Progit_v2 정리

1장. 시작하기

버전관리 도구란 무엇인가? git이란 또 무엇인가?

1. 버전 관리란(VCS)?

파일 변화를 시간에 따라 기록했다가 나중에 특정 시점의 버전을 다시 꺼내올 수 있는 시스템이다.

- 로컬 버전 관리 : 버전을 관리하기 위해 디렉토리로 파일을 복사하는 방법
- 중앙집중식 버전 관리(CVCS) : 누가 무엇을 하고 있는지 알 수 있고 관리자는 누가 무엇을 하지 꼼꼼하게 관리 가능 그러나 중앙서버에 문제가 생긴다면 그 동안 서버를 사용 할 수 없고 백업도 할 수 없다. 중앙서버의 하드디스크에 문제가 생기면 모든 히스토리를 잃는다.
- 분산 버전 관리 시스템(DVCS) : 그냥 저장소에 전부 복제한다. 서버에 문제가 생겨도 이 복제물로 다시 시작할 수 있다.

2. Git 이란 무엇인가?

다른 VCS(파일들의 목록을 관리) 하지만 Git은 시간순으로 관리하면서 파일들의 집합을 관리

Git은 커밋하거나 프로젝트의 상태를 저장할 때마다 파일이 존재하는 순간을 중요하게 여김

- 거의 모든 명령을 로컬에서 실행 : 오프라인에서 작업하고 업로드 가능
- Git의 무결성 : 데이터를 저장하기 전에 항상 체크섬[1]을 구하고 그 체크섬으로 데이터를 관리한다
- 데이터를 추가할 뿐 : Git으로 무얼 하든 Git DB에 데이터가 추가됨, 되돌리거나 삭제할 방법이 없다.

[1] Commit 마다 생성 저장소에서 이 40자리 이름을 보고 각 커밋을 구분하고 선택함

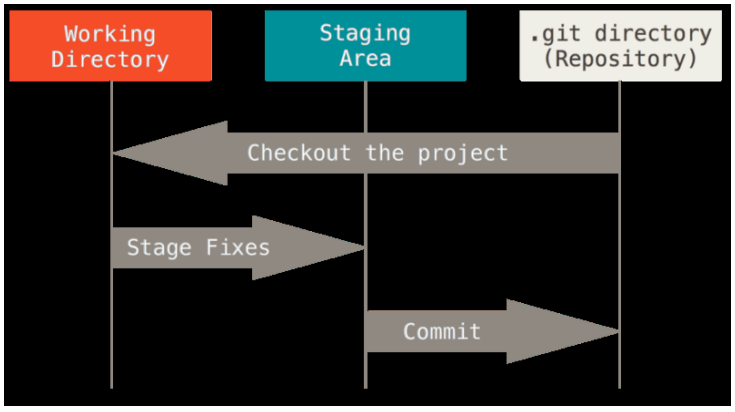
3. Git Directory & Working Tree & Staging Area (세가지 상태)

Git 파일은 Committed(데이터가 로컬 DB에 안전하게 저장 됨),Modified(수정한 파일을 아직 로컬 DB에 커밋하지 않은 것) Staged(현재 수정한 파일을 곧 커밋할 것)

- Git Directory : Git이 프로젝트의 메타데이터와 객체 DB를 저장하는 곳을 말한다. 이 Git 디렉토리가 Git의 핵심이며 다른 컴퓨터에 있는 저장소를 Clone 할 때 Git 디렉토리가 만들어 진다.
- Working Tree : 프로젝트의 특정 버전을 Checkout[2]한 것이다. Git 디렉토리는 지금 작업하는 디스크에 있고 그 디렉토리 안에 압축된 DB에서 파일을 가져와 워킹 트리를 만든다.
- Staginig Area : Git 디렉토리에 있고, 단순한 파일이고 곧 커밋할 파일에 대한 정보를 저장한다. 종종 'Index'라고도 불림

Git에서 하는 일은 기본적으로 아래와 같다.

1. 워킹 트리에서 파일을 수정한다.
2. Staging Area에 파일을 Stage 해서 커밋할 스냅샷을 만든다.
3. Staging Area에 있는 파일들을 커밋해서 Git 디렉토리에 영구적인 스냅샷으로 저장



Git 디렉토리에 있는 파일들은 Committed 상태, 파일을 수정하고 Staging Area에 추가했다면 Staged 상태, 그리고 Checkout 하고 나서 수정했지만, 아직 Staging Area에 추가하지 않았다면

[2] Checkout : 클라이언트가 중앙 서버에서 파일을 받아서 사용

Modified 상태

2장. Git의 기초

저장소를 만들고 설정하는 방법, 파일을 추적(Track)하거나 추적을 그만두는 방법, 변경 내용을 Stage 하고 커밋 하는 방법 등이 존재한다.

1. Git 저장소 만들기

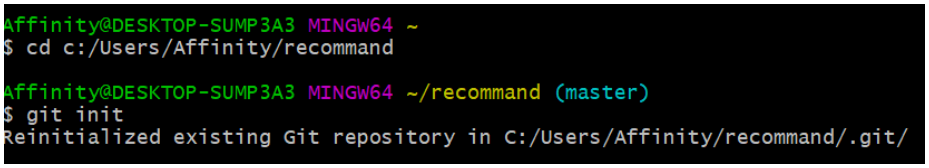
기존 프로젝트를 Git으로 관리하고 싶을 때, 프로젝트의 디렉토리로 이동한다.

Windows:

```
$ cd /c/user/your_repository
```

그리고 아래와 같은 명령을 실행한다:

```
$ git init
```



git init은 .git이라는 하위 디렉토리를 만든다. .git 디렉토리에는 저장소에 필요한 뼈대 파일(Skeleton)이 들어 있다. 이 명령만으로는 아직 프로젝트의 어떤 파일도 관리하지 않는다. Git이 파일을 관리하게 하려면 저장소에 파일을 추가하고 커밋해야 한다.

```
$ git add *.c
$ git add LICENSE
$ git commit -m 'initial project version'
```

명령어 몇 개로 순식간에 Git 저장소를 만들고 파일 버전 관리를 시작했다.

2. 기존 저장소를 Clone 하기

다른 프로젝트에 **참여하거나(Contribute) Git 저장소를 복사하고 싶을 때** git clone 명령을 사용한다. Git과 Subversion과 다른 가장 큰 차이점은 **서버에 있는 거의 모든 데이터를 복사**한

다는 것이다. Git clone을 실행하면 프로젝트의 히스토리를 전부 받아온다.

```
$ git clone https://github.com/libgit2/libgit2
```
[3]

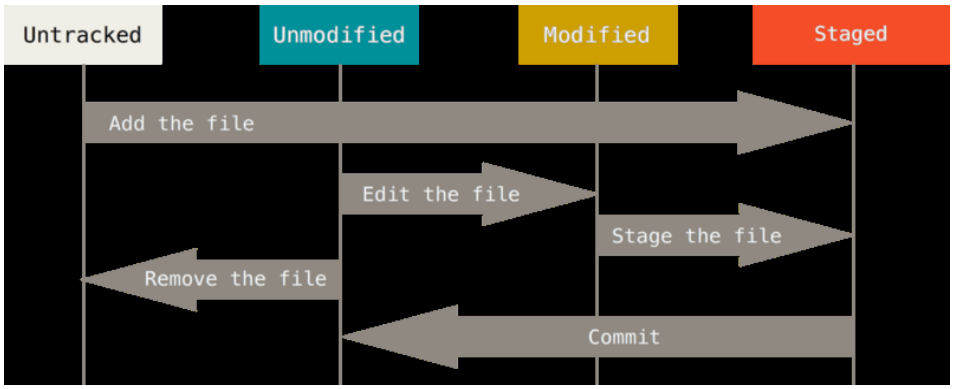
git clone [url] 명령으로 저장소를 Clone 한다.

3. 수정하고 저장소에 저장하기

파일을 수정하다가 저장하고 싶으면 스냅샷을 커밋한다.

워킹 디렉토리의 모든 파일은 크게 Tracked(관리대상임)와 Untracked(관리대상이 아님)로 나눈다. Tracked 파일은 이미 스냅샷에 포함돼 있던 파일이다. Tracked 파일은 Unmodified(수정하지 않음)와 Modified(수정함)그리고 Staged(커밋으로 저장소에 기록할)상태 중 하나이다. 그리고 나머지 파일은 모두 Untracked 파일이다. Untracked 파일은 워킹 디렉토리에 있는 파일 중 스냅샷에도 Staging Area에도 포함되지 않은 파일이다. 처음 저장소를 Clone 하면 모든 파일은 Tracked이면서 Unmodified 상태이다. 파일을 Checkout 하고 나서 아무것도 수정하지 않았기 때문에 그렇다.

마지막 커밋 이후 아직 아무것도 수정하지 않은 상태에서 어떤 파일을 수정하면 Git은 그 파일을 Modified 상태로 인식한다. 실제로 커밋을 하기 위해서는 이 수정한 파일을 Staged 상태로 만들고, Staged 상태의 파일을 커밋한다. 이런 라이프사이클을 계속 반복한다.



[3] Libgit2 라이브러리 소스코드를 Clone하는 코드

4. 파일의 상태 확인하기

파일의 상태를 확인하려면 git status[4] 명령을 사용한다.

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working directory clean
```

위의 내용은 파일을 하나도 수정하지 않았다는 것이다. Tracked 파일은 하나도 수정되지 않았다는 의미다.

5. 파일을 새로 추적하기

git add 명령으로 파일을 새로 추적할 수 있다.

```
$ 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
```

"Changes to be committed"에 들어 있는 파일은 Staged상태라는 것을 의미한다. 커밋하면 git add를 실행한 시점의 파일이 커밋되어 저장소 히스토리에 남는다. git add 명령은 파일 또는 디렉토리의 경로를 매개변수로 받는다. 디렉토리면 모든 파일들까지 재귀적으로 추가한다.

6. Modified 상태의 파일을 Stage 하기

[4] git status –s 또는 –short 처럼 옵션을 주면 짤막하게 보여준다.

```
$ 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

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:   CONTRIBUTING.md
```

'Changes not staged for commit"인 CONTRIBUTING.md는Tracked 상태지만 아직 Staged한 상태는 아니라는 것이다. Staged 상태는 git add를 통해서 한다. Add의 의미는 프로젝트에 다음 커밋에 추가한다고 받아 들이자.

7. 파일 무시하기

Git이 관리할 필요가 없는 파일(로그,빌드 시스템 파일)을 무시할려면 .gitignore 파일을 만들고 그 안에 무시할 패턴을 적는다. gitignore 파일은 보통 처음에 만들어 두는 것이 좋다. Git 저장소에 커밋하고 싶지 않는 파일을 실수로 커밋하는 일을 방지할 수 있기 때문이다.

```
$ cat .gitignore
*.[oa]
*~[5]
```

.[oa]는 확장자가 ".o" 나 ".a"인 파일을 Git이 무시하라는 것, *~는 ~로 끝나는 모든 파일을 무시하라는 것이다.

- gitignore 파일에 입력하는 패턴 규칙

  1. 아무것도 없는 라인이나, `#`로 시작하는 라인은 무시한다.
  2. 표준 Glob 패턴을 사용[6]한다.
  3. 슬래시(/)로 시작하면 하위 디렉토리에 적용되지(Recursivity) 않는다.
  4. 디렉토리는 슬래시(/)를 끝에 사용하는 것으로 표현한다.

---

[5] .gitignore 파일의 예시

[6] 정규 표현식을 단순하게 만든 것, 쉘에서 많이 사용함.
애스터리스크(*)는 문자가 하나도 없거나 하나 이상을 의미하고, [abc] 는 중괄호 안에 있는 문자 중 하나를 의미한다(그러니까 이 경우에는 a, b,c). 물음표(?)는 문자 하나를 말하고, [0-9] 처럼 중괄호 안의 캐릭터 사이에 하이픈(-)을 사용하면 그 캐릭터 사이에 있는 문자 하나를 말한다. 애스터리스크 2개를 사용하여 디렉토리 안의 디렉토리 까지 지정할 수 있다. a/**/z 패턴은 a/z, a/b/z, a/b/c/z 디렉토리에 사용할 수 있다.

5. 느낌표(!)로 시작하는 패턴의 파일은 무시하지 않는다.

8. Staged와 Unstaged 상태의 변경 내용을 보기

어떤 내용이 변경됐는지 살펴볼려면 `git diff`[7]를 사용해야 한다. Patch 처럼 어떤 라인을 추가하고 삭제했는지가 궁금할 때 사용한다. `git diff`명령어는 마지막으로 커밋한 후에 수정한 것들을 전부를 보여주지 않는다. `git diff`는 Unstaged 상태인 것들만 보여준다.

`CONTRIBUTING.md` 파일을 Stage 한 후에 다시 수정해도 `git diff` 명령을 사용할 수 있다. 이때는 Staged 상태인 것과 Unstaged 상태인 것을 비교한다.

```
$ git add CONTRIBUTING.md
$ echo '# test line' >> 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)

    modified:   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:   CONTRIBUTING.md
```

`git diff` 명령으로 Unstaged 상태인 변경 부분을 확인할 수 있다.

9. 변경사항 커밋하기

Unstaged 상태의 파일은 커밋되지 않으므로 Staging Area에 파일을 정리해야 한다. Git은 생성하거나 수정하고 나서 git add명령으로 추가하지 않은 파일은 커밋하지 않는다. Modified상태로 남아 있다. Staged 상태인지 확인하고 git commit을 실행하여 커밋한다.

```
$ git commit -m "Story 182: Fix benchmarks for speed"
[master 463dc4f] Story 182: Fix benchmarks for speed
 2 files changed, 2 insertions(+)
 create mode 100644 README
```

[7] Git status는 파일이 변경됐다는 사실을 확인하는 용도

(master)브랜치에 커밋했고 체크섬은 (463dc4f)라는 것이며 파일은 2개가 변경됐고 2라인이 추가 되었다는 것을 알려준다.

Git은 Stagning Area에 속한 스냅샷을 커밋한다는 것을 기억해야 한다. 커밋할 때마다 프로젝트의 스냅샷을 기록하기 때문에 나중에 스냅샷끼리 비교하거나 예전 스냅샷으로 되돌릴 수 있다.

10. Staging Area 생략하기

유용하지만 복잡하기만 하고 필요하지 않을 때도 있는 Staging Area는 `git commit` 명령을 실행할 때 `-a` 옵션을 추가하면 Git은 Tracked 상태의 파일을 자동으로 Staging Area에 넣는다. 그래서 `git add` 명령을 실행하는 수고를 덜 수 있다.

```
$ git status
On branch master
Your branch is up-to-date with 'origin/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:   CONTRIBUTING.md

no changes added to commit (use "git add" and/or "git commit -a")
$ git commit -a -m 'added new benchmarks'
[master 83e38c7] added new benchmarks
 1 file changed, 5 insertions(+), 0 deletions(-)
```

커밋하기 전에 `git add` 명령으로 `CONTRIBUTING.md` 파일을 추가하지 않았다는 점, -a 옵션을 사용하면 모든 파일이 자동으로 추가된다.

11. 파일 삭제하기

Git에서 파일을 제거하려면 git rm 명령으로 Tracked 상태의 파일을 삭제한 후에(정확하게는 Staging Area에서 삭제하는 것) 커밋해야 한다. 이 명령은 워킹 디렉토리에 있는 파일도 삭제하기 때문에 실제로 파일도 지워진다. Git 명령을 사용하지 않고 단순히 워킹 디렉터리에서 파일을 삭제하고 git status 명령으로 상태를 확인하면 Git은 현재"Changes not staged for commit"(즉, Unstaged 상태)라고 표시해준다.

```
$ rm grit.gemspec
$ 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:    grit.gemspec

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

git rm 을 실행하면 삭제한 파일은 Staged 상태가 된다.

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

    deleted:    grit.gemspec
```

커밋하면 파일은 삭제되고 Git은 이 파일을 더는 추적하지 않는다. 이미 파일을 수정했거나 Staging Area에 추가했다면 –f 옵션을 주어 강제로 삭제해야 한다. 이 점은 실수로 데이터를 삭제하지 못하도록 하는 안전장치다. 커밋 하지 않고 수정한 데이터는 Git으로 복구할 수 없기 때문이다.

또 Staging Area에서만 제거하고 워킹 디렉토리에 있는 파일은 지우지 않고 남겨둘 수 있다. 다시 말해 하드디스크에 있는 파일은 그대로 두고 Git만 추적하지 않게 한다.
이것은 .gitignore파일에 추가하는 것을 빼먹었거나 대용량 로그 파일이나 컴파일된 파일인 .a파일 같은 것을 실수로 추가했을 때 쓴다. –cached 옵션을 사용하여 명령을 실행한다.

```
$ git rm --cached README
```

여러 개의 파일이나 디렉토리를 한꺼번에 삭제할 수도 있다. 아래와 같이 git rm 명령에 File-glob 패턴을 사용한다.

```
$ git rm log/\*.log
```

* 앞에 ₩ [8]을 사용한 것을 기억하자. 파일명 확장 기능은 쉘에만 있는 것이 아니라 Git 자체에도 있기 때문에 필요하다. 이 명령은 log/ 디렉토리에 있는 .log 파일을 모두 삭제한다. 아래의 예제처럼 할 수도 있다

```
$ git rm \*~
```

.

[8] 역슬레시

이 명령은 ~로 끝나는 파일을 모두 삭제한다.

12. 파일 이름 변경하기

Git은 파일 이름이 변경됐다는 별도의 정보를 저장하지 않는다. 하지만 Git은 알고 있다.

```
$ git mv file_from file_to
```

git mv : 파일이름 변경

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

git mv는 일종의 단축 명령어로, 위에 명령을 실행해준 것 뿐이다. 중요한 것은 이름을 변경하고 나서 꼭 rm/add 명령을 실행해야 한다.

13. 커밋 히스토리 조회하기

새로 저장소를 만들어 몇 번 커밋을 했을 수도 있고, 커밋 히스토리가 있는 저장소를 Clone 했을 수도 있다. 저장소의 히스토리를 보고 싶을 때는 git log를 사용한다.

- git log –p : 각 커밋의 diff결과를 보여준다.
- git log –2 : 최근 두 개의 결과만 보여주는 옵션
- git log --stat :각 커밋의 통계 정보를 알 수 있음 , 어떤 파일이 수정됐는지, 얼마나 많은 파일이 변경됐는지, 또 얼마나 많은 라인을 추가하거나 삭제했는지 보여준다.
- git --pretty[9] : 이 옵션을 통해 히스토리 내용을 보여줄 때 기본 형식 이외에 여러 가지 중에 하나를 선택할 수 있다.

  ```
  $ git log --pretty=format:"%h - %an, %ar : %s"
  ca82a6d - Scott Chacon, 6 years ago : changed the version number
  085bb3b - Scott Chacon, 6 years ago : removed unnecessary test
  a11bef0 - Scott Chacon, 6 years ago : first commit
  ```

- git --format : 나만의 포맷으로 결과를 출력하고 싶을 때 사용한다.

[9] --pretty=online : 각 커밋을 한 라인으로 보여준다(많은 커밋을 한 번에 조회할 때 유용),--pretty=short,full,fuller 옵션이 존재하며 정보를 조금씩 가감해서 보여준다.

`git log --pretty=format` 에 쓸 몇가지 유용한 옵션` 포맷에서 사용하는 유용한 옵션.

Table 1. `git log --pretty=format` 에 쓸 몇가지 유용한 옵션`

옵션	설명
%H	커밋 해시
%h	짧은 길이 커밋 해시
%T	트리 해시
%t	짧은 길이 트리 해시
%P	부모 해시
%p	짧은 길이 부모 해시
%an	저자 이름
%ae	저자 메일
%ad	저자 시각 (형식은 –date= 옵션 참고)
%ar	저자 상대적 시각
%cn	커미터 이름
%ce	커미터 메일
%cd	커미터 시각
%cr	커미터 상대적 시각
%s	요약

저자(Author) 와 커미터(Committer) 를 구분하는 것이 조금 이상해 보일 수 있다. 저자는 원래 작업을 수행한 원작자이고 커밋터는 마지막으로 이 작업을 적용한(저장소에 포함시킨) 사람이다. 만약 당신이 어떤 프로젝트에 패치를 보냈고 그 프로젝트의 담당자가 패치를 적용했다면 두 명의 정보를 모두 알 필요가 있다. 그래서 이 경우 당신이 저자고 그 담당자가 커미터다.

git log 명령의 기본적인 옵션과 출력물의 형식에 관련된 옵션을 살펴보았다. git log 명령은 앞서 살펴본 것보다 더 많은 옵션을 지원한다. git log 주요 옵션 는 지금 설명한 것과 함께 유용하게 사용할 수 있는 옵션이다. 각 옵션으로 어떻게 log 명령을 제어할 수 있는지 보여준다.

Table 2. git log 주요 옵션

옵션	설명
-p	각 커밋에 적용된 패치를 보여준다.
--stat	각 커밋에서 수정된 파일의 통계정보를 보여준다.
--shortstat	--stat 명령의 결과 중에서 수정한 파일, 추가된 라인, 삭제된 라인만 보여준다.
--name-only	커밋 정보중에서 수정된 파일의 목록만 보여준다.
--name-status	수정된 파일의 목록을 보여줄 뿐만 아니라 파일을 추가한 것인지, 수정한 것인지, 삭제한 것인지도 보여준다.
--abbrev-commit	40자 짜리 SHA-1 체크섬을 전부 보여주는 것이 아니라 처음 몇 자만 보여준다.
--relative-date	정확한 시간을 보여주는 것이 아니라 ‘`2 weeks ago”처럼 상대적인 형식으로 보여준다.
--graph	브랜치와 머지 히스토리 정보까지 아스키 그래프로 보여준다.
--pretty	지정한 형식으로 보여준다. 이 옵션에는 oneline, short, full, fuller, format이 있다. format은 원하는 형식으로 출력하고자 할 때 사용한다.

14. 조회 제한조건

출력 형식과 관련된 옵션을 살펴봤지만 git log 명령은 조회 범위를 제한하는 옵션들도 있다. 히스토리 전부가 아니라 부분만 조회한다. 이미 최근 두 개만 조회하는 -2[10] 옵션은 살펴봤다. Git은 기본적으로 출력을 pager류의 프로그램을 거쳐서 내보내므로 한 번에 한 페이지씩 보여준다.

반면 --since 나 --until 같은 시간을 기준으로 조회하는 옵션은 매우 유용하다. 지난 2주 동안 만들어진 커밋 들만 조회하는 명령은 아래와 같다.

```
$ git log --since=2.weeks
```

이 옵션은 다양한 형식을 지원한다. "2008-01-15" 같이 정확한 날짜도 사용할 수 있고 "2 years 1 day 3 minutes ago" 같이 상대적인 기간을 사용할 수도 있다.
또 다른 기준도 있다. --author 옵션으로 저자를 지정하여 검색할 수도 있고 --grep 옵션으로 커밋 메시지에서 키워드를 검색할 수도 있다.[11]
진짜 유용한 옵션으로 -S 가 있는데 이 옵션은 코드에서 추가되거나 제거된 내용 중에 특정 텍스트가 포함되어 있는지를 검색한다. 예를 들어 어떤 함수가 추가되거나 제거된 커밋만을 찾아보려면 아래와 같은 명령을 사용한다.

```
$ git log --Sfunction_name
```

마지막으로 파일 경로로 검색하는 옵션이 있는데 이것도 정말 유용하다. 디렉토리나 파일 이름을 사용하여 그 파일이 변경된 log의 결과를 검색할 수 있다. 이 옵션은 -- 와 함께 경로 이름을 사용하는데 명령어 끝 부분에 쓴다.[12] git log 조회 범위를 제한하는 옵션은 조회 범위를 제한하는 옵션들이다.

Table 3. git log 조회 범위를 제한하는 옵션

옵션	설명
-(n)	최근 n 개의 커밋만 조회한다.
--since, --after	명시한 날짜 이후의 커밋만 검색한다.
--until, --before	명시한 날짜 이전의 커밋만 조회한다.
--author	입력한 저자의 커밋만 보여준다.
--committer	입력한 커미터의 커밋만 보여준다.
--grep	커밋 메시지 안의 텍스트를 검색한다.
-S	커밋 변경(추가/삭제) 내용 안의 텍스트를 검색한다.

[10] 실제 사용법은 `-<n>`이고 n은 최근 n개의 커밋을 의미한다.

[11] author와 grep 옵션을 함께 사용하여 모두 만족하는 커밋을 찾으려면 --all-match옵션도 반드시 함께 사용해야 한다

[12] git log -- path1 path2

15. 되돌리기

Git을 사용하면 실수는 대부분 복구할 수 있지만 되돌린 것은 복구할 수 없다.

종종 완료한 커밋을 수정해야 할때는 --amend 옵션을 사용한다.

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

이 명령은 Staging Area를 사용하여 커밋한다. 만약 마지막으로 커밋하고 나서 수정한 것이 없다면(커밋하자마자 바로 이 명령을 실행하는 경우) 조금 전에 한 커밋과 모든 것이 같다. 이때는 커밋 메시지만 수정한다. 편집기가 실행되면 이전 커밋 메시지가 자동으로 포함된다. 메시지를 수정하지 않고 그대로 커밋해도 기존의 커밋을 덮어쓴다.

커밋을 했는데 Stage 하는 것을 깜빡하고 빠트린 파일이 있으면 아래와 같이 고칠 수 있다.

```
$ git commit -m 'initial commit'
$ git add forgotten_file
$ git commit --amend
```

실행한 명령어 3개는 모두 커밋 한 개로 기록된다. 두 번째 커밋은 첫 번째 커밋을 덮어쓴다.

16. 파일 상태를 Unstage로 변경하기

다음은 Staging Area와 워킹 디렉토리 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. 두 파일 모두 Staging Area에 들어 있다. 이제 둘 중 하나를 어떻게 꺼낼까?

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

    renamed:    README.md -> README
    modified:   CONTRIBUTING.md
```

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

    renamed:    README.md -> 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:   CONTRIBUTING.md
```

17. Modified 파일 되돌리기

어떻게 해야 CONTRIBUTING.md 파일을 수정하고 나서 다시 되돌릴 수 있을까? 그러니까 최근 커밋된 버전으로(아니면 처음 Clone 했을 때처럼 워킹 디렉토리에 처음 Checkout 한 그 내용으로) 되돌리는 방법이 무얼까?

```
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:   CONTRIBUTING.md
```

위의 메시지는 수정한 파일을 되돌리는 방법을 알려준다.

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

    renamed:    README.md -> README
```

정상적으로 복원된 것을 알 수 있다.

IMPORTANT `git checkout -- [file]` 명령은 꽤 위험한 명령이라는 것을 알아야 한다. 원래 파일로 덮어썼기 때문에 수정한 내용은 전부 사라진다. 수정한 내용이 진짜 마음에 들지 않을 때만 사용하자.

리모트 저장소

리모트 저장소는 인터넷이나 네트워크 어딘가에 있는 저장소를 말한다. 저장소는 여러 개가 있을 수 있는데, 읽고 쓰기 모두 가능한 저장소와 읽기만 가능한 저장소가 있다. 다른 사람들과 함께 일한다는 것은 리모트 저장소를 관리하면서 데이터를 거기에 Push 하고 Pull 하는 것이다. 리모트 저장소를 관리 한다는 것은 저장소를 추가,삭제하는 것뿐만 아니라 브랜츠를 관리하고 추적할지 말지 등을 관리하는 것을 말한다.

18. 리모트 저장소 확인하기

git remote 명령으로 현재 프로젝트에 등록된 리모트 저장소를 확인할 수 있다. 저장소를 Clone 하면 origin이라는 리모트 저장소가 자동으로 등록된다.

```
$ 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
```

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

-v 옵션을 주어 단축이름과 URL을 함께 볼 수 있다.

19. 리모트 저장소 추가하기

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)
```

URL 대신 pb라는 이름을 사용할 수 있다. 예를 들어 로컬 저장소에는 없지만 Paul의 저장소에 있는 것을 가져오려면

```
$ 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 로컬 브랜치중 하나에 Merge 하거나 Checkout 해서 브랜치 내용을 자세히 확인할 수 있다.

20. 리모트 저장소를 Pull 하거나 Fetch 하기

```
$ git fetch [remote-name]
```

이 명령은 로컬에는 없지만, 리모트 저장소에는 있는 데이터를 모두 가져온다. 그러면 리모트 저장소의 모든 브랜치를 로컬에서 ㅈ버근할 수 있어서 언제든지 Merge를 하거나 내용을 살펴 볼 수 있다.

저장소를 Clone 하면 명령은 자동으로 리모트 저장소를 'origin'이라는 이름으로 추가한다. 그래서 나중에 git fetch origin 명령을 실행하면 Clone 한 이후에(혹은 마지막으로 가져온 이후에) 수정된 것을 모두 가져온다. git fetch 명령은 리모트 저장소의 데이터를 모두 로컬로 가져오지만, 자동으로 Merge 하지 않는다. 그래서 로컬에서 하던 작업을 정리하고 나서 수동으로 Merge 해야 한다.[13]

---

[13] git pull 명령으로 리모트 저장소 브랜치에서 데이터를 가져올 뿐만 아니라 자동으로 로컬 브랜치와 Merge 시킬 수 있다.

21. 리모트 저장소에 Push 하기

프로젝트를 공유하고 싶을 때 Upstream 저장소에 Push 할 수 있다. 이 명령은 git push [리모트 저장소 이름] [브랜치 이름]`으로 단순하다.(((git commands, push))) master 브랜치를 origin 서버에 Push하려면 아래와 같이 서버에 Push 한다.

```
$ git push origin master
```

이 명령은 Clone 한 리모트 저장소에 쓰기 권한이 있고, Clone 하고 난 이후 아무도 Upstream 저장소에 Push 하지 않았을 때만 사용할 수 있다. 다시 말해서 Clone 한 사람이 여러 명 있을 때, 다른 사람이 Push 한 후에 Push 하려고 하면 Push 할 수 없다. 먼저 다른 사람이 작업한 것을 가져와서 Merge 한 후에 Push 할 수 있다.

22. 리모트 저장소 살펴보기

Git remote show [리모트 저장소 이름]명령으로 리모트 저장소의 구체적인 정보를 확인 할 수 있다.

```
$ git remote show origin
* remote origin
  Fetch URL: https://github.com/schacon/ticgit
  Push  URL: https://github.com/schacon/ticgit
  HEAD branch: master
  Remote branches:
    master                               tracked
    dev-branch                           tracked
  Local branch configured for 'git pull':
    master merges with remote master
  Local ref configured for 'git push':
    master pushes to master (up to date)
```

리모트 저장소의 URL과 추적하는 브랜치를 출력한다. 이 명령은 git pull 명령을 실행할 때 master 브랜치와 Merge할 브랜치가 무엇인지 보여 준다. git pull 명령은 리모트 저장소 브랜치의 데이터를 모두 가져오고 나서 자동으로 Merge 할 것이다. 그리고 가져온 모든 리모트 저장소 정보도 출력한다.

23. 리모트 저장소 이름을 바꾸거나 리모트 저장소를 삭제하기

git remote rename 명령으로 리모트 저장소의 이름을 변경할 수 있다.

```
$ git remote rename pb paul
$ git remote
origin
paul
```

로컬에서 관리하던 리모트 저장소의 브랜치 이름도 바뀐다. pb/master에서 paul/master라고 사용해야 한다.

리모트 저장소를 삭제해야 한다면 git remote remove 또는 rm 명령을 사용한다. 서버 정보가 바뀌었을 때(별도의 미러가 필요하지 않을 때, 기여자가 활동하지 않을 때)필요하다.

```
$ git remote remove paul
$ git remote
origin
```

태그

Git에서의 태그는 보통 릴리즈 할 때 사용한다.(V1.0, 등등) 이번에는 태그를 조회하고 생성하는 법, 태그의 종류를 알아본다.

24. 태그 조회하기

git tag : 만들어져 있는 태그가 있는지 확인

```
$ git tag
v0.1
v1.3
```

검색 패턴을 사용하여 태그를 검색할 수 있다. 1.8.5버전의 태그들만 검색하고 싶으면 아래와 같이 실행하자.

```
$ git tag -l "v1.8.5*"
v1.8.5
v1.8.5-rc0
v1.8.5-rc1
v1.8.5-rc2
v1.8.5-rc3
v1.8.5.1
v1.8.5.2
v1.8.5.3
v1.8.5.4
v1.8.5.5
```

25. 태그 붙이기

Git의 태그는 Lightweight 태그와 Annotated 태그로 두 종류가 있다.

Lightweight 태그[14] : 브랜치와 비슷하지만 브랜치처럼 가리키는 지점을 최신 커밋으로 이동시키지 않고 단순히 특정 커밋에 대한 포인터일 뿐이다.

Annotated 태그[15] : Git DB에 태그를 만든사람의 이름, 이메일과 태그를 만든 날짜, 태그 메시지도 저장한다.

---

[14] 임시로 생성하는 태그이거나 정보를 유지할 필요가 없는 경우 사용

[15] 일반적으로 이 태그로 모든 정보를 사용 할 수 있도록 하는 것이 좋다.

26. Annotated 태그

Annotated 태그를 만드는 방법은 간단하며 tag 명령을 실행할 때 –a 옵션을 추가

```
$ git tag -a v1.4 -m "my version 1.4"
$ git tag
v0.1
v1.3
v1.4
```

-m 옵션으로 태그를 저장할 때 메시지를 함께 저장 할 수 있다.

git show명령으로 태그 정보와 커밋 정보를 모두 확인할 수 있다.

```
$ git show v1.4
tag v1.4
Tagger: Ben Straub <ben@straub.cc>
Date:   Sat May 3 20:19:12 2014 -0700

my version 1.4

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

    changed the version number
```

커밋 정보를 보여주기 전에 먼저 태그를 만든 사람이 누구인지, 언제 태그를 만들었는지, 그리고 태그 메시지가 무엇인지 보여준다.

27. Lightweight 태그

Lightweight 택스트 파일에 커밋 체크섬을 저장하는 것뿐, 다른 정보를 저장하지 않는다. –a, -s,-m 옵션을 사용하지 않는다.

```
$ git tag v1.4-lw
$ git tag
v0.1
v1.3
v1.4
v1.4-lw
v1.5
```

이 태그에 git show를 실행하면 별도의 태그 정보는 없고 단순히 커밋 정보만 보여준다.

28. 나중에 태그하기

예전 커밋에 대해서도 태그할 수 있다. 커밋 히스토리는 아래와 같다고 가정한다.

```
$ git log --pretty=oneline
15027957951b64cf874c3557a0f3547bd83b3ff6 Merge branch 'experiment'
a6b4c97498bd301d84096da251c98a07c7723e65 beginning write support
0d52aaab4479697da7686c15f77a3d64d9165190 one more thing
6d52a271eda8725415634dd79daabbc4d9b6008e Merge branch 'experiment'
0b7434d86859cc7b8c3d5e1dddfed66ff742fcbc added a commit function
4682c3261057305bdd616e23b64b0857d832627b added a todo file
166ae0c4d3f420721acbb115cc33848dfcc2121a started write support
9fceb02d0ae598e95dc970b74767f19372d61af8 updated rakefile
964f16d36dfccde844893cac5b347e7b3d44abbc commit the todo
8a5cbc430f1a9c3d00faaeffd07798508422908a updated readme
```

"updated rakefile" 커밋을 v1.2로 태그하지 못했다고 해도 나중에 태그를 붙일 수 있다. 특정 커밋에 태그하기 위해서 명령의 끝에 커밋 체크섬을 명시한다.

```
$ git tag -a v1.2 9fceb02
```

29. 태그 공유하기

Git push 명령은 자동으로 리모트 서버에 태그를 전송하지 않는다. 태그는 별도로 서버에 Push 해야 한다. 브랜치를 공유하는 것과 같은 방법으로 할 수 있다. 'git push origin [태그 이름]'을 실행한다.

```
$ git push origin v1.5
Counting objects: 14, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (12/12), done.
Writing objects: 100% (14/14), 2.05 KiB | 0 bytes/s, done.
Total 14 (delta 3), reused 0 (delta 0)
To git@github.com:schacon/simplegit.git
 * [new tag]         v1.5 -> v1.5
```

한 번에 태그를 여러 개 Push 하고 싶으면 --tags 옵션을 추가하여 git push를 실행한다. 이 명령으로 리모트 서버에 없는 태그를 모두 전송할 수 있다.

```
$ git push origin --tags
Counting objects: 1, done.
Writing objects: 100% (1/1), 160 bytes | 0 bytes/s, done.
Total 1 (delta 0), reused 0 (delta 0)
To git@github.com:schacon/simplegit.git
 * [new tag]         v1.4 -> v1.4
 * [new tag]         v1.4-lw -> v1.4-lw
```

이제 누군가 저장소에서 Clone 하거나 Pull을 하면 모든 태그 정보도 함께 전송된다.

30. 태그를 Checkout 하기

태그는 브랜치와는 달리 가리키는 커밋을 바꿀 수 없는 이름이기 때문에 Checkout 해서 사용할 수 없다. 태그가 가리키는 특정 커밋 기반의 브랜치를 만들어 작업하려면 아래와 같이 새로 브랜치를 생성한다.

```
$ git checkout -b version2 v2.0.0
Switched to a new branch 'version2'
```

이렇게 브랜치를 만든 후에 version2 브랜치에 커밋하면 브랜치는 업데이트 되지만, v2.0.0태그는 가리키는 커밋이 변하지 않았으므로 두 내용이 가리키는 커밋이 다르다는 것을 알 수 있다.

Git Alias

명령을 완벽하게 입력하지 않으면 Git은 알아듣지 못한다. Git의 명령을 전부 입력하는 것이 귀찮다면 git config를 사용하여 각 명령의 Alias을 쉽게 만들 수 있다.

```
$ git config --global alias.co checkout
$ git config --global alias.br branch
$ git config --global alias.ci commit
$ git config --global alias.st status
```

git commit 대신 git ci 만으로 커밋할 수 있다.

```
$ git config --global alias.visual '!gitk'
```

이것으로 쉽게 새로운 명령을 만들 수 있다. 그리고 Git의 명령어뿐만 아니라 외부 명령어도 실행할 수 있다. ! 를 제일 앞에 추가하면 외부 명령을 실행한다. 커스텀 스크립트를 만들어서 사용할 때 매우 유용하다. 아래 명령은 git visual이라고 입력하면 gitk 가 실행된다.

3장.Git 브랜치

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.

1. 브랜치란 무엇인가

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.

```
$ 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.

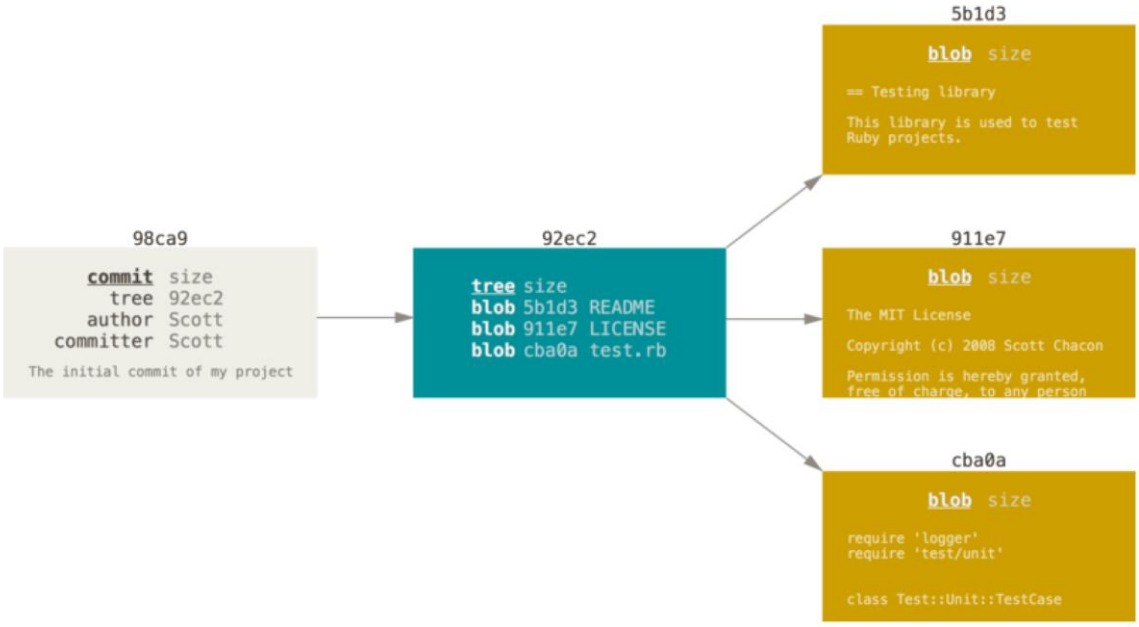
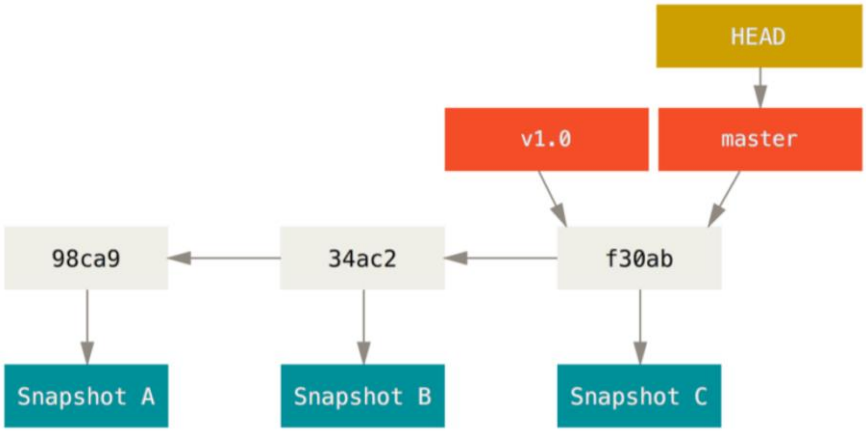


Figure 9. 커밋과 트리 데이터

Git의 브랜치는 커밋 사이를 가볍게 이동할 수 있는 포인터 같은 것이다. 기본적으로 Git은 master 브랜치를 만들고 처음 커밋하면 이 master[16] 브랜치가 생성된 커밋을 가리킨다. 이후 커밋을 만들면 브랜치는 자동으로 가장 마지막 커밋을 가리킨다.



[16] Git 버전 관리 시스템에서 “master” 브랜치는 특별하지 않다. 다른 브랜치와 다른 것이 없다. 다만 모든저장소에서 “master” 브랜치가 존재하는 이유는 `git init` 명령으로 초기화할 때 자동으로 만들어진 이 브랜치를 애써 다른 이름으로 변경하지 않기 때문이다.

2. 새 브랜치 생성하기

git branch 명령으로 브랜치를 만들 수 있다.

```
$ git branch testing
```

새로 만든 브랜치도 지금 작업하고 있던 마지막 커밋을 가리킨다.

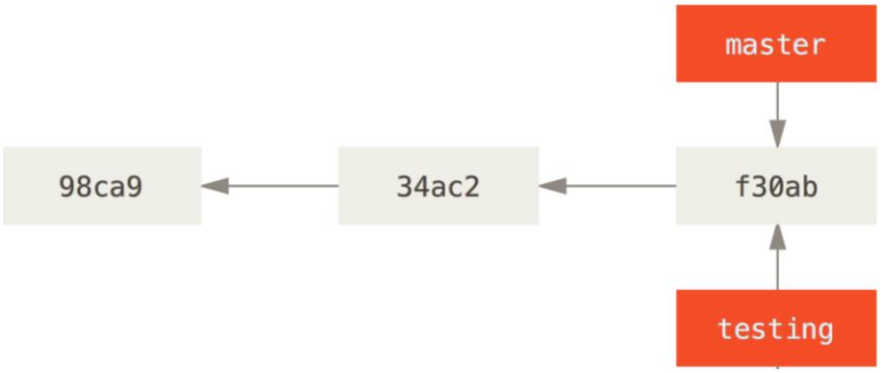


Figure 12. 한 커밋 히스토리를 가리키는 두 브랜치

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 master브랜치를 가리키고 있다. git branch 명령은 브랜치를 만들기만 하고 브랜치를 옮기지 않는다.

Figure 13. 현재 작업 중인 브랜치를 가리키는 HEAD

git log명령에 --decorate옵션을 사용하면 쉽게 브랜치가 어떤 커밋을 가리키는지도 확인 할 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 f30ab커밋 옆에 위치하여 이런식으로 브랜치가 카리키는 커밋을 확인할 수 있다.

3. 브랜치 이동하기

git checkout 명령으로 다른 브랜치로 이동할 수 있다.

```
$ git checkout testing
```

이렇게 하면 HEAD는 testing 브랜치를 가리킨다.

Figure 14. HEAD는 testing 브랜치를 가리킴

커밋을 새로 한번 하고 나면,

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

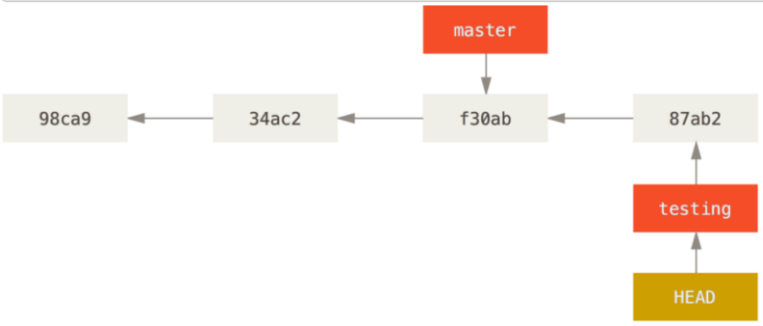


Figure 15. HEAD가 가리키는 testing 브랜치가 새 커밋을 가리킴

새로 커밋해서 testing 브랜치는 앞으로 이동했지만 master 브랜치는 여전히 이전 커밋을 가리킨다. master 브랜치로 되돌아가보자.

```
$ git checkout master
```

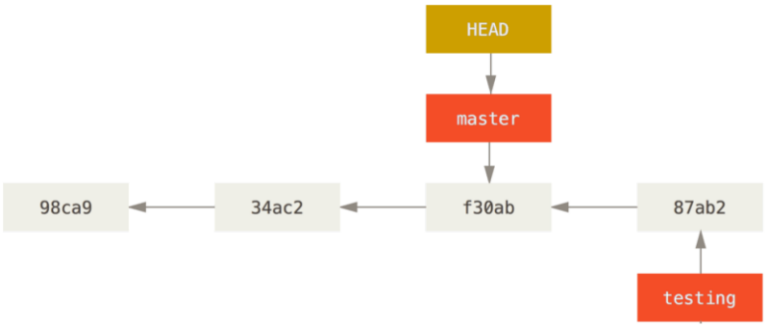


Figure 16. HEAD가 Checkout 한 브랜치로 이동함

master 브랜치가 가리키는 커밋을 HEAD가 가리키게 하고 워킹 디렉토리의 파일도 그 시점[17]으로 되돌려 놓았다. 앞으로 커밋하면 다른 브랜치의 작업과 별개로 진행되어 testing 브랜치에서 임시로 작업 후 master로 돌아온다.

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

파일을 수정하고 다시 커밋을 하면 프로젝트 히스토리는 분리돼 진행된다. 브랜치를 하나 만들어 그 브랜치에서 작업 후 다시 원래 브랜치로 되돌아와 다른 일을 하면 두 작업은 서로 독립적으로 각 브랜치에 존재한다.

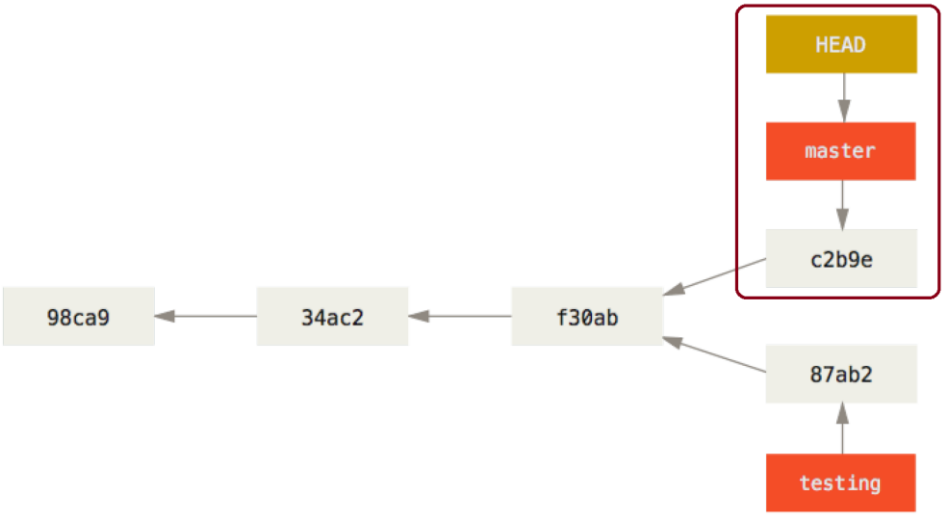


Figure 17. 갈라지는 브랜치

커밋 사이를 자유롭게 이동하다가 때가 되면 두 브랜치를 Merge 한다.
Branch,checkout,commit을 써서 말이다.

```
$ 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
```

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.

브랜치는 어떤 한 커밋을 가리키는 40글자의 SHA-1 체크섬 파일에 불과하여 만들기도 지우기도 쉽다.

---

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

4. 브랜치와 Merge의 기초

브랜치와 Merge는 보통 이런식으로 진행한다.

1. 작업 중인 웹사이트가 있다.
2. 새로운 이슈를 처리할 새 Branch를 하나 생성한다.
3. 새로 만든 Branch에서 작업을 진행한다.

이때 중요한 문제가 생겨서 그것을 해결하는 **Hotfix**를 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.

5. 브랜치의 기초

지금 작업하는 프로젝트에서 이전에 커밋을 몇 번 했다고 가정

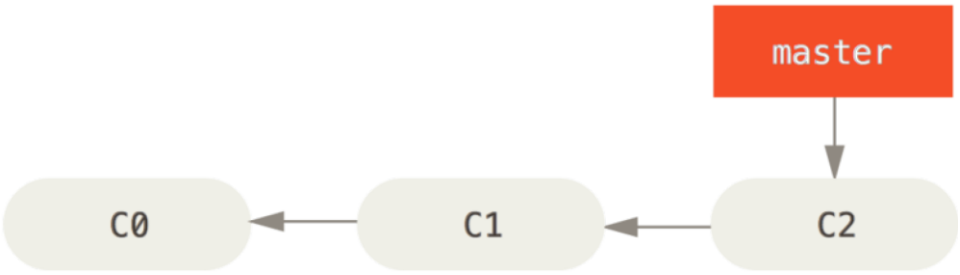


Figure 18. 현재 커밋 히스토리

53번 이슈를 처리한다고 하면 이 이슈에 집중할 수 있는 브랜치를 새로 하나 만든다.

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

브랜치를 만들면서 Checkout까지하려면 git checkout 명령에 -b라는 옵션을 추가한다.

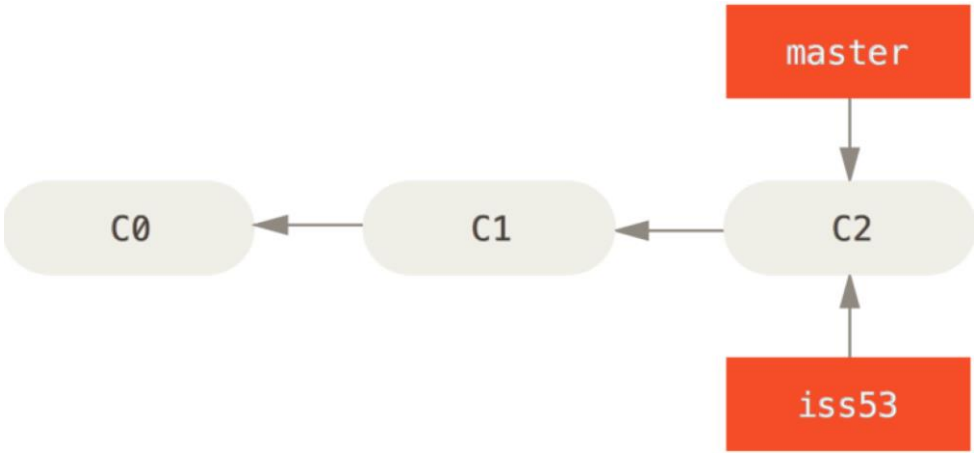


Figure 19. 브랜치 포인터를 새로 만듦

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

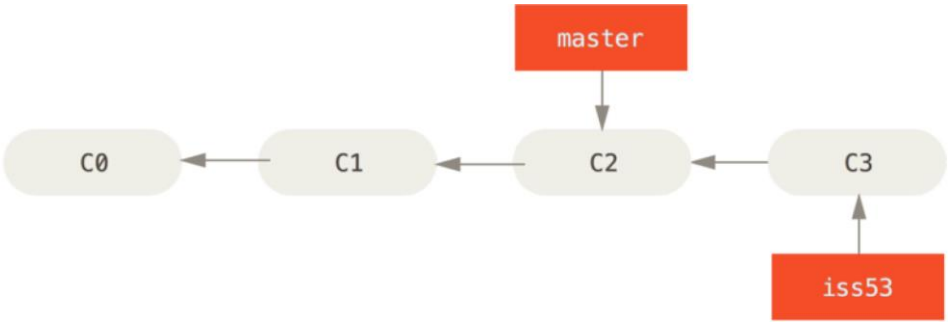


Figure 20. 진행 중인 iss53 브랜치

만약 만드는 사이트에 문제가 생겨 즉시 고쳐야 할 경우, master브랜치로 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. 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.

해결해야 할 핫픽스가 생긴다면, hotfix라는 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(+)
```

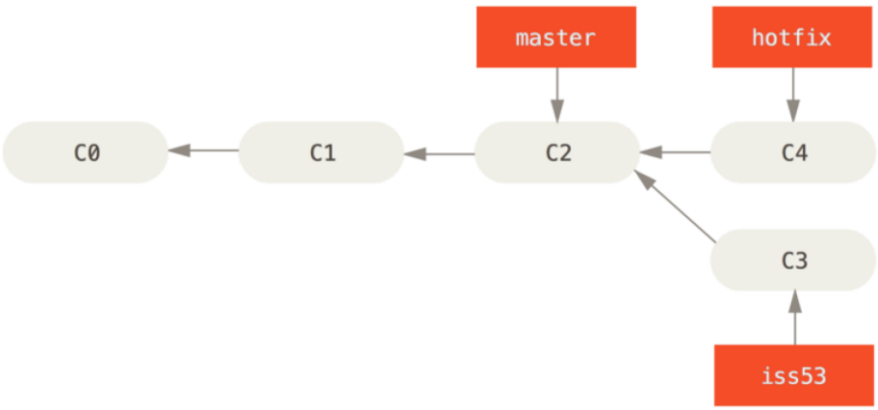


Figure 21. master 브랜치에서 갈라져 나온 hotfix 브랜치

운영환경에 적용하려면 문제를 제대로 고쳤는지 테스트하고 master 브랜치에 합쳐야 한다.

git merge명령으로 아래와 같이 한다.

Remote에 push한 것은 되돌리면 안 된다. 잘못 push한 것을 되돌리는 revert를 만들어서 push 해야 한다.
git reset 이랑 checkout 부분 공부(p.239)