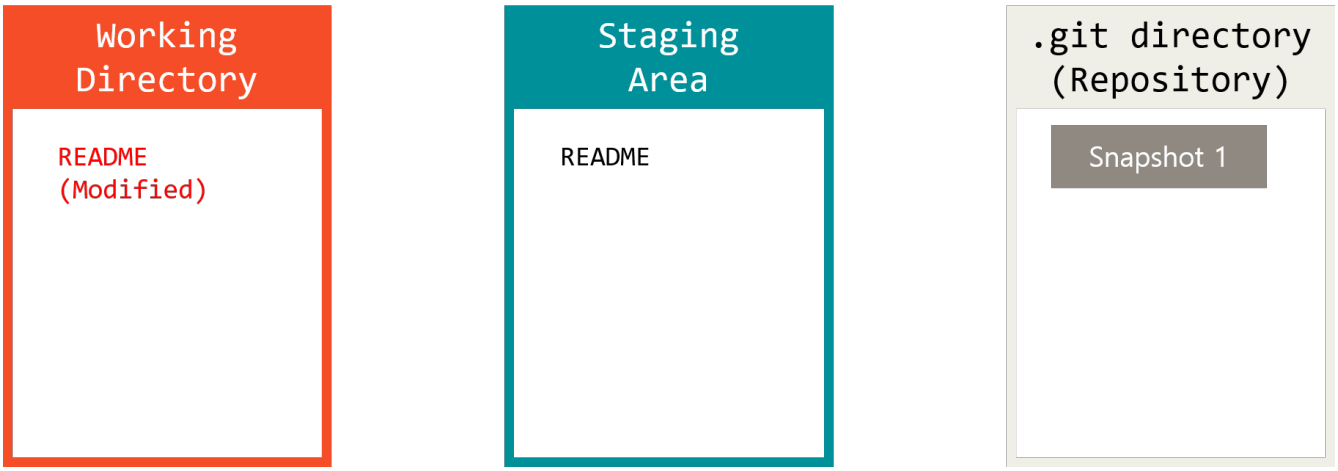


Figure 8. 수정된 파일이 있는 저장소

`Staged 상태`로 만들려면 `git add` 명령을 실행해야 한다. `git add` 명령은 단순히 새로운 파일을 추가할 뿐만 아니라, 수정된 내용을 **Staging Area**에 올릴때도 사용된다. 즉, add의 의미는 **Staging Area**에 파일의 수정된 사항을

반영한다고 받아들이는게 좋다. `git add` 명령을 실행하여 `README` 파일을 `Staged 상태`로 만들고 `git status` 명령으로 결과를 확인해보자.

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

      modified:   README
```

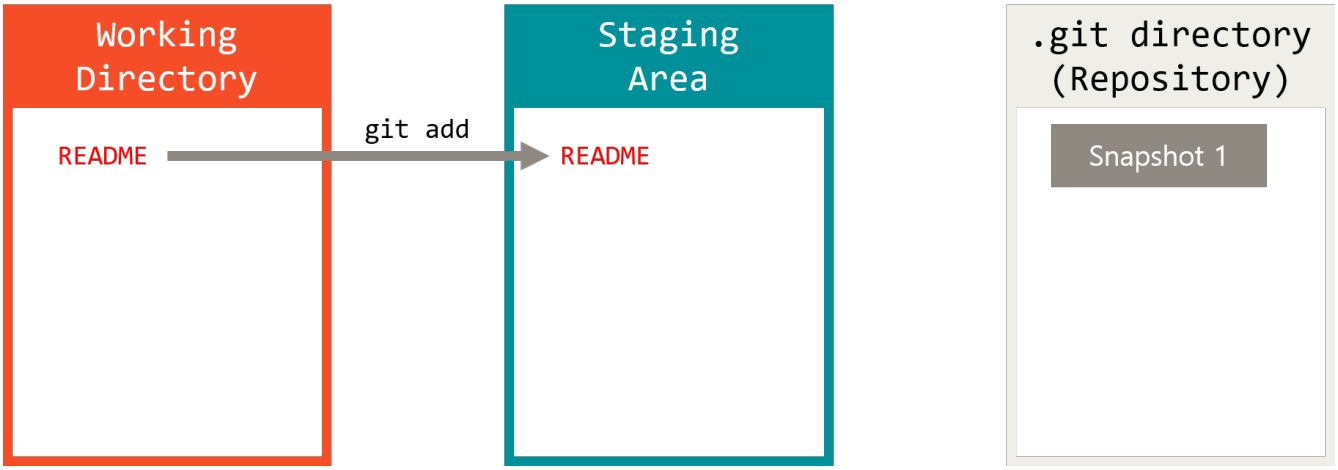


Figure 9. README를 add한 저장소

`README` 파일은 `Staged 상태`이므로 다음 커밋에 포함된다. 하지만 `README` 파일을 더 수정해야 한다는 것을 알게 되었다. 이 상황에서 `README` 파일을 열고 수정한다. 이제 커밋할 준비가 다 됐다고 생각할 테지만, Git은 그렇지 않다. `git status` 명령으로 파일의 상태를 다시 확인해보자.

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

      modified:   README

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:   README
```

`README` 가 `Staged 상태`이면서 *동시에* `Unstaged 상태`로 나온다. 어떻게 이런 일이 가능할까? 현재 저장소의 모습은 아래의 모습과 같다.

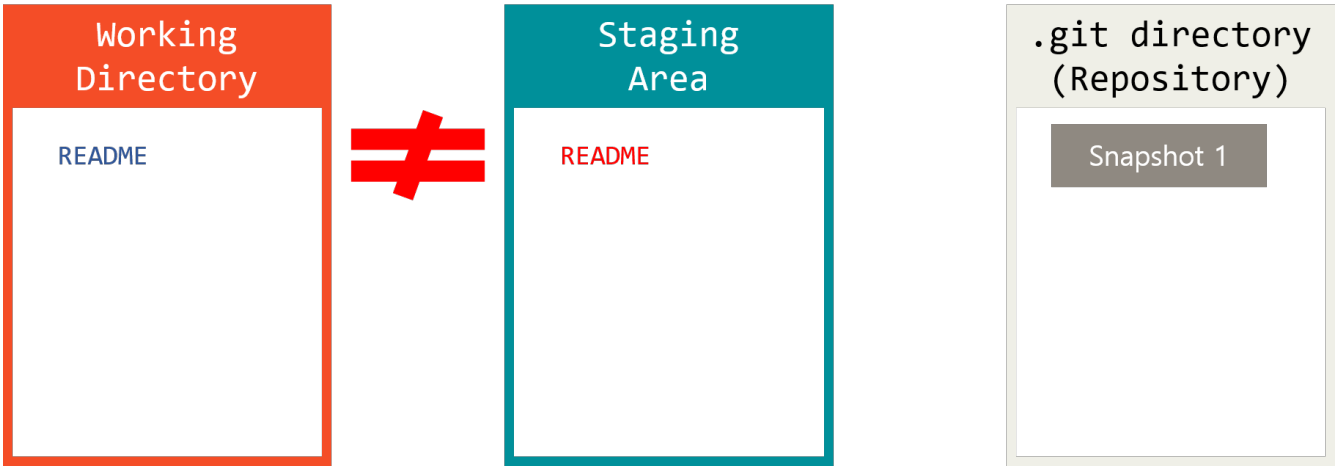


Figure 10. README를 수정한 저장소

마지막으로 `git add`를 한 `README` 파일이 **Staging Area**에 있고, **워킹 트리**에는 그와 다른 수정된 파일이 있다. 각각의 내용이 다르기 때문에 `Unstaged 상태`라고 나온 것이다.

정리하자면, `git add` 명령을 실행하면 실행 시점의 **워킹 트리** 파일을 **Staging Area**에 올린다. 그 이후 **워킹 트리** 파일을 수정하면 *다시* `git add`를 사용하여 **Staging Area**에 올려야 한다.

```
$ git add CONTRIBUTING.md
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   README
    modified:   CONTRIBUTING.md
```

2.2.6. 파일 무시하기

어떤 파일은 Git이 관리할 필요가 없다. 보통 로그 파일이나 빌드 시스템이 자동으로 생성한 파일이 그렇다. 그런 파일을 무시하려면 `.gitignore` 파일을 만들고 그 안에 무시할 파일 패턴을 적는다. 아래는 `.gitignore` 파일의 예이다.

```
$ cat .gitignore
*.o
*.a
```

- **Line 1~2**: 확장자가 `.o` 나 `.a` 인 파일을 Git이 무시하라는 것이다

GitHub은 다양한 프로젝트에서 자주 사용하는 `.gitignore` 예제를 관리하고 있다. 어떤 내용을 넣을지 막막하다면 https://github.com/github/gitignore 사이트에서 적당한 예제를 찾을 수 있다.

2.2.7. Staged와 Unstaged 상태의 변경 내용을 보기

단순히 파일이 변경됐다는 사실이 아니라 어떤 내용이 변경됐는지 살펴보려면 `git status` 명령이 아니라 `git diff` 명령을 사용해야 한다. `git diff` 명령을 사용하는데 Patch처럼 어떤 라인을 추가했고 삭제했는지가 궁금할 때 사용한다.

```
$ git diff
diff --git a/README.md b/README.md
index 8ebb991..643e24f 100644
--- a/README.md
+++ b/README.md
@@ -65,7 +65,8 @@ branch directly, things can get messy.
 Please include a nice description of your changes when you submit your PR;
 if we have to read the whole diff to figure out why you're contributing
 in the first place, you're less likely to get feedback and have your change
-merged in.
+merged in. Also, split your changes into comprehensive chunks if your patch is
+longer than a dozen lines.

 If you are starting to work on a particular area, feel free to submit a PR
 that highlights your work in progress (and note in the PR title that it's
```

이 명령은 **워킹 트리**에 있는 것과 **Staging Area**에 있는 것을 비교한다. 그래서 수정하고 아직 `git add` 하지 않은 것을 보여준다.

만약 **Staging Area**와 **Git 저장소**의 최신 스냅샷의 변경 사항을 보고 싶으면 `git diff --staged` 옵션을 사용한다.

```
$ git diff --staged
diff --git a/README b/README
new file mode 100644
index 0000000..03902a1
--- /dev/null
+++ b/README
@@ -0,0 +1 @@
+My Project
```

꼭 잊지 말아야 할 것이 있는데 `git diff` 는 `Unstaged` 상태인 것들만 보여준다. 수정한 파일을 모두 **Staging Area**에 넣었다면 `git diff` 명령은 아무것도 출력하지 않는다.

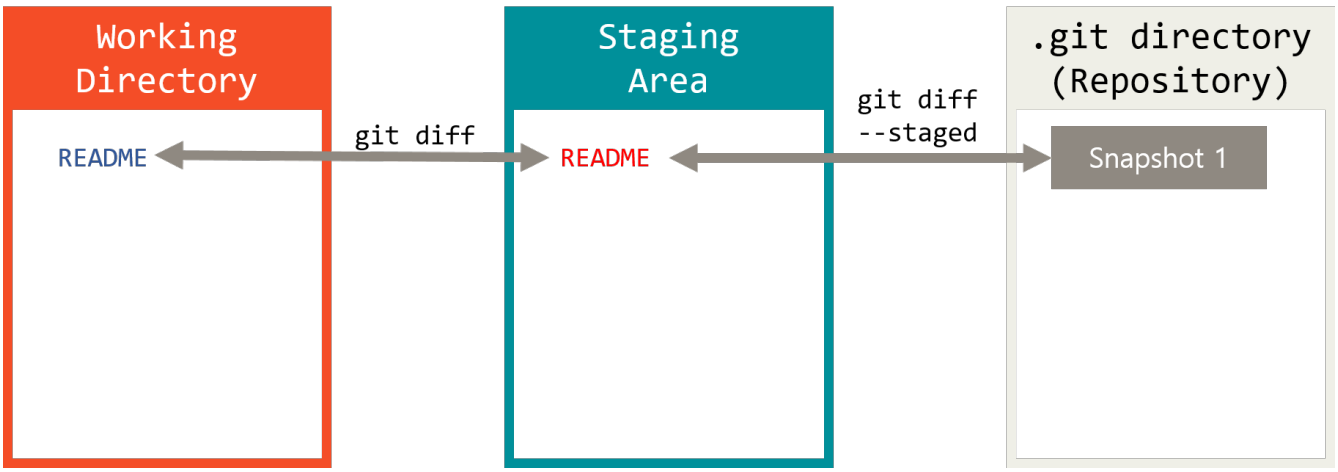


Figure 11. git diff

2.2.8. 파일 삭제하기

Git에서 파일을 제거하려면 `git rm` 명령으로 **Staging Area**에서와 **워킹 트리**에서 삭제할 수 있다.

Git 명령을 사용하지 않고 단순히 **워킹 트리**에서 파일을 삭제하고 `git status` 명령으로 상태를 확인하면 Git은 현재 `Changes not staged for commit` (즉, *Unstaged* 상태)라고 표시해준다.

```
$ rm PROJECTS.md
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        deleted:    PROJECTS.md

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

워킹 트리와 **Staging Area**가 서로 다르므로 `git add` 또는 `git rm`로 **Staging Area**를 업데이트해줘야 커밋에 반영된다. 커밋되지 않은 **Staging Area**또는 **워킹 트리**의 내용은 복구가 불가능하니 주의해야한다.

2.2.9. 파일 이름 변경하기

Git은 아래와 같이 파일 이름을 변경할 수 있다.

```
$ git mv file_from file_to
```

이 명령을 실행하고 Git의 상태를 확인해보면 Git은 이름이 바뀐 사실을 알고 있다.

```
$ git mv README.md README
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    renamed:    README.md -> README
```

사실 `git mv` 명령은 아래 명령어를 수행한 것과 동일한 것이다.

```
$ mv README.md README
$ git rm README.md
$ git add README
```

2.3. 커밋 히스토리 조회하기

새로 저장소를 만들어서 몇 번 커밋을 했을 수도 있고, 커밋 히스토리가 있는 저장소를 Clone 했을 수도 있다. 어쨌든 가끔 저장소의 히스토리를 보고 싶을 때가 있다. Git에는 히스토리를 조회하는 명령어인 `git log` 가 있다.

이 예제에서는 simplegit 이라는 매우 단순한 프로젝트를 사용한다. 아래와 같이 이 프로젝트를 Clone 한다.

```
$ git clone https://github.com/schacon/simplegit-progit
```

이 프로젝트 디렉토리에서 git log 명령을 실행하면 아래와 같이 출력된다.

```
$ git log
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    changed the version number

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 16:40:33 2008 -0700

    removed unnecessary test

commit a11bef06a3f659402fe7563abf99ad00de2209e6
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 10:31:28 2008 -0700

    first commit
```

특별한 아규먼트 없이 git log 명령을 실행하면 저장소의 커밋 히스토리를 시간순으로 보여준다. 즉, 가장 최근의 커밋이 가장 먼저 나온다. 그리고 이어서 각 커밋의 SHA-1 체크섬, 저자 이름, 저자 이메일, 커밋한 날짜, 커밋 메시지를 보여준다.

원하는 히스토리를 검색할 수 있도록 git log 명령은 매우 다양한 옵션을 지원한다. 여기에서는 자주 사용하는 옵션을 설명한다.

여러 옵션 중 -p, --patch 는 굉장히 유용한 옵션이다. -p 는 각 커밋의 diff 결과를 보여준다. 다른 유용한 옵션으로 `-2`가 있는데 최근 두 개의 결과만 보여주는 옵션이다:

```
$ git log -p -2
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    changed the version number

diff --git a/Rakefile b/Rakefile
index a874b73..8f94139 100644
--- a/Rakefile
+++ b/Rakefile
@@ -5,7 +5,7 @@ require 'rake/gempackagetask'
 spec = Gem::Specification.new do |s|
     s.platform  =   Gem::Platform::RUBY
     s.name      =   "simplegit"
-    s.version   =   "0.1.0"
+    s.version   =   "0.1.1"
     s.author    =   "Scott Chacon"
     s.email     =   "schacon@gee-mail.com"
     s.summary   =   "A simple gem for using Git in Ruby code."

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 16:40:33 2008 -0700

    removed unnecessary test

diff --git a/lib/simplegit.rb b/lib/simplegit.rb
index a0a60ae..47c6340 100644
--- a/lib/simplegit.rb
+++ b/lib/simplegit.rb
@@ -18,8 +18,3 @@ class SimpleGit
     end

 end
-
-if $0 == __FILE__
-  git = SimpleGit.new
-  puts git.show
-end
```

이 옵션은 직접 diff를 실행한 것과 같은 결과를 출력하기 때문에 동료가 무엇을 커밋했는지 리뷰하고 빨리 조회하는데 유용하다. 또 `git log` 명령에는 히스토리의 통계를 보여주는 옵션도 있다. `--stat` 옵션으로 각 커밋의 통계 정보를 조회할 수 있다.

```
$ git log --stat
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    changed the version number

 Rakefile | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 16:40:33 2008 -0700

    removed unnecessary test

 lib/simplegit.rb | 5 -----
 1 file changed, 5 deletions(-)

commit a11bef06a3f659402fe7563abf99ad00de2209e6
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 10:31:28 2008 -0700

    first commit

 README           |  6 ++++++
 Rakefile         | 23 +++++++++++++++++++++++
 lib/simplegit.rb | 25 +++++++++++++++++++++++++
 3 files changed, 54 insertions(+)
```

이 결과에서 `--stat` 옵션은 어떤 파일이 수정됐는지, 얼마나 많은 파일이 변경됐는지, 또 얼마나 많은 라인을 추가하거나 삭제했는지 보여준다. 요약정보는 가장 뒤쪽에 보여준다.

`git log` 명령의 기본적인 옵션과 출력물의 형식에 관련된 옵션을 살펴보았다. `git log` 명령은 앞서 살펴본 것보다 더 많은 옵션을 지원한다.

2.4. 되돌리기

일을 하다보면 모든 단계에서 어떤 것은 되돌리고(Undo) 싶을 때가 있다. 이번에는 우리가 한 일을 되돌리는 방법을 살펴본다. Git을 사용하면 실수는 대부분 복구할 수 있지만 되돌린 것은 복구할 수 없다.

2.4.1. 파일 상태를 Unstage로 변경하기

다음은 **Staging Area**와 **워킹 트리** 사이를 넘나드는 방법을 설명한다. 두 영역의 상태를 확인할 때마다 변경된 상태를 되돌리는 방법을 알려주기 때문에 매우 편리하다. 예를 들어 파일을 두 개 수정하고서 따로따로 커밋하려고 했지만, 실수로 `git add *` 라고 실행해 버렸다. 이제 둘 중 하나를 어떻게 꺼낼까? 우선 `git status` 명령으로 확인해보자.

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

    new file:  CONTRIBUTING.md
    modified:  README
```

3번째 줄에 `git reset HEAD <file>…` 메시지가 보인다. 이 명령으로 `Unstaged` 상태로 변경할 수 있다. `README` 파일을 Unstaged 상태로 변경해보자.

```
$ git reset HEAD README
Unstaged changes after reset:
M   README
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        new file:   CONTRIBUTING.md

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:   README
```

`README` 파일은 Unstaged 상태가 됐다. 이 동작을 풀어 설명하면, **Staging Area**의 `README` 파일이 HEAD가 가르키는 **Git 저장소**의 스냅샷의 README로 리셋된 것이다.

`git reset` 명령은 매우 위험하다. `--hard` 옵션과 함께 사용하면 더욱 위험하다. 하지만 위에서 처럼 옵션 없이 사용하면 워킹 디렉토리의 파일은 건드리지 않는다.

2.4.2. Modified 파일 되돌리기

워킹 트리의 파일을 되돌리는 방법이 무엇일까? `git status` 명령이 친절하게 알려준다.

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

        new file:   CONTRIBUTING.md

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:   README
```

위의 메시지가 알려주는 대로 `git checkout -- README`를 해보자.

```
$ git checkout -- README
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        new file:   CONTRIBUTING.md
```

정상적으로 복원된 것을 알 수 있다.

`git checkout -- [file]` 명령은 **워킹 트리**의 파일을 최신의 스냅샷의 파일로 덮어 쓴다. 파일을 덮어썼기 때문에 수정한 내용은 전부 사라진다. 수정한 내용이 진짜 마음에 들지 않을 때만 사용하자. 단, **Staging Area**의 내용은 건드리지 않는다.

변경한 내용을 쉽게 버릴수는 없고 당장 되돌려야만 하는 상황이라면 `branch`나 `Stash`를 사용하자. 해당 `stash`는 이 책에서 다루지 않으니 인터넷에서 검색해보라.

2.5. 리모트 저장소

리모트 저장소를 관리할 줄 알아야 다른 사람과 함께 일할 수 있다. 리모트 저장소는 인터넷이나 네트워크 어딘가에 있는 저장소를 말한다. 다른 사람들과 함께 일한다는 것은 리모트 저장소를 관리하면서 데이터를 거기에 Push 하고 Pull 하는 것이다. 이번에는 리모트 저장소를 관리하는 방법에 대해 설명한다.

2.5.1. 리모트 저장소 확인하기

`git remote` 명령으로 현재 프로젝트에 등록된 리모트 저장소를 확인할 수 있다. 이 명령은 리모트 저장소의 단축 이름을 보여준다. 저장소를 Clone 하면 `origin`이라는 리모트 저장소가 자동으로 등록되기 때문에 `origin`이라는 이름을 볼 수 있다.

```
$ git clone https://github.com/schacon/ticgit
Cloning into 'ticgit'...
remote: Reusing existing pack: 1857, done.
remote: Total 1857 (delta 0), reused 0 (delta 0)
Receiving objects: 100% (1857/1857), 374.35 KiB | 268.00 KiB/s, done.
Resolving deltas: 100% (772/772), done.
Checking connectivity... done.
$ cd ticgit
$ git remote
origin
```

-v 옵션을 주어 단축이름과 URL을 함께 볼 수 있다.

```
$ git remote -v
origin  https://github.com/schacon/ticgit (fetch)
origin  https://github.com/schacon/ticgit (push)
```

2.5.2. 리모트 저장소 추가하기

이전 절에서도 `git clone` 명령이 묵시적으로 `origin` 리모트 저장소를 어떻게 추가되는지 설명했었지만 수박 겉핥기식으로 살펴봤을 뿐이었다. 여기에서는 리모트 저장소를 추가하는 방법을 자세하게 설명한다. 기존 워킹 디렉토리에 새 리모트 저장소를 쉽게 추가할 수 있는데 `git remote add <단축이름> <url>` 명령을 사용한다.

```
$ git remote
origin
$ git remote add pb https://github.com/paulboone/ticgit
$ git remote -v
origin  https://github.com/schacon/ticgit (fetch)
origin  https://github.com/schacon/ticgit (push)
pb  https://github.com/paulboone/ticgit (fetch)
pb  https://github.com/paulboone/ticgit (push)
```

이제 URL 대신에 `pb` 라는 이름을 사용할 수 있다. 예를 들어 로컬 저장소에는 없지만 Paul의 저장소에 있는 것을 가져오려면 아래과 같이 실행한다.

```
$ git fetch pb
remote: Counting objects: 43, done.
remote: Compressing objects: 100% (36/36), done.
remote: Total 43 (delta 10), reused 31 (delta 5)
Unpacking objects: 100% (43/43), done.
From https://github.com/paulboone/ticgit
 * [new branch]      master     -> pb/master
 * [new branch]      ticgit     -> pb/ticgit
```

로컬에서 `pb/master` 가 Paul의 master 브랜치이다. 이 브랜치를 브랜치를 어떻게 사용하는지는 Git 브랜치 에서 자세히 살펴본다.

2.6. 요약

이제 우리는 로컬에서 사용할 수 있는 Git 명령에 대한 기본 지식은 갖추었다. 저장소를 만들고 Clone 하는 방법, 수정하고 나서 Stage 하고 커밋하는 방법, 저장소의 히스토리를 조회하는 방법 등을 살펴보았다. 이어지는 장에서는 Git의 가장 강력한 기능인 브랜치 모델을 살펴볼 것이다.

Chapter 3. Git 브랜치

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 독립적으로 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.

3.1. 브랜치란 무엇인가

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

Git은 데이터를 일련의 스냅샷으로 기록한다는 것을 시작하기 에서 보여줬다.

커밋하면 Git은 현 **Staging Area**에 있는 데이터의 스냅샷에 대한 포인터, 저자나 커밋 메시지 같은 메타데이터, 이전 커밋에 대한 포인터 등을 포함하는 커밋 개체를 저장한다.

이전 커밋 포인터가 있어서 현재 커밋이 무엇을 기준으로 바뀌었는지를 알 수 있다. 최초 커밋을 제외한 나머지 커밋은 이전 커밋 포인터가 적어도 하나씩 있다.

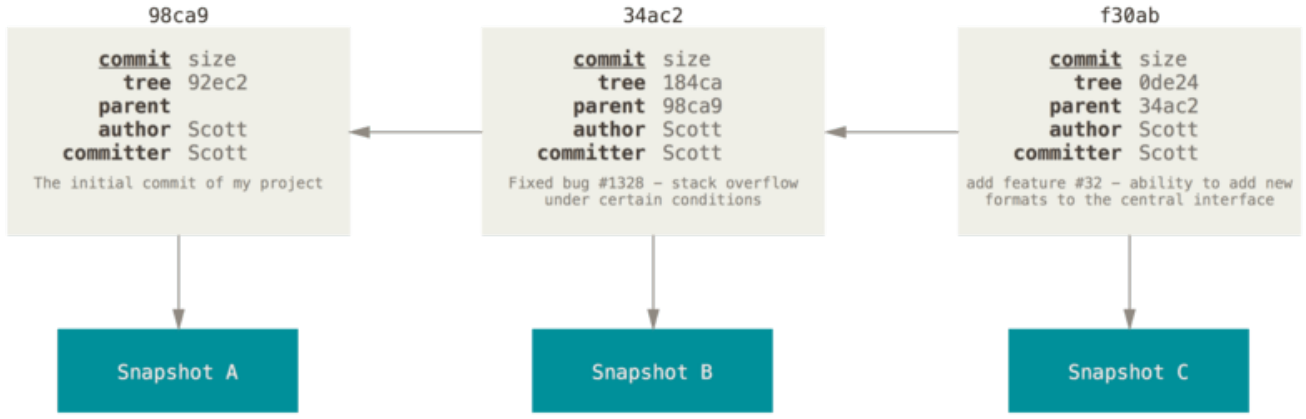


Figure 12. 커밋과 이전 커밋

Git의 **브랜치**는 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.

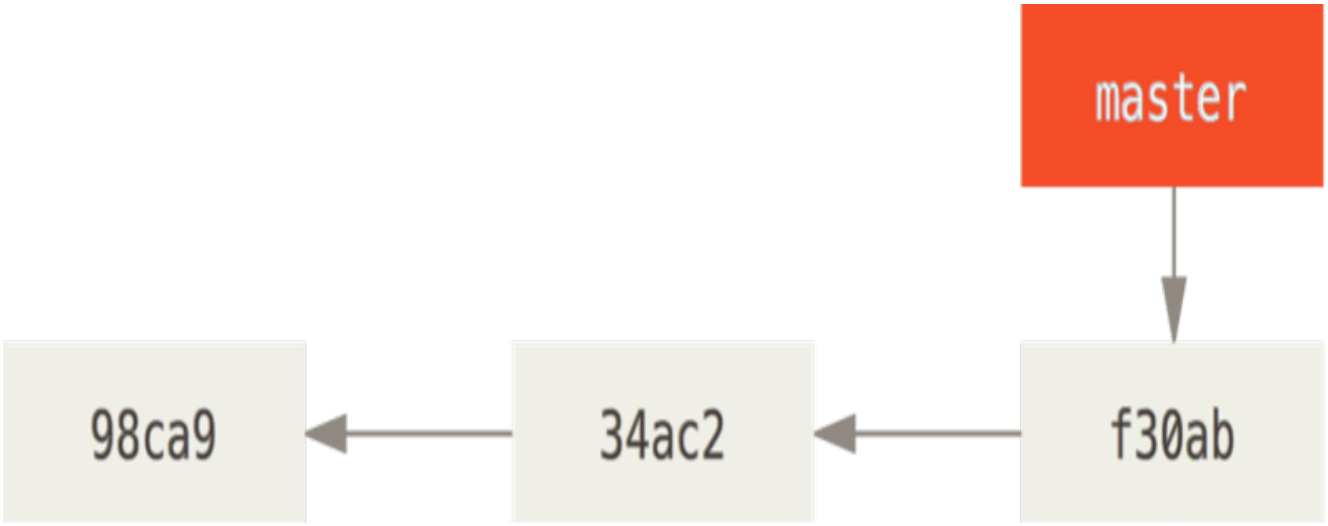


Figure 13. master 브랜치

3.1.1. 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.

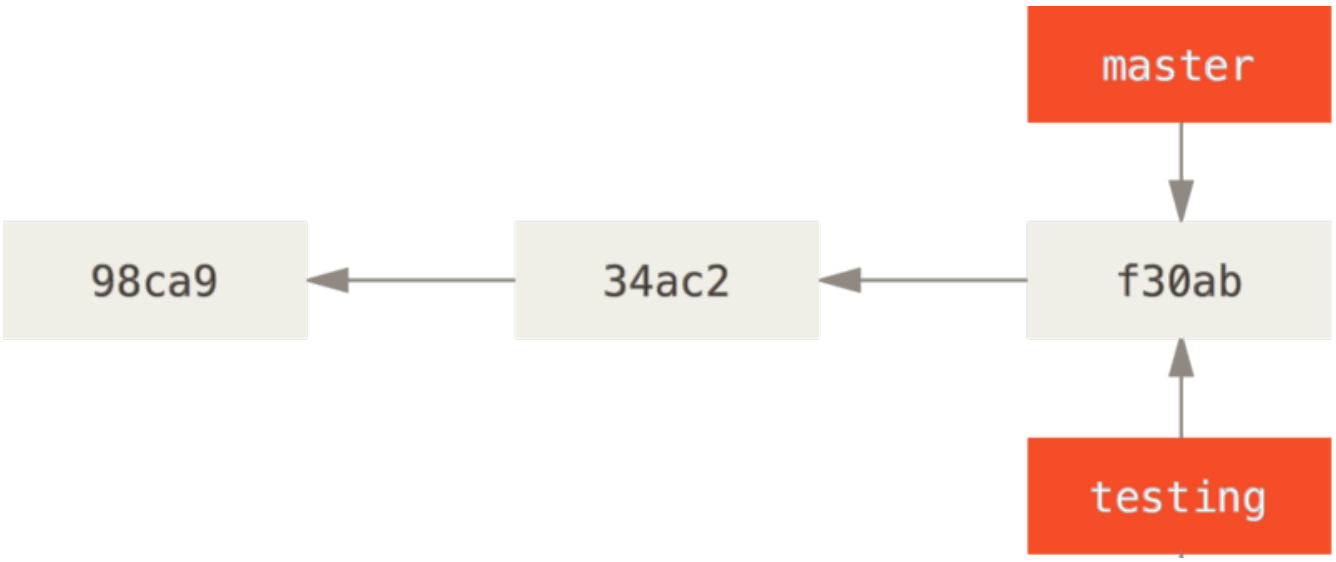


Figure 14. 한 커밋 히스토리를 가리키는 두 브랜치

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.

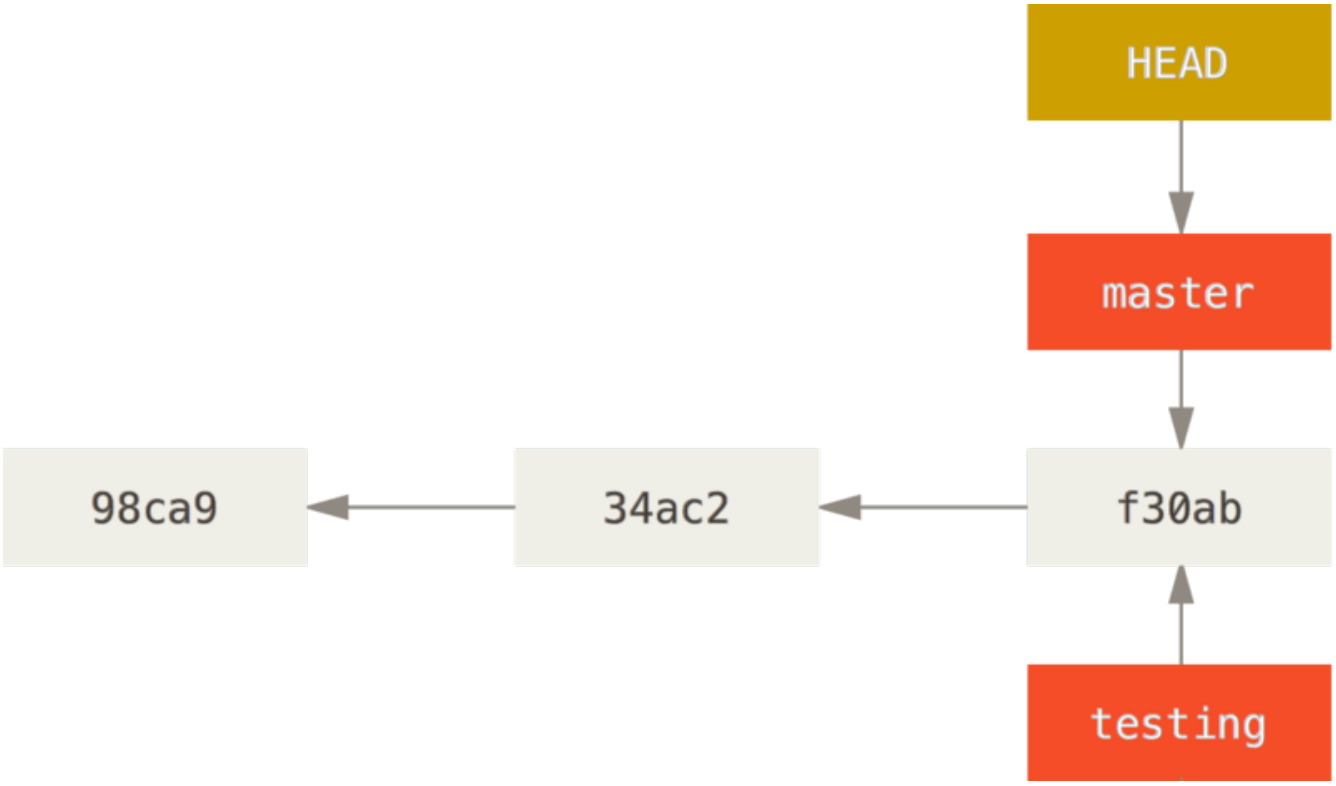


Figure 15. 현재 작업 중인 브랜치를 가리키는 HEAD

`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.

3.1.2. 브랜치 이동하기

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

```
$ git checkout testing
```

이렇게 하면 `HEAD`는 testing 브랜치를 가리킨다.

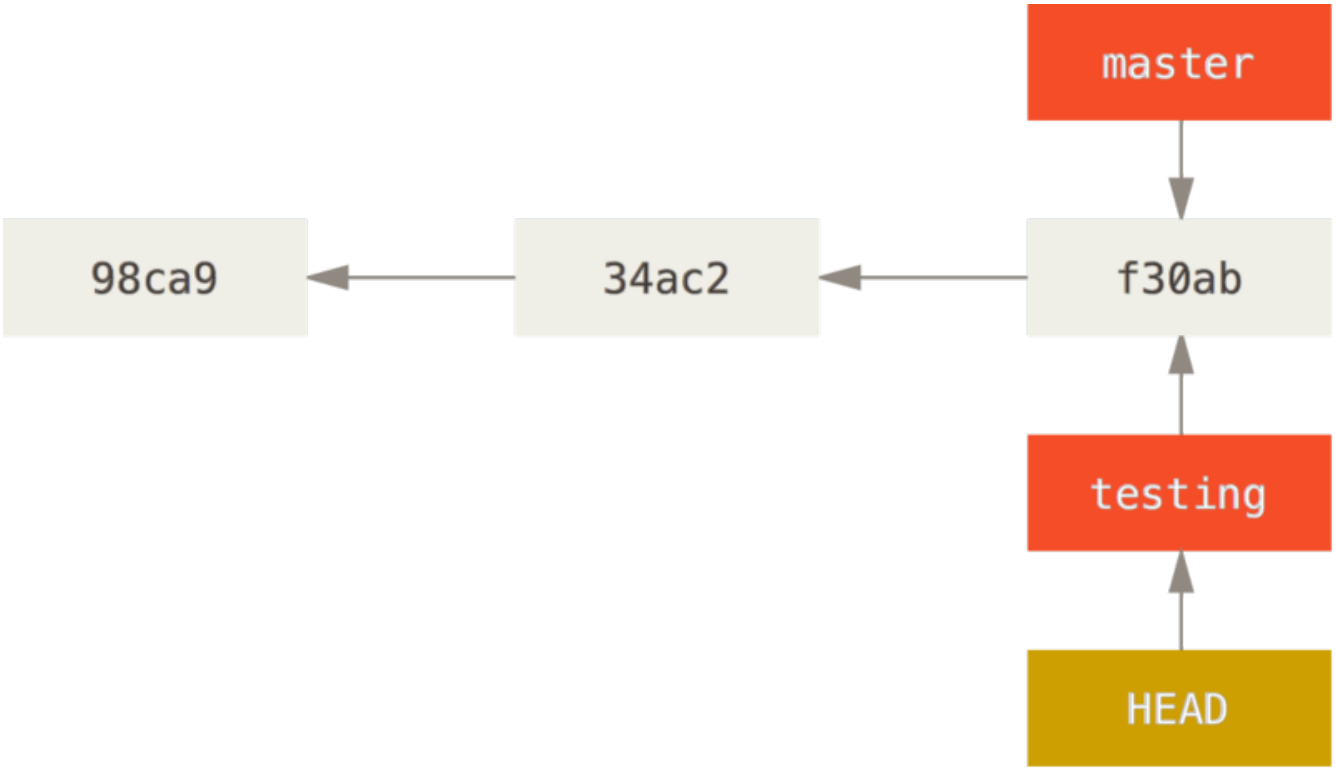


Figure 16. HEAD는 testing 브랜치를 가리킴

자, 이제 핵심이 보일 거다! 커밋을 새로 한 번 해보자.

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

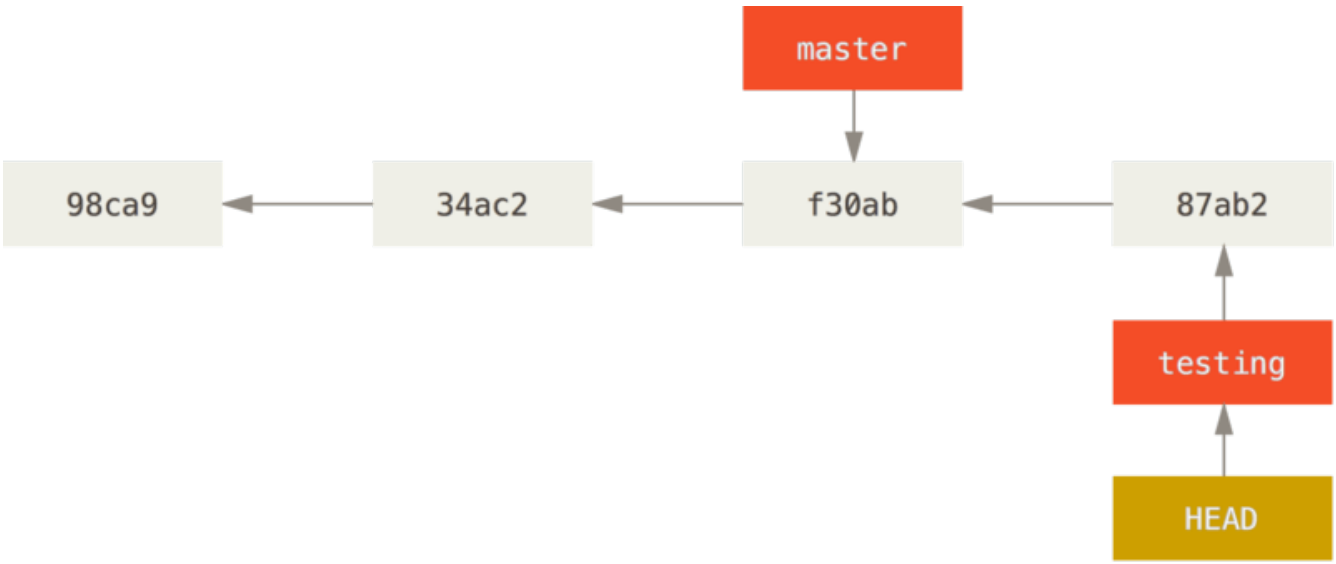


Figure 17. HEAD가 가리키는 testing 브랜치가 새 커밋을 가리킴

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

```
$ git checkout master
```

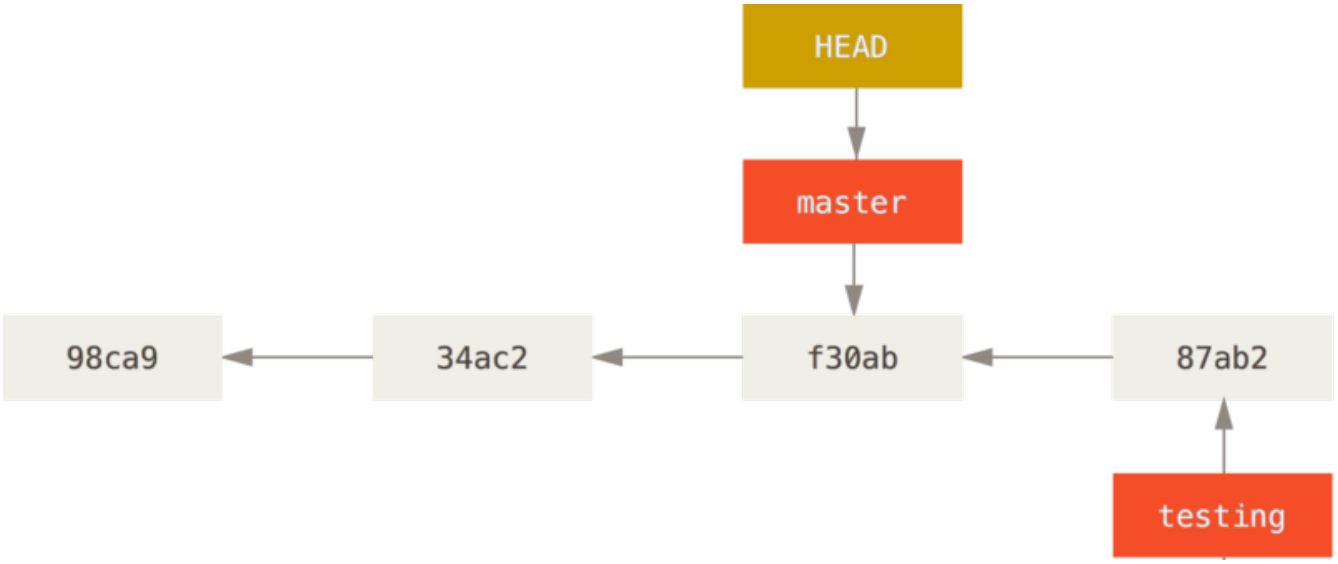


Figure 18. HEAD가 Checkout 한 브랜치로 이동함

방금 실행한 명령이 한 일은 두 가지다.

1. `master` 브랜치가 가리키는 커밋을 HEAD가 가리키게 한다
2. 워킹 트리의 파일도 그 시점으로 되돌려 놓는다.

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 트리**의 파일이 변경된다*

브랜치를 이동하면 **워킹 트리**의 파일이 변경된다는 점을 기억해두어야 한다. 이전에 작업했던 브랜치로 이동하면 **워킹 트리**의 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.

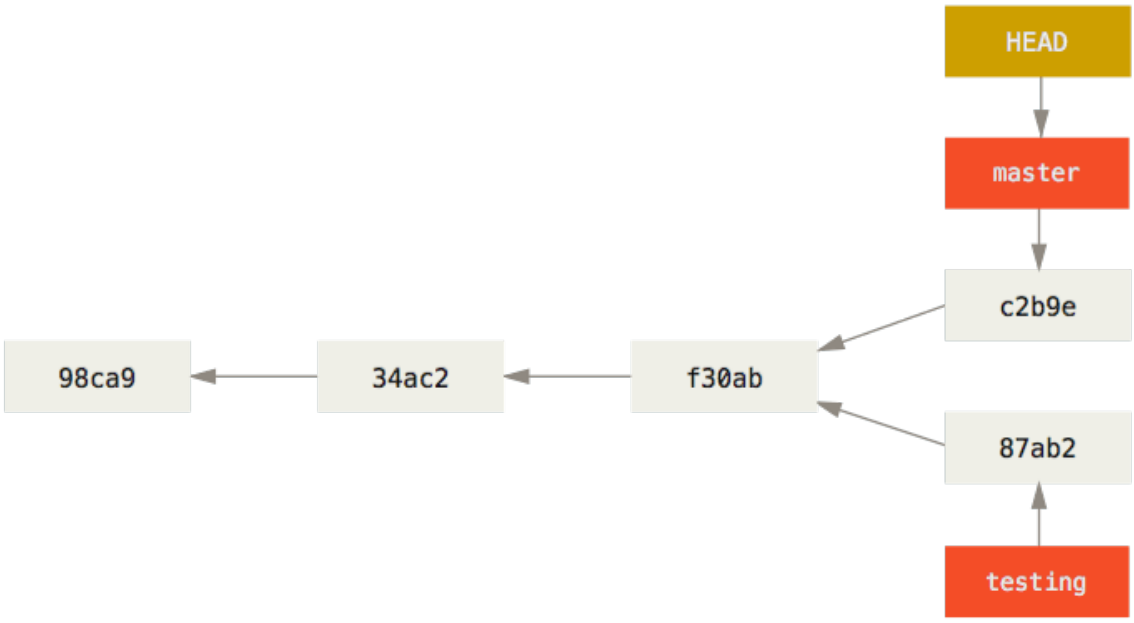


Figure 19. 갈라지는 브랜치

`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.

3.2. 브랜치와 Merge 의 기초

실제 개발과정에서 겪을 만한 예제를 하나 살펴보자. 브랜치와 Merge는 보통 이런 식으로 진행한다.

1. 웹사이트가 있고 뭔가 작업을 진행하고 있다.
2. 새로운 이슈를 처리할 새 Branch를 하나 생성한다.

3. 새로 만든 Branch에서 작업을 진행한다.

이때 중요한 문제가 생겨서 그것을 해결하는 Hotfix를 먼저 만들어야 한다. 그러면 아래와 같이 할 수 있다.

1. 새로운 이슈를 처리하기 이전의 운영(Production) 브랜치로 이동한다.
2. Hotfix 브랜치를 새로 하나 생성한다.
3. 수정한 Hotfix 테스트를 마치고 운영 브랜치로 Merge 한다.
4. 다시 작업하던 브랜치로 옮겨가서 하던 일 진행한다.

3.2.1. 브랜치의 기초

먼저 지금 작업하는 프로젝트에서 이전에 `master` 브랜치에 커밋을 몇 번 했다고 가정한다.

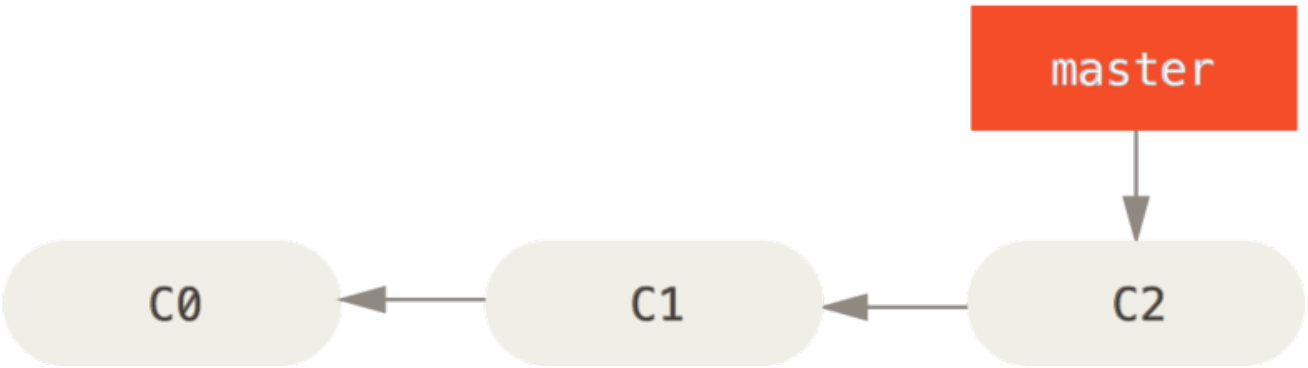


Figure 20. 현재 커밋 히스토리

이슈 관리 시스템에 등록된 53번 이슈를 처리한다고 하면 이 이슈에 집중할 수 있는 브랜치를 새로 하나 만든다. 브랜치를 만들면서 Checkout까지 한 번에 하려면 `git checkout` 명령에 `-b` 라는 옵션을 추가한다.

```
$ git checkout -b iss53
Switched to a new branch "iss53"
```

위 명령은 아래 명령을 줄여놓은 것이다.

```
$ git branch iss53
$ git checkout iss53
```

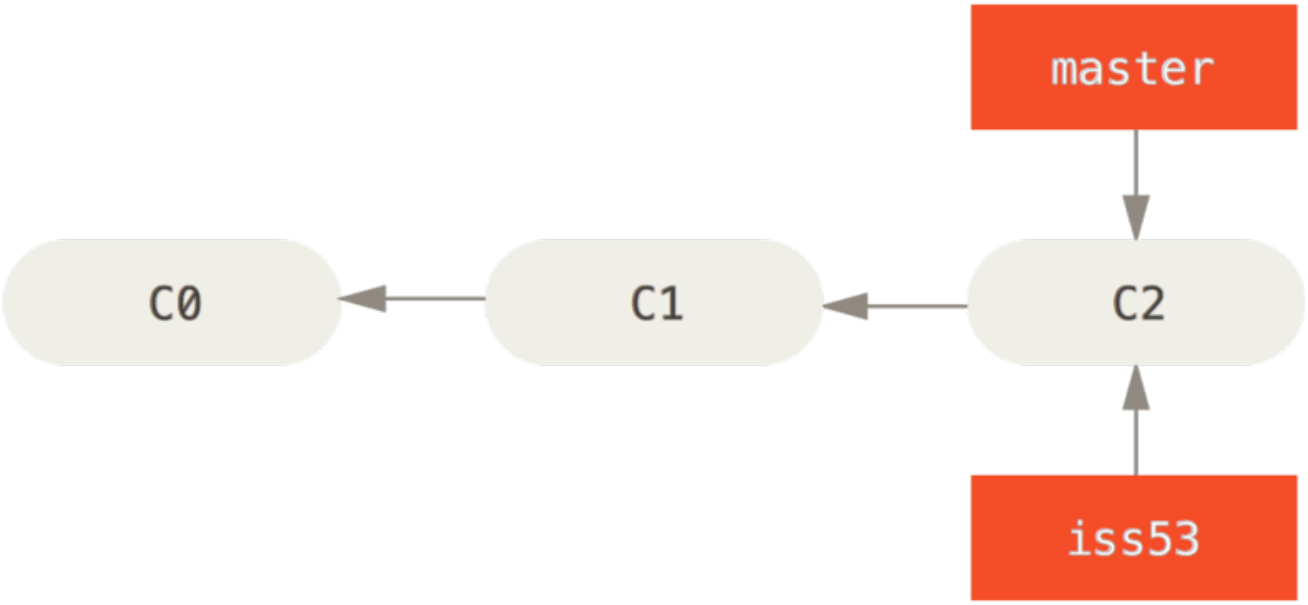


Figure 21. 브랜치 포인터를 새로 만듦

iss53 브랜치를 Checkout 했기 때문에(즉, HEAD 는 iss53 브랜치를 가리킨다) 뭔가 일을 하고 커밋하면 iss53 브랜치가 앞으로 나아간다.

```
$ vim index.html
$ git commit -a -m 'added a new footer [issue 53]'
```

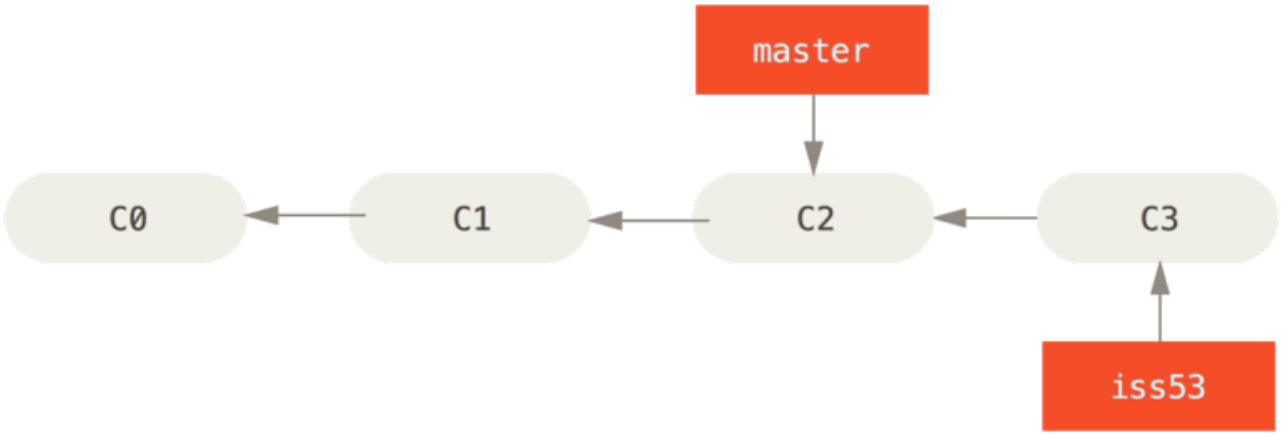


Figure 22. 진행 중인 iss53 브랜치

다른 상황을 가정해보자. 만드는 사이트에 문제가 생겨서 즉시 고쳐야 한다. 버그를 해결한 Hotfix에 iss53 이 섞이는 것을 방지하기 위해 iss53 과 관련된 코드를 어딘가에 저장해두고 원래 운영 환경의 소스로 복구해야 한다. Git을 사용하면 이런 노력을 들일 필요 없이 그냥 master 브랜치로 돌아가면 된다.

그렇지만, 브랜치를 이동하려면 해야 할 일이 있다. 아직 커밋하지 않은 파일이 Checkout 할 브랜치와 충돌 나면 브랜치를 변경할 수 없다. 브랜치를 변경할 때는 워킹 디렉토리를 정리하는 것이 좋다. 이런 문제를 다루는 방법은(주로, Stash이나 커밋 Amend에 대해) 나중에 다룰 것이다. 지금은 작업하던 것을 모두 커밋하고 master 브랜치로 옮긴다:

```
$ git checkout master
Switched to branch 'master'
```

이때 워킹 디렉토리는 53번 이슈를 시작하기 이전 모습으로 되돌려지기 때문에 새로운 문제에 집중할 수 있는 환경이 만들어진다. Git은 자동으로 워킹 디렉토리에 파일들을 추가하고, 지우고, 수정해서 Checkout 한 브랜치의 마지막 스냅샷으로 되돌려 놓는다는 것을 기억해야 한다.

이젠 해결해야 할 핫픽스가 생겼을 때를 살펴보자. `hotfix`라는 브랜치를 만들고 새로운 이슈를 해결할 때까지 사용한다.

```
$ git checkout -b hotfix
Switched to a new branch 'hotfix'
$ vim index.html
$ git commit -a -m 'fixed the broken email address'
[hotfix 1fb7853] fixed the broken email address
 1 file changed, 2 insertions(+)
```

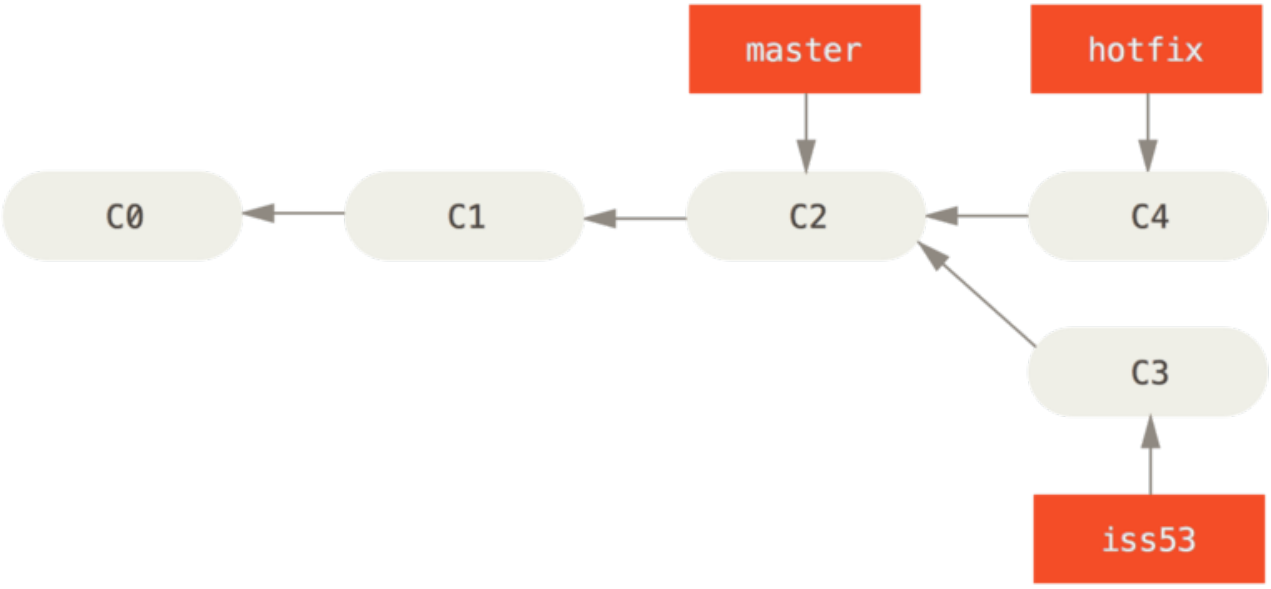


Figure 23. `master` 브랜치에서 갈라져 나온 hotfix 브랜치

운영 환경에 적용하려면 문제를 제대로 고쳤는지 테스트하고 최종적으로 운영환경에 배포하기 위해 `hotfix` 브랜치를 `master` 브랜치에 합쳐야 한다. `git merge` 명령으로 아래와 같이 한다.

```
$ git checkout master
$ git merge hotfix
Updating f42c576..3a0874c
Fast-forward
 index.html | 2 ++
 1 file changed, 2 insertions(+)
```

Merge 메시지에서 `fast-forward` 가 보이는가. `hotfix` 브랜치가 가리키는 `C4` 커밋이 `C2` 커밋에 기반한 브랜치이기 때문에 브랜치 포인터는 Merge 과정 없이 그저 최신 커밋으로 이동한다. 이런 Merge 방식을 `Fast forward` 라고 부른다. 다시 말해 A 브랜치에서 다른 B 브랜치를 Merge 할 때 B 브랜치가 A 브랜치 이후의 커밋을 가리키고 있으면 그저 A 브랜치가 B 브랜치와 동일한 커밋을 가리키도록 이동시킬 뿐이다.

이제 `hotfix`는 `master` 브랜치에 포함됐고 운영환경에 적용할 수 있는 상태가 되었다고 가정해보자.

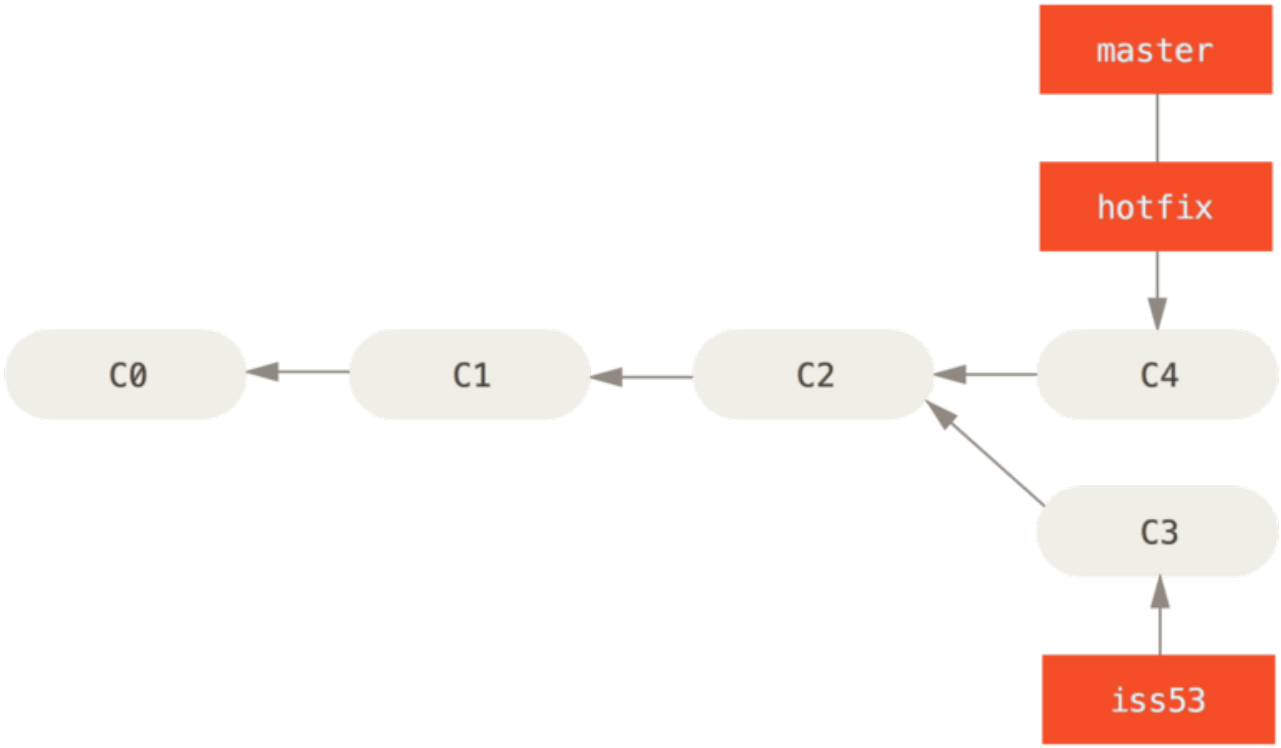


Figure 24. Merge 후 hotfix 같은 것을 가리키는 master 브랜치

급한 문제를 해결하고 `master` 브랜치에 적용하고 나면 다시 일하던 브랜치로 돌아가야 한다. 이제 더 이상 필요없는 `hotfix` 브랜치는 삭제한다. `git branch` 명령에 `-d` 옵션을 주고 브랜치를 삭제한다.

```
$ git branch -d hotfix
Deleted branch hotfix (3a0874c).
```

자 이제 이슈 53번을 처리하던 환경으로 되돌아가서 하던 일을 계속 하자.

```
$ git checkout iss53
Switched to branch "iss53"
$ vim index.html
$ git commit -a -m 'finished the new footer [issue 53]'
[iss53 ad82d7a] finished the new footer [issue 53]
1 file changed, 1 insertion(+)
```

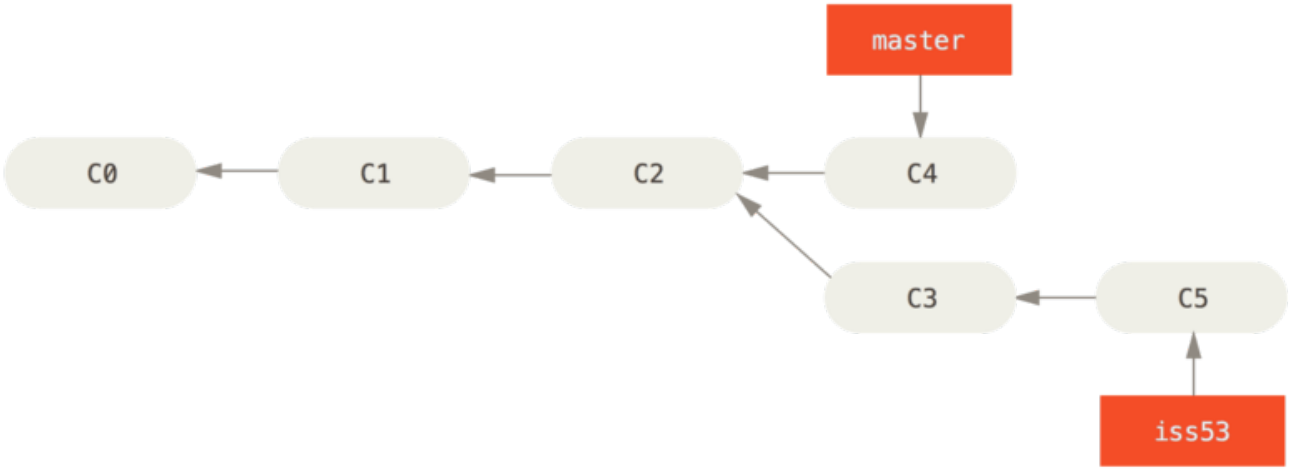


Figure 25. master와 별개로 진행하는 iss53 브랜치

위에서 작업한 `hotfix` 가 `iss53` 브랜치에 영향을 끼치지 않는다는 점을 이해하는 것이 중요하다. `git merge master` 명령으로 `master` 브랜치를 `iss53` 브랜치에 Merge 하면 `iss53` 브랜치에 `hotfix` 가 적용된다. 아니면 `iss53` 브랜치가 `master` 에 Merge 할 수 있는 수준이 될 때까지 기다렸다가 Merge 하면 `hotfix` 와 `iss53` 브랜치가 합쳐진다.

3.2.2. Merge 의 기초

53번 이슈를 다 구현하고 master 브랜치에 Merge 하는 과정을 살펴보자. `iss53` 브랜치를 `master` 브랜치에 Merge 하는 것은 앞서 살펴본 `hotfix` 브랜치를 Merge 하는 것과 비슷하다. `git merge` 명령으로 합칠 브랜치에서 합쳐질 브랜치를 Merge 하면 된다.

```
$ git checkout master
Switched to branch 'master'
$ git merge iss53
Merge made by the 'recursive' strategy.
index.html |    1 +
1 file changed, 1 insertion(+)
```

`hotfix` 를 Merge 했을 때와 메시지가 다르다. 현재 브랜치가 가리키는 커밋이 Merge 할 브랜치의 조상이 아니므로 Git은 `Fast-forward`로 Merge 하지 않는다. 이 경우에는 Git은 각 브랜치가 가리키는 커밋 두 개와 공통 조상 하나를 사용하여 3-way Merge를 한다.

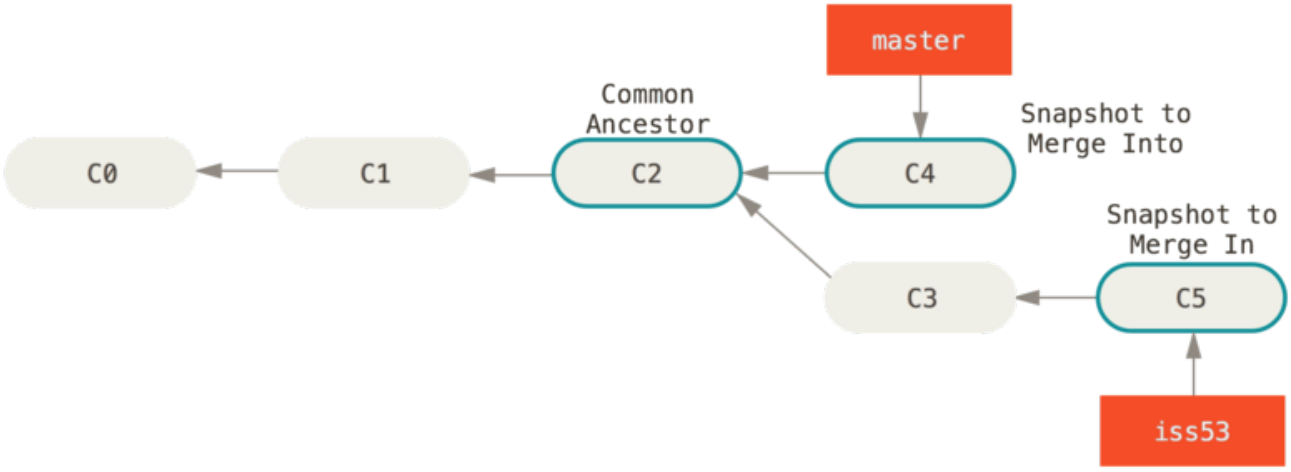


Figure 26. 커밋 3개를 Merge

단순히 브랜치 포인터를 최신 커밋으로 옮기는 게 아니라 3-way Merge 의 결과를 별도의 커밋으로 만들고 나서 해당 브랜치가 그 커밋을 가리키도록 이동시킨다. 그래서 이런 커밋은 부모가 여러 개고 Merge 커밋이라고 부른다.

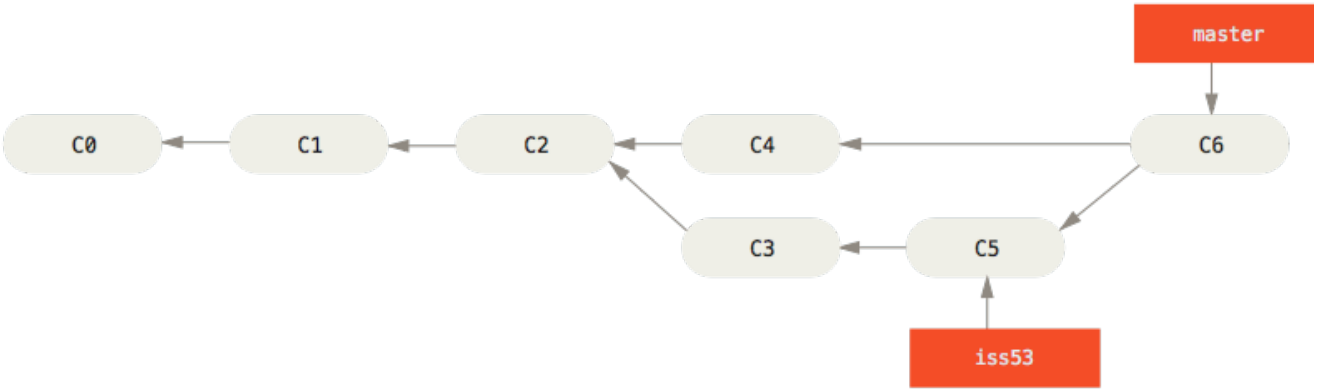


Figure 27. Merge 커밋

iss53 브랜치를 master에 Merge 하고 나면 더는 iss53 브랜치가 필요 없다. 다음 명령으로 브랜치를 삭제하고 이슈의 상태를 처리 완료로 표시한다.

```
$ git branch -d iss53
```

3.2.3. 충돌의 기초

가끔씩 3-way Merge가 실패할 때도 있다. Merge 하는 두 브랜치에서 같은 파일의 한 부분을 동시에 수정하고 Merge 하면 Git은 해당 부분을 Merge 하지 못한다. 예를 들어, 53번 이슈와 `hotfix` 가 같은 부분을 수정했다면 Git은 Merge 하지 못하고 아래와 같은 충돌(Conflict) 메시지를 출력한다.

```
$ git merge iss53
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
Automatic merge failed; fix conflicts and then commit the result.
```

Git은 자동으로 Merge 하지 못해서 새 커밋이 생기지 않는다. 변경사항의 충돌을 개발자가 해결하지 않는 한 Merge 과정을 진행할 수 없다. Merge 충돌이 일어났을 때 Git이 어떤 파일을 Merge 할 수 없었는지 살펴보려면 `git status` 명령을 이용한다.

```
$ git status
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")

Unmerged paths:
  (use "git add <file>..." to mark resolution)

    both modified:      index.html

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

충돌이 일어난 파일은 unmerged 상태로 표시된다. Git은 충돌이 난 부분을 표준 형식에 따라 표시해준다. 그러면 개발자는 해당 부분을 수동으로 해결한다. 충돌 난 부분은 아래와 같이 표시된다.

```
<<<<<<< HEAD:index.html
<div id="footer">contact : email.support@github.com</div>
=======
<div id="footer">
 please contact us at support@github.com
</div>
>>>>>>> iss53:index.html
```

`=======` 위쪽의 내용은 `HEAD` 버전(merge 명령을 실행할 때 작업하던 `master` 브랜치)의 내용이고 아래쪽은 `iss53` 브랜치의 내용이다. 충돌을 해결하려면 위쪽이나 아래쪽 내용 중에서 고르거나 새로 작성하여 Merge 한다. 아래는 아예 새로 작성하여 충돌을 해결하는 예제다.

```
<div id="footer">
please contact us at email.support@github.com
</div>
```

충돌한 양쪽에서 조금씩 가져와서 새로 수정했다. 그리고 `<<<<<<<`, `=======`, `>>>>>>>`가 포함된 행을 삭제했다. 이렇게 충돌한 부분을 해결하고 `git add` 명령으로 다시 Git에 저장한다.

Merge 도구를 종료하면 Git은 잘 Merge 했는지 물어본다. 잘 마쳤다고 입력하면 자동으로 `git add` 가 수행되고 해당 파일이 **Staging Area**에 저장된다. `git status` 명령으로 충돌이 해결된 상태인지 다시 한번 확인해볼 수 있다.

```
$ git status
On branch master
All conflicts fixed but you are still merging.
  (use "git commit" to conclude merge)

Changes to be committed:

    modified:   index.html
```

충돌을 해결하고 나서 해당 파일이 **Staging Area**에 저장됐는지 확인했으면 `git commit` 명령으로 Merge 한 것을 커밋한다.

3.3. 브랜치 관리

지금까지 브랜치를 만들고, Merge 하고, 삭제하는 방법에 대해서 살펴봤다. 브랜치를 관리하는 데 필요한 다른 명령도 살펴보자.

`git branch` 명령은 단순히 브랜치를 만들고 삭제하는 것이 아니다. 아무런 옵션 없이 실행하면 브랜치의 목록을 보여준다.

```
$ git branch
  iss53
* master
  testing
```

`*` 기호가 붙어 있는 `master` 브랜치는 현재 Checkout 해서 작업하는 브랜치를 나타낸다. 즉, 지금 수정한 내용을 커밋하면 `master` 브랜치에 커밋되고 포인터가 앞으로 한 단계 나아간다.

3.4. 리모트 브랜치

리모트 트래킹 브랜치는 리모트 브랜치를 추적하는 브랜치다. 리모트 트래킹 브랜치는 로컬에 있지만 임의로 움직일 수 없다. 리모트 서버에 연결할 때마다 리모트의 브랜치 업데이트 내용에 따라서 자동으로 갱신될 뿐이다. 리모트 트래킹 브랜치는 일종의 북마크라고 할 수 있다. 리모트 저장소에 마지막으로 연결했던 순간에 브랜치가 무슨 커밋을 가리키고 있었는지를 나타낸다.

리모트 트래킹 브랜치의 이름은 `<remote>/<branch>` 형식으로 되어 있다. `git.ourcompany.com` 이라는 Git 서버가 있고 이 서버의 저장소를 하나 Clone 하면 Git은 자동으로 `origin` 이라는 이름을 붙인다. `origin` 으로부터 저장소 데이터를 모두 내려받고 `master` 브랜치를 가리키는 포인터를 만든다. 이 포인터는 `origin/master` 라고 부르고 멋대로 조종할 수 없다. 그리고 Git은 로컬의 `master` 브랜치가 `origin/master` 를 가리키게 한다. 이제 이 `master` 브랜치에서 작업을 시작할 수 있다.

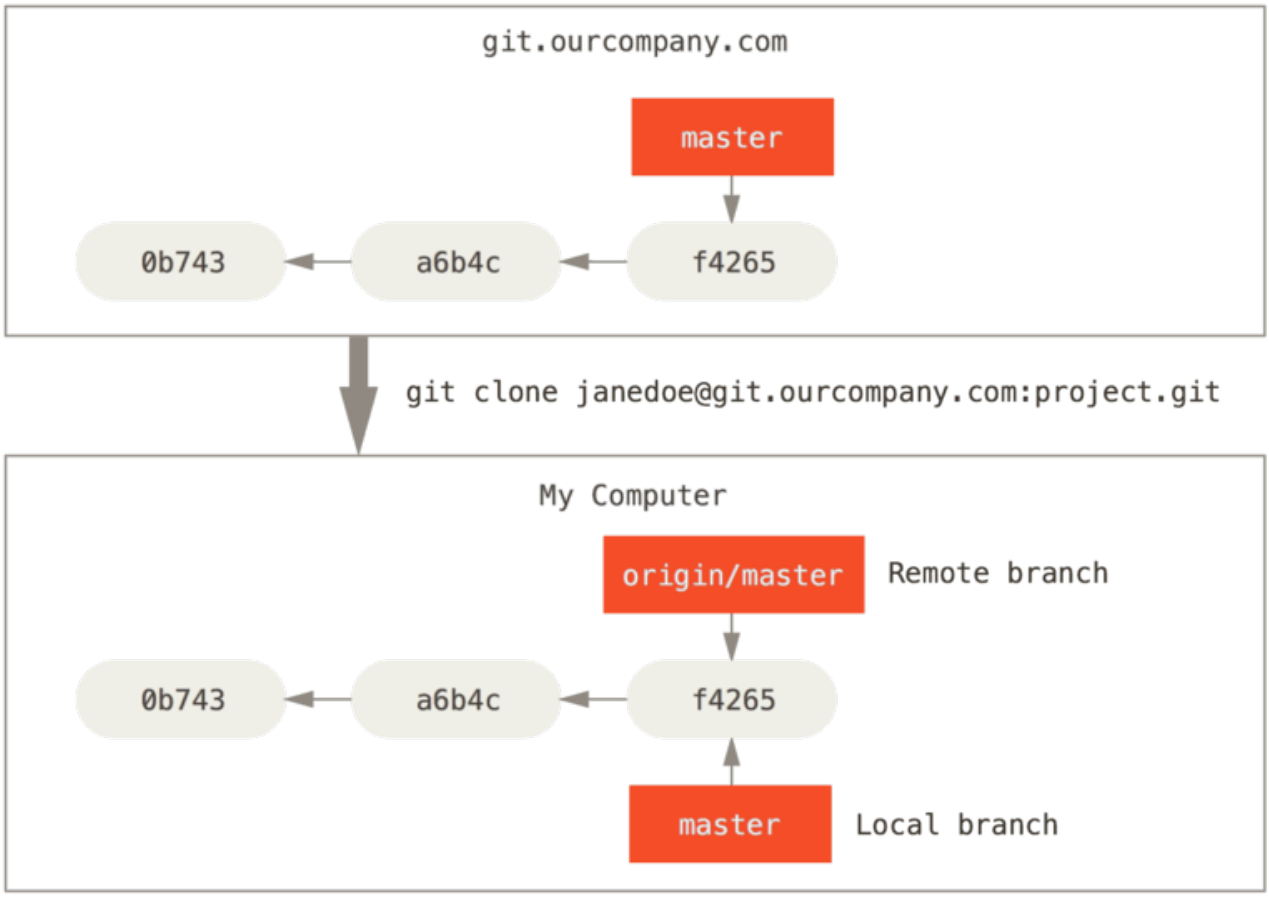
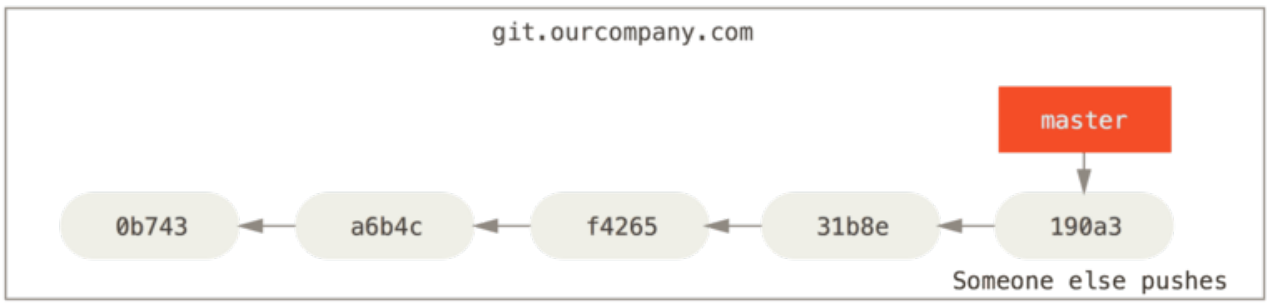


Figure 28. Clone 이후 서버와 로컬의 master 브랜치

로컬 저장소에서 어떤 작업을 하고 있는데 동시에 다른 팀원이 `git.ourcompany.com` 서버에 Push 하고 `master` 브랜치를 업데이트한다. 그러면 이제 팀원 간의 히스토리는 서로 달라진다. 서버 저장소로부터 어떤 데이터도 주고받지 않아서 `origin/master` 포인터는 그대로다.



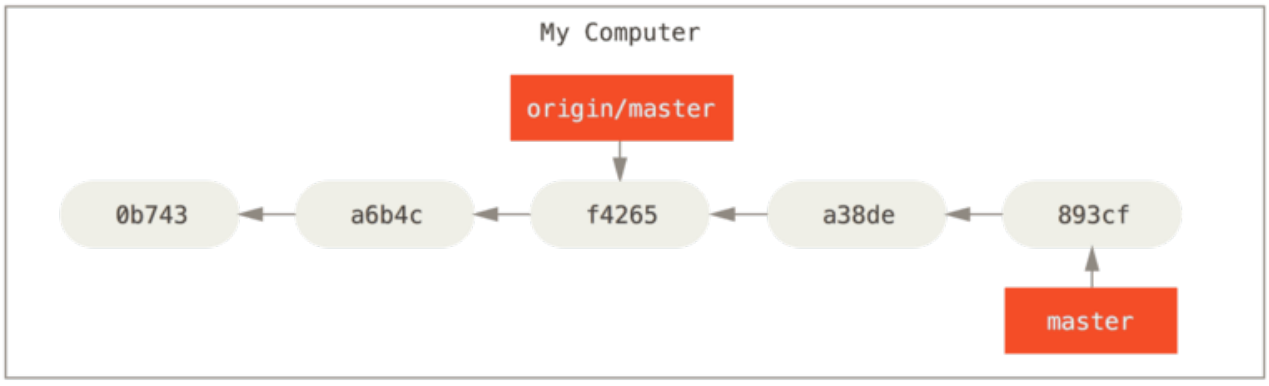


Figure 29. 로컬과 서버의 커밋 히스토리는 독립적임

리모트 서버로부터 저장소 정보를 동기화하려면 `git fetch origin` 명령을 사용한다. 명령을 실행하면 우선 `origin` 서버의 주소 정보(이 예에서는 `git.ourcompany.com`)를 찾아서, 현재 로컬의 저장소가 갖고 있지 않은 새로운 정보가 있으면 모두 내려받고, 받은 데이터를 로컬 저장소에 업데이트하고 나서, origin/master 포인터의 위치를 최신 커밋으로 이동시킨다.

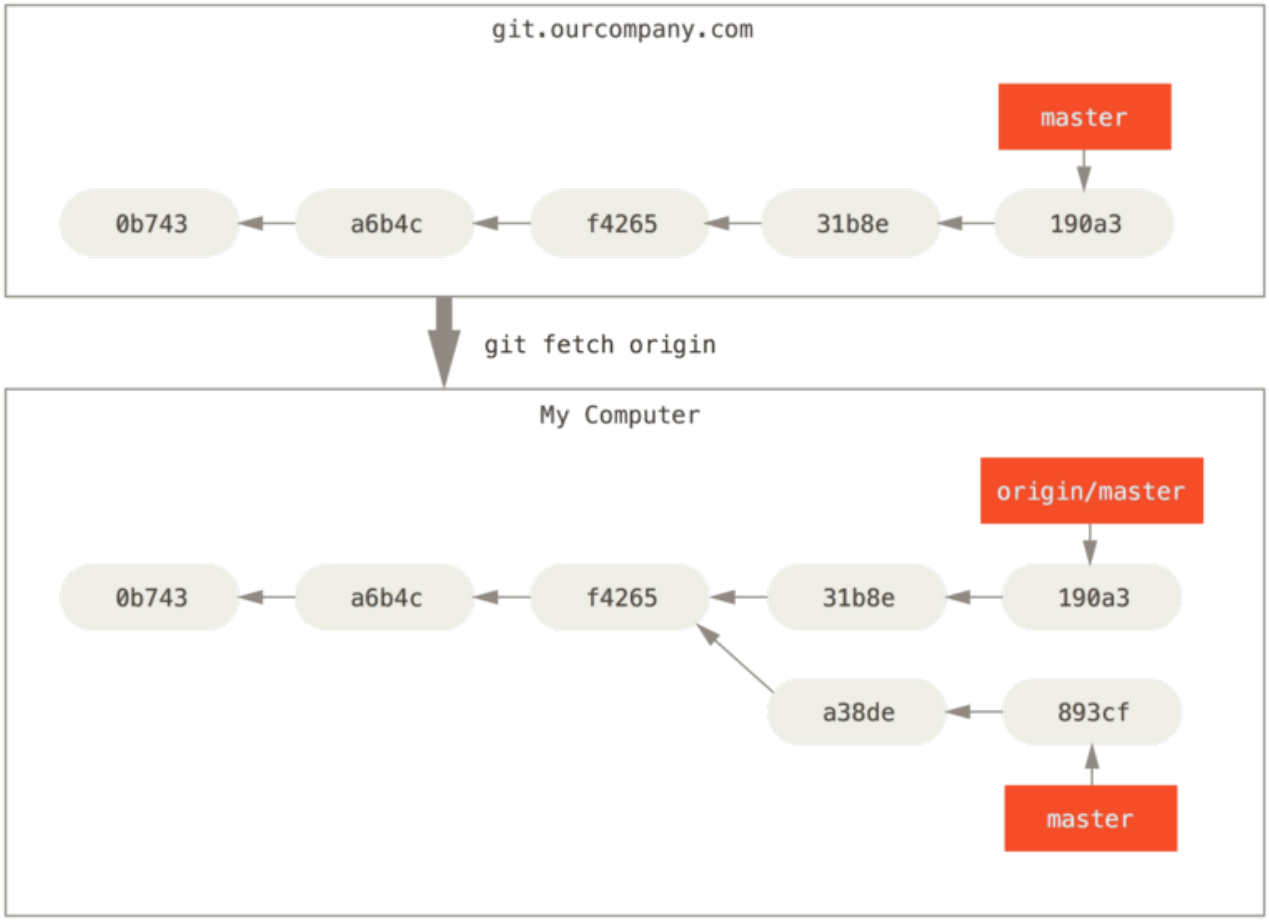


Figure 30. `git fetch` 명령은 리모트 브랜치 정보를 업데이트

3.4.1. Push 하기

로컬의 브랜치를 서버로 전송하려면 쓰기 권한이 있는 리모트 저장소에 Push 해야 한다. 로컬 저장소의 브랜치는 자동으로 리모트 저장소로 전송되지 않는다. 명시적으로 브랜치를 Push 해야 정보가 전송된다. 따라서 리모트 저장소에 전송하지 않고 로컬 브랜치에만 두는 비공개 브랜치를 만들 수 있다. 또 다른 사람과 협업하기 위해 토픽 브랜치만 전송할 수도 있다.

`serverfix` 라는 브랜치를 다른 사람과 공유할 때도 브랜치를 처음 Push 하는 것과 같은 방법으로 Push 한다. 아래와 같이 `git push <remote> <branch>` 명령을 사용한다.

```
$ git push origin serverfix
Counting objects: 24, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (15/15), done.
Writing objects: 100% (24/24), 1.91 KiB | 0 bytes/s, done.
Total 24 (delta 2), reused 0 (delta 0)
To https://github.com/schacon/simplegit
 * [new branch]      serverfix -> serverfix
```

이것은 `serverfix` 라는 로컬 브랜치를 서버로 Push 하는데 리모트의 `serverfix` 브랜치로 업데이트한다는 것을 의미한다.

나중에 누군가 저장소를 Fetch 하고 나서 서버에 있는 `serverfix` 브랜치에 접근할 때 `origin/serverfix` 라는 이름으로 접근할 수 있다.

```
$ git fetch origin
remote: Counting objects: 7, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 3 (delta 0)
Unpacking objects: 100% (3/3), done.
From https://github.com/schacon/simplegit
 * [new branch]      serverfix    -> origin/serverfix
```

여기서 짚고 넘어가야 할 게 있다. Fetch 명령으로 리모트 트래킹 브랜치를 내려받는다고 해서 로컬 저장소에 수정할 수 있는 브랜치가 새로 생기는 것이 아니다. 다시 말해서 `serverfix` 라는 브랜치가 생기는 것이 아니라 그저 수정 못하는 `origin/serverfix` 브랜치 포인터가 생기는 것이다.

새로 받은 브랜치의 내용을 Merge 하려면 `git merge origin/serverfix` 명령을 사용한다. Merge 하지 않고 리모트 트래킹 브랜치에서 시작하는 새 브랜치를 만들려면 아래와 같은 명령을 사용한다.

```
$ git checkout -b serverfix origin/serverfix
Branch serverfix set up to track remote branch serverfix from origin.
Switched to a new branch 'serverfix'
```

그러면 `origin/serverfix` 에서 시작하고 수정할 수 있는 `serverfix` 라는 로컬 브랜치가 만들어진다.

3.4.2. 브랜치 추적

리모트 트래킹 브랜치를 로컬 브랜치로 Checkout 하면 자동으로 `트래킹(Tracking) 브랜치` 가 만들어진다 (트래킹 하는 대상 브랜치를 `Upstream 브랜치` 라고 부른다). 트래킹 브랜치는 리모트 브랜치와 직접적인 연결고리가 있는 로컬 브랜치이다. 트래킹 브랜치에서 `git pull` 명령을 내리면 리모트 저장소로부터 데이터를 내려받아 연결된 리모트 브랜치와 자동으로 Merge 한다.

서버로부터 저장소를 Clone을 하면 Git은 자동으로 `master` 브랜치를 `origin/master` 브랜치의 트래킹 브랜치로 만든다. 트래킹 브랜치를 직접 만들 수 있는데 리모트를 `origin` 이 아닌 다른 리모트로 할 수도 있고, 브랜치도 `master` 가 아닌 다른 브랜치로 추적하게 할 수 있다. `git checkout -b <branch> <remote>/<branch>` 명령으로 간단히 트래킹 브랜치를 만들 수 있다. `--track` 옵션을 사용하여 로컬 브랜치 이름을 자동으로 생성할 수 있다.

```
$ git checkout --track origin/serverfix
Branch serverfix set up to track remote branch serverfix from origin.
Switched to a new branch 'serverfix'
```

이 명령은 매우 자주 쓰여서 더 생략할 수 있다. 입력한 브랜치가 있는 (a) 리모트가 딱 하나 있고 (b) 로컬에는 없으면 Git은 트래킹 브랜치를 만들어 준다.

```
$ git checkout serverfix
Branch serverfix set up to track remote branch serverfix from origin.
Switched to a new branch 'serverfix'
```

이미 로컬에 존재하는 브랜치가 리모트의 특정 브랜치를 추적하게 하려면 `git branch` 명령에 `-u` 나 `--set-upstream-to` 옵션을 붙여서 아래와 같이 설정한다.

```
$ git branch -u origin/serverfix
Branch serverfix set up to track remote branch serverfix from origin.
```

추적 브랜치가 현재 어떻게 설정되어 있는지 확인하려면 `git branch` 명령에 `-vv` 옵션을 더한다. 이 명령을 실행하면 로컬 브랜치 목록과 로컬 브랜치가 추적하고 있는 리모트 브랜치도 함께 보여준다. 게다가, 로컬 브랜치가 앞서가는지 뒤쳐지는지에 대한 내용도 보여준다.

```
$ git branch -vv
  iss53     7e424c3 [origin/iss53: ahead 2] forgot the brackets
  master    1ae2a45 [origin/master] deploying index fix
* serverfix f8674d9 [teamone/server-fix-good: ahead 3, behind 1] this should do it
  testing   5ea463a trying something new
```

위의 결과를 보면 `iss53` 브랜치는 `origin/iss53` 리모트 브랜치를 추적하고 있다는 것을 알 수 있고 `ahead` 표시를 통해 로컬 브랜치가 커밋 2개 앞서 있다(리모트 브랜치에는 없는 커밋이 로컬에는 존재)는 것을 알 수 있다. `master` 브랜치는 `origin/master` 브랜치를 추적하고 있으며 두 브랜치가 가리키는 커밋 내용이 같은 상태이다. 로컬 브랜치 중 `serverfix` 브랜치는 `server-fix-good` 이라는 `teamone` 리모트 서버의 브랜치를 추적하고 있으며 커밋 3개 앞서 있으며 동시에 커밋 1개로 뒤쳐져 있다. 이 말은 `serverfix` 브랜치에 서버로 보내지 않은 커밋이 3개, 서버의 브랜치에서 아직 로컬 브랜치로 머지하지 않은 커밋이 1개 있다는 말이다. 마지막 `testing` 브랜치는 추적하는 브랜치가 없는 상태이다.

3.4.3. Pull 하기

`git fetch` 명령을 실행하면 서버에는 존재하지만, 로컬에는 아직 없는 데이터를 받아와서 저장한다. 이 때 워킹 디렉토리의 파일 내용은 변경되지 않고 그대로 남는다. 서버로부터 데이터를 가져와서 저장해두고 사용자가 Merge 하도록 준비만 해둔다. 간단히 말하면 `git pull` 명령은 대부분 `git fetch` 명령을 실행하고 나서 자동으로 `git merge` 명령을 수행하는 것 뿐이다. 바로 앞 절에서 살펴본 대로 `clone` 이나 `checkout` 명령을 실행하여 추적 브랜치가 설정되면 `git pull` 명령은 서버로부터 데이터를 가져와서 현재 로컬 브랜치와 서버의 추적 브랜치를 Merge 한다.

3.4.4. 리모트 브랜치 삭제

동료와 협업하기 위해 리모트 브랜치를 만들었다가 작업을 마치고 `master` 브랜치로 Merge 했다. 협업하는 데 사용했던 그 리모트 브랜치는 이제 더 이상 필요하지 않기에 삭제할 수 있다. `git push` 명령에 `--delete` 옵션을

사용하여 리모트 브랜치를 삭제할 수 있다. `serverfix` 라는 리모트 브랜치를 삭제하려면 아래와 같이 실행한다.

```
$ git push origin --delete serverfix
To https://github.com/schacon/simplegit
 - [deleted]         serverfix
```

위 명령을 실행하면 서버에서 브랜치(즉 커밋을 가리키는 포인터) 하나가 사라진다. 서버에서 가비지 컬렉터가 동작하지 않는 한 데이터는 사라지지 않기 때문에 종종 의도치 않게 삭제한 경우에도 커밋한 데이터를 살릴 수 있다.

3.5. 요약

우리는 이 장에서 Git으로 브랜치를 만들고 Merge 의 기본적인 사용법을 다루었다. 이제 브랜치를 만들고 옮겨다니고 Merge 하는 것에 익숙해졌을 것으로 생각한다.

Chapter 4. 분산 환경에서의 Git

앞 장에서 다른 개발자와 코드를 공유하는 리모트 저장소를 만드는 법을 배웠다. 로컬에서 작업하는 데 필요한 기본적인 명령어에는 어느 정도 익숙해졌다. 이제는 분산 환경에서 Git이 제공하는 기능을 어떻게 효율적으로 사용할지를 배운다.

이번 장에서는 분산 환경에서 Git을 어떻게 사용할 수 있을지 살펴본다. 프로젝트 기여자 입장에서 중요한 몇 가지 요소를 살펴본다.

4.1. 브랜치 워크플로

이 절에서는 Git 브랜치가 유용한 워크플로를 살펴본다. 여기서 설명하는 워크플로를 개발에 적용하면 도움이 될 것이다.

4.1.1. Long-Running 브랜치

Git은 장기간에 걸쳐서 한 브랜치를 다른 브랜치와 여러 번 Merge 하는 것이 쉬운 편이다. 그래서 개발 과정에서 필요한 용도에 따라 브랜치를 만들어 두고 계속 사용할 수 있다. 그리고 정기적으로 브랜치를 다른 브랜치로 Merge 한다.

이런 접근법에 따라서 Git 개발자가 많이 선호하는 워크플로가 하나 있다.

- `master` 브랜치에는 배포했거나 배포할 코드만 Merge 해서 안정 버전의 코드를 둔다.
- `develop` 브랜치에는 개발을 진행하고 안정화하는 용도로 사용한다. 이 브랜치는 언젠가 안정 상태가 되겠지만, 항상 안정 상태를 유지해야 하는 것이 아니다. 테스트를 거쳐서 안정적이라고 판단되면 `master` 브랜치에 Merge 한다.
- `topic` 브랜치(앞서 살펴본 `iss53` 브랜치 같은 짧은 호흡 브랜치)에는 해당 토픽을 처리하고 테스트해서 버그도 없고 안정적이면 그때 Merge한다.

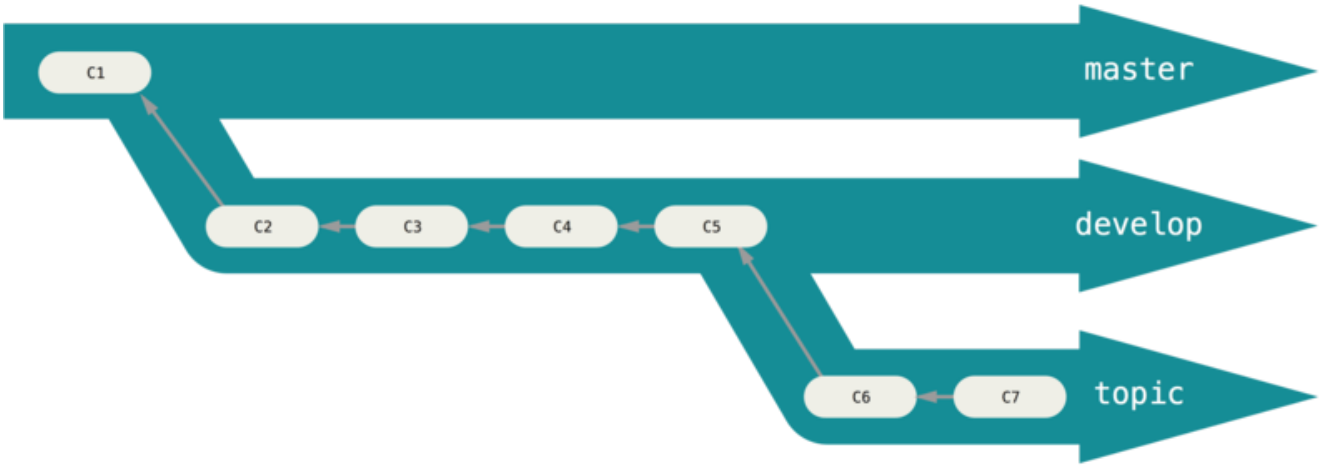


Figure 31. 각 브랜치를 하나의 실험실 로 생각

중요한 개념은 브랜치를 이용해 여러 단계에 걸쳐서 안정화해 나아가면서 충분히 안정화가 됐을 때 안정 브랜치로 Merge 한다는 점이다. 다시 말해서 Long-Running의 브랜치가 여러 개일 필요는 없지만 정말 유용하다는 점이다. 특히 규모가 크고 복잡한 프로젝트일수록 그 유용성이 반짝반짝 빛난다.

4.2. 히스토리 단장하기

Git으로 일하다 보면 어떤 이유로든 로컬 커밋 히스토리를 수정해야 할 때가 있다. 결정을 나중으로 미룰 수 있던 것은 Git의 장점이다. Staging Area로 커밋할 파일을 고르는 일을 커밋하는 순간으로 미룰 수 있고 Stash 명령으로 하던

일을 미룰 수 있다. 게다가 이미 커밋해서 결정한 내용을 수정할 수 있다. 그리고 수정할 수 있는 것도 매우 다양하다. 커밋들의 순서도 변경할 수 있고 커밋 메시지와 커밋한 파일도 변경할 수 있다. 여러 개의 커밋을 하나로 합치거나 반대로 커밋 하나를 여러 개로 분리할 수도 있다. 아니면 커밋 전체를 삭제할 수도 있다. 하지만, 이 모든 것은 다른 사람과 코드를 공유하기 전에 해야 한다.

이 절에서는 사람들과 코드를 공유하기 전에 커밋 히스토리를 예쁘게 단장하는 방법에 대해서 설명한다.

Git이 동작하는 기본 원리 중 하나는 Git은 로컬에 모든 버전관리 데이터를 로컬에 복사(Clone) 해두고 있다는 점이다. 이 때문에 자유롭게 히스토리를 *로컬에서* 수정해 볼 수 있는 자유도 누릴 수 있다. 다만 로컬의 버전관리 데이터 혹은 커밋이 외부로 Push가 된 후라면 이야기는 완전 딴판이된다. Push된 데이터는 수정에 대해선 완전이 끝난 것이다. 고쳐야 할 이유가 생겼더라도 새로 수정작업을 추가해야지 이전 커밋 자체를 수정할 수는 없다. 그렇기에 온전하게 수정 작업을 마무리했다는 확신 없이 작업 내용을 공유하는 저장소로 보내는(Push) 것은 피해야 할 행동이다.

4.2.1. 마지막 커밋을 수정하기

히스토리를 단장하는 일 중에서는 마지막 커밋을 수정하는 것이 가장 자주 하는 일이다. 기본적으로 두 가지로 나눌 수 있는데 하나는 단순히 커밋 메시지만를 수정하는 것이고 다른 하나는 나중에 수정한 파일을 마지막 커밋 안에 밀어넣는 것이다.

커밋 메시지를 수정하는 방법은 매우 간단하다.

```
$ git commit --amend
```

이 명령은 자동으로 텍스트 편집기를 실행시켜서 마지막 커밋 메시지를 열어준다. 여기에 메시지를 바꾸고 편집기를 닫으면 편집기는 바뀐 메시지로 마지막 커밋을 수정한다.

반대로 커밋 메시지가 아니라 프로젝트 내용을 수정한 경우가 있다. 커밋하고 난 후 새로 만든 파일이나 수정한 파일을 가장 최근 커밋에 집어넣을 수 있다. 기본적으로 방법은 같다. 파일을 수정하고 `git add` 명령으로 Staging Area에 넣는다. 그리고 `git commit --amend` 명령으로 커밋하면 커밋 자체가 수정되면서 추가로 수정사항을 밀어넣을 수 있다.

이때 SHA-1 값이 바뀌기 때문에 과거의 커밋을 변경할 때 주의해야 한다.

4.2.2. 커밋 메시지를 여러 개 수정하기

최근 커밋이 아니라 예전 커밋을 수정하려면 다른 도구가 필요하다. 히스토리 수정하기 위해 만들어진 도구는 없지만 `rebase` 명령을 이용하여 수정할 수 있다. 현재 작업하는 브랜치에서 각 커밋을 하나하나 수정하는 것이 아니라 어느 시점부터 HEAD까지의 커밋을 한 번에 Rebase 한다. 대화형 Rebase 도구를 사용하면 커밋을 처리할 때마다 잠시 멈춘다. 그러면 각 커밋의 메시지를 수정하거나 파일을 추가하고 변경하는 등의 일을 진행할 수 있다. `git rebase` 명령에 `-i` 옵션을 추가하면 대화형 모드로 Rebase 할 수 있다. 어떤 시점부터 HEAD까지 Rebase 할 것인지 인자로 넘기면 된다.

마지막 커밋 메시지 세 개를 모두 수정하거나 그 중 몇 개를 수정하는 시나리오를 살펴보자. `git rebase -i` 의 인자로 편집하려는 마지막 커밋의 부모를 `HEAD~2^`나 `HEAD~3`로 해서 넘긴다. 마지막 세 개의 커밋을 수정하는 것이기 때문에 `~3`이 좀 더 기억하기 쉽다. 그렇지만, 실질적으로 가리키게 되는 것은 수정하려는 커밋의 부모인 네 번째 이전 커밋이다.

```
$ git rebase -i HEAD~3
```

이 명령은 Rebase 하는 것이기 때문에 메시지의 수정 여부에 관계없이 `HEAD~3..HEAD` 범위에 있는 모든 커밋을 수정한다. 다시 강조하지만 이미 중앙서버에 Push 한 커밋은 절대 고치지 말아야 한다. Push 한 커밋을 Rebase 하면 결국 같은 내용을 두 번 Push 하는 것이기 때문에 다른 개발자들이 혼란스러워 할 것이다.

실행하면 Git은 수정하려는 커밋 목록이 첨부된 스크립트를 텍스트 편집기로 열어준다.

```
pick f7f3f6d changed my name a bit
pick 310154e updated README formatting and added blame
pick a5f4a0d added cat-file

# Rebase 710f0f8..a5f4a0d onto 710f0f8
#
# Commands:
#  p, pick = use commit
#  r, reword = use commit, but edit the commit message
#  e, edit = use commit, but stop for amending
#  s, squash = use commit, but meld into previous commit
#  f, fixup = like "squash", but discard this commit's log message
#  x, exec = run command (the rest of the line) using shell
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#
# Note that empty commits are commented out
```

이 커밋은 모두 `log` 명령과는 정반대의 순서로 나열된다. 대화형 Rebase는 스크립트에 적혀 있는 순서대로 `HEAD~3`부터 적용하기 시작하고 위에서 아래로 각각의 커밋을 순서대로 수정한다. 순서대로 적용하는 것이기 때문에 제일 위에 있는 것이 최신이 아니라 가장 오래된 것이다.

특정 커밋에서 실행을 멈추게 하려면 스크립트를 수정해야 한다. `pick` 이라는 단어를 'edit'로 수정하면 그 커밋에서 멈춘다. 가장 오래된 커밋 메시지를 수정하려면 아래와 같이 편집한다.

```
edit f7f3f6d changed my name a bit
pick 310154e updated README formatting and added blame
pick a5f4a0d added cat-file
```

저장하고 편집기를 종료하면 Git은 목록에 있는 커밋 중에서 가장 오래된 커밋으로 이동하고, 아래와 같은 메시지를 보여주고, 명령 프롬프트를 보여준다.

```
$ git rebase -i HEAD~3
Stopped at f7f3f6d... changed my name a bit
You can amend the commit now, with

       git commit --amend

Once you're satisfied with your changes, run

       git rebase --continue
```

명령 프롬프트가 나타날 때 Git은 Rebase 과정에서 현재 정확히 뭘 해야 하는지 메시지로 알려준다. 아래와 같은 명령을 실행하고

```
$ git commit --amend
```

커밋 메시지를 수정하고 텍스트 편집기를 종료하고 나서 아래 명령을 실행한다.

```
$ git rebase --continue
```

이렇게 나머지 두 개의 커밋에 적용하면 끝이다. 다른 것도 pick을 edit로 수정해서 이 작업을 몇 번이든 반복할 수 있다. 매번 Git이 멈출 때마다 커밋을 정정할 수 있고 완료할 때까지 계속 할 수 있다.

4.2.3. 커밋 순서 바꾸기

대화형 Rebase 도구로 커밋 전체를 삭제하거나 순서를 조정할 수 있다. `added cat-file` 커밋을 삭제하고 다른 두 커밋의 순서를 변경하려면 아래와 같은 Rebase 스크립트를

```
pick f7f3f6d changed my name a bit
pick 310154e updated README formatting and added blame
pick a5f4a0d added cat-file
```

아래와 같이 수정한다.

```
pick 310154e updated README formatting and added blame
pick f7f3f6d changed my name a bit
```

수정한 내용을 저장하고 편집기를 종료하면 Git은 브랜치를 이 커밋의 부모로 이동시키고서 `310154e` 와 `f7f3f6d` 를 순서대로 적용한다. 명령이 끝나고 나면 커밋 순서가 변경됐고 ``added cat-file'' 커밋이 제거된 것을 확인할 수 있다.

4.2.4. 커밋 합치기

대화형 Rebase 명령을 이용하여 여러 개의 커밋을 꾹꾹 눌러서 커밋 하나로 만들어 버릴 수 있다. Rebase 스크립트에 자동으로 포함된 도움말에 설명이 있다.

```
#
# Commands:
#  p, pick = use commit
#  r, reword = use commit, but edit the commit message
#  e, edit = use commit, but stop for amending
#  s, squash = use commit, but meld into previous commit
#  f, fixup = like "squash", but discard this commit's log message
#  x, exec = run command (the rest of the line) using shell
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#
# Note that empty commits are commented out
```

pick 이나 edit 말고 squash 를 입력하면 Git은 해당 커밋과 바로 이전 커밋을 합칠 것이고 커밋 메시지도 Merge 한다. 그래서 3개의 커밋을 모두 합치려면 스크립트를 아래와 같이 수정한다.

```
pick f7f3f6d changed my name a bit
squash 310154e updated README formatting and added blame
squash a5f4a0d added cat-file
```

저장하고 나서 편집기를 종료하면 Git은 3개의 커밋 메시지를 Merge 할 수 있도록 에디터를 바로 실행해준다.

```
# This is a combination of 3 commits.
# The first commit's message is:
changed my name a bit

# This is the 2nd commit message:

updated README formatting and added blame

# This is the 3rd commit message:

added cat-file
```

이 메시지를 저장하면 3개의 커밋이 모두 합쳐진 커밋 한 개만 남는다.

4.2.5. 커밋 분리하기

커밋을 분리한다는 것은 기존의 커밋을 해제하고(혹은 되돌려 놓고) Stage를 여러 개로 분리하고 나서 그것을 원하는 횟수만큼 다시 커밋하는 것이다. 예로 들었던 커밋 세 개 중에서 가운데 것을 분리해보자. 이 커밋의 updated README formatting and added blame 을 updated README formatting 과 added blame 으로 분리하는 것이다. rebase -i 스크립트에서 해당 커밋을 "edit"로 변경한다.

```
pick f7f3f6d changed my name a bit
edit 310154e updated README formatting and added blame
pick a5f4a0d added cat-file
```

저장하고 나서 명령 프롬프트로 넘어간 다음에 그 커밋을 해제하고 그 내용을 다시 두 개로 나눠서 커밋하면 된다. 저장하고 편집기를 종료하면 Git은 제일 오래된 커밋의 부모로 이동하고서 `f7f3f6d` 과 `310154e` 을 처리하고 콘솔 프롬프트를 보여준다. 여기서 커밋을 해제하는 `git reset HEAD^` 라는 명령으로 커밋을 해제한다. 그러면 수정했던 파일은 Unstaged 상태가 된다. 그다음에 파일을 Stage 한 후 커밋하는 일을 원하는 만큼 반복하고 나서 `git rebase --continue` 라는 명령을 실행하면 남은 Rebase 작업이 끝난다.

```
$ git reset HEAD^
$ git add README
$ git commit -m 'updated README formatting'
$ git add lib/simplegit.rb
$ git commit -m 'added blame'
$ git rebase --continue
```

나머지 `a5f4a0d` 커밋도 처리되면 히스토리는 아래와 같다.

```
$ git log -4 --pretty=format:"%h %s"
1c002dd added cat-file
9b29157 added blame
35cfb2b updated README formatting
f3cc40e changed my name a bit
```

다시 강조하지만 Rebase 하면 목록에 있는 모든 커밋의 SHA-1 값은 변경된다. 절대로 이미 서버에 Push 한 커밋을 수정하면 안 된다.

4.3. 요약

이제 Git 프로젝트 운영하고, 자신의 커밋을 보기 좋게 만드는 법을 배웠다. 이제 다른 사람과 쉽게 협업할 수 있는 Git 개발자가 된 것을 축하한다.