2019-09-02

**Pro Git // https://git-scm.com/book/ko/v2**

**https://github.com/progit/progit/tree/master/ko**

## 1장 시작하기

### 1. 관리 시스템 (Version Control System)

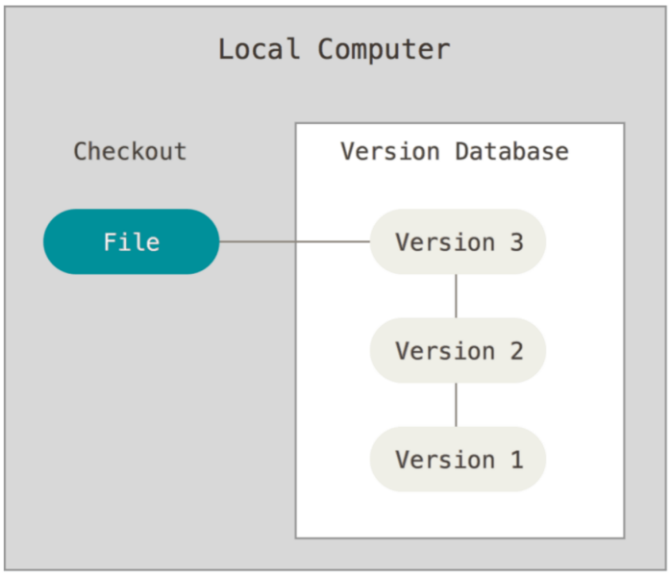
https://ko.wikipedia.org/wiki/%EB%B2%84%EC%A0%84_%EA%B4%80%EB%A6%AC

**버전 관리**(version control, revision control), 소스 관리(source control), 소스 코드 관리(source code management, SCM)란 동일한 정보에 대한 여러 버전을 관리하는 것을 말한다. 공학과 소프트웨어 개발에서 팀 단위로 개발 중인 소스 코드나, 청사진 같은 설계도 등의 디지털 문서를 관리하는데 사용된다. "버전"을 통해서 시간적으로 변경 사항과 그 변경 사항을 작성한 작업자를 추적할 수 있다.

---

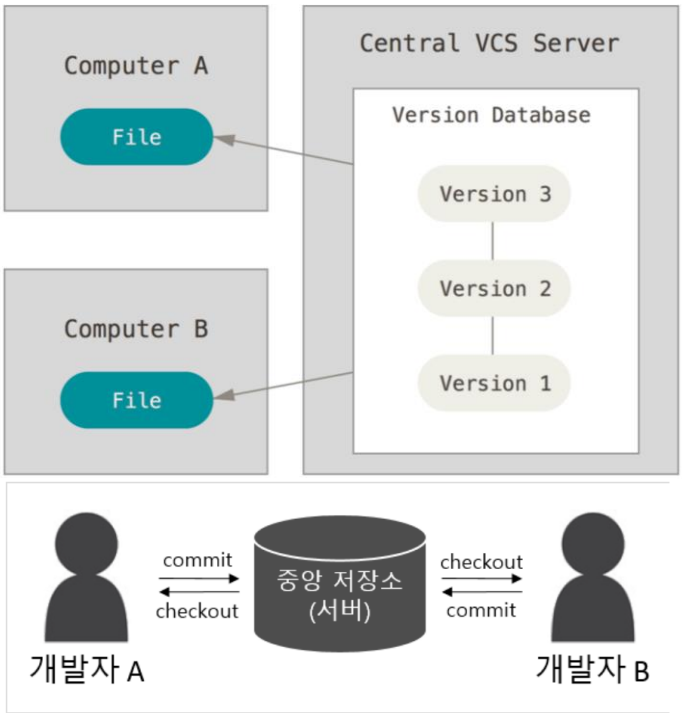
**- 로컬 버전 관리 시스템**

유명했던 VCS 도구들 중 현재에도 널리 쓰이는 것으로 RCS(Revision Control System)라 불리는 시스템이 있다. RCS의 기본적인 동작 방식은 각 리비전들 간의 패치 세트(patch set)라고 하는 데이터의 차이점들을 특별한 형식의 파일에 저장, 특정 시점의 파일 내용을 보고 싶을 때 해당 시점까지의 패치들을 모두 더하여 파일을 만들어내는 것이다.



**- 중앙집중식 버전 관리 시스템 (Centralized Version Control System; CVCS)**

프로젝트를 진행하다 보면 다른 개발자와 함께 작업해야 하는 경우가 많다. 이럴 때 생기는 문제를 해결하기 위해 CVCS (중앙집중식 VCS)가 개발됐다. CVCS에서는 버전 관리되는 모든 파일을 저장하는 하나의 서버와 이 중앙 서버에서 파일들을 가져오는(checkout) 다수의 클라이언트가 존재한다.



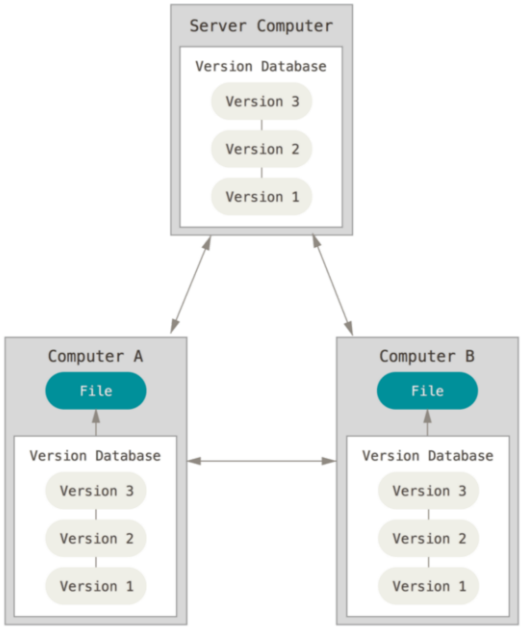

https://gangju.tistory.com/12

누구나 다른 사람들이 무엇을 하고 있는지 알 수 있고, 관리자는 누가 무엇을 할 수 있는지 꼼꼼하게 관리할 수 있다.

그러나 CVCS는 심각한 단점이 있다. 중앙 서버가 잘못되면 모든 것이 잘못된다는 점이다. 서버가 다운될 경우 서버가 다시 복구될 때까지 다른 사람과의 협업도, 진행 중이던 작업을 버전 관리하는 것도 불가능해진다. 그리고 중앙 데이터베이스가 있는 하드디스크에 문제가 생기면 프로젝트의 모든 히스토리를 잃는다. 로컬 VCS 시스템도 같은 문제가 있다. 프로젝트의 모든 이력이 한곳에만 있을 경우 이것은 피할 수 없는 문제다.

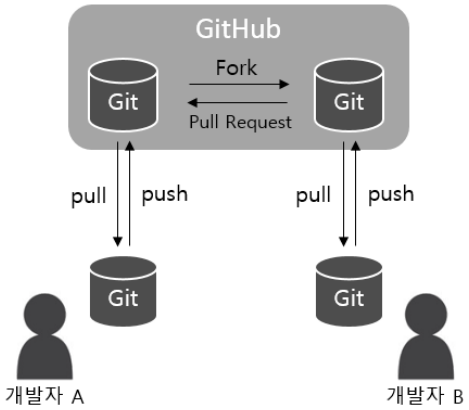
**- 분산 버전 관리 시스템(Distributed Version Control System; DVCS)**

DVCS에서의 클라이언트는 단순히 파일의 마지막 스냅샷을 Checkout 하지 않는다. 그냥 저장소를 전부 복제한다. 서버에 문제가 생기면 이 복제물로 다시 작업을 시작할 수 있다. 체크아웃(checkout)을 할 때마다 전체 백업이 일어나는 셈이다.




게다가 대부분의 DVCS 에서는 다수의 원격 저장소(remote repository)를 갖는 것이 가능하기 때문에 동시에 여러 그룹과 여러 방법으로 함께 작업할 수 있다. 이로 인해 계층 모델(hierarchical model) 등 중앙집중 시스템에서는 할 수 없는 다양한 작업 방식(workflow)들을 사용해볼 수 있다.

아래의 그림의 Fork(포크)는 GitHub 에 있는 특정 리포지토리를 자신의 계정으로 복제하는 것이고, 이렇게 복제된 리포지토리는 원래 리포지토리와 완전히 다른 리포지토리가 된다. 또한 서버(GitHub)와 사용자 만의 push, pull 이 아닌 다른 사용자 사이에도 직접 push, pull 이 가능하다.

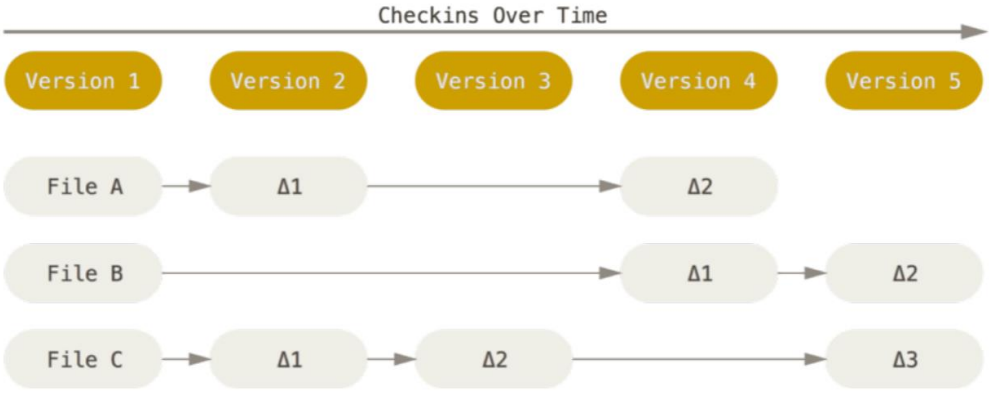


## 2. Git 기초

### Git 의 핵심

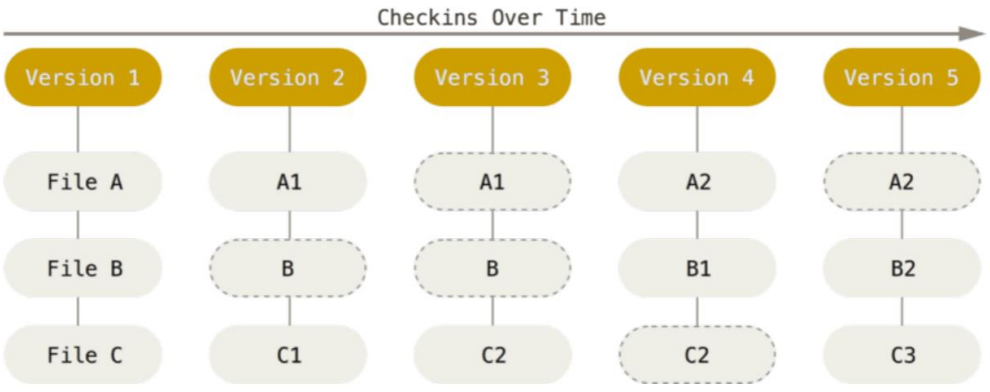
### 1. 차이(델타)가 아닌 스냅샷

- 대부분의 VCS 시스템 (파일의 차이를 관리)




CVS, Subversion, Perforce, Bazaar 등의 시스템은 각 파일의 변화를 시간순으로 관리하면서 파일들의 집합을 관리한다.

- Git(스냅샷을 관리)




Git 은 데이터를 파일 시스템 스냅샷으로 취급하고 크기가 아주 작다. Git 은 커밋하거나 프로젝트의 상태를 저장할 때마다 **파일이 존재하는 그 순간을 중요하게 여긴다.** 파일이 달라지지 않았으면 Git 은 성능을 위해서 파일을 새로 저장하지 않는다. 단지 이전 상태의 파일에 대한 링크만 저장한다. Git 은 데이터를 **스냅샷의 스트림**처럼 취급한다.

**2. 거의 모든 명령을 로컬에서 실행**

거의 모든 명령이 로컬 파일과 데이터만 사용하기 때문에 네트워크에 있는 다른 컴퓨터는 필요 없다. 프로젝트의 모든 히스토리가 로컬 디스크에 있기 때문에 모든 명령을 순식간에 실행된다. 파일을 비교하기 위해 리모트에 있는 서버에 접근하고 나서 예전 버전을 가져올 필요가 없다. 즉 오프라인 상태에서도 비교할 수 있다. 네트워크에 접속하고 있지 않아도 커밋할 수 있다. 다른 VCS 시스템에서는 불가능한 일이다.

**3. Git 의 무결성**

Git 은 데이터를 저장하기 전에 항상 체크섬을 구하고 그 체크섬으로 데이터를 관리한다. 그래서 체크섬을 이해하는 Git 없이는 어떠한 파일이나 디렉토리도 변경할 수 없다. 체크섬은 Git 에서 사용하는 가장 기본적인(Atomic) 데이터 단위이자 Git 의 기본 철학이다. Git 없이는 체크섬을 다룰 수 없어서 파일의 상태도 알 수 없고 심지어 데이터를 잃어버릴 수도 없다. Git 은 SHA-1 해시를 사용하여 체크섬을 만든다. 만든 체크섬은 40 자 길이의 16 진수 문자열이다. 파일의 내용이나 디렉토리 구조를 이용하여 체크섬을 구한다. SHA-1 은 아래처럼 생겼다.

24b9da6552252987aa493b52f8696cd6d3b00373

실제로 Git 은 파일을 이름으로 저장하지 않고 해당 파일의 해시로 저장하여 식별한다.

https://ko.wikipedia.org/wiki/%EC%B2%B4%ED%81%AC%EC%84%AC

**체크섬**(checksum)은 중복 검사의 한 형태로, 오류 정정을 통해, 공간(전자 통신)이나 시간(기억 장치) 속에서 송신된 자료의 무결성을 보호하는 단순한 방법이다.

---

https://ko.wikipedia.org/wiki/SHA

**SHA**(Secure Hash Algorithm, 안전한 해시 알고리즘) 함수들은 서로 관련된 암호학적 해시 함수들의 모음이다.

SHA-1 은 SHA 함수들 중 가장 많이 쓰이며, TLS, SSL, PGP, SSH, IPSec 등 많은 보안 프로토콜과 프로그램에서 사용되고 있다.

SHA-1 은 SHA-0 의 압축 함수에 비트 회전 연산을 하나 추가한 것이다.

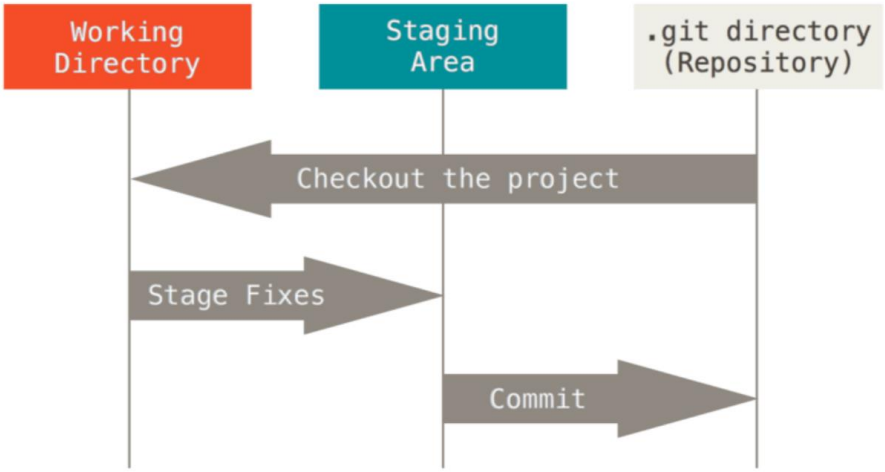
---

**4. Git 은 데이터를 추가할 뿐**

Git 으로 무얼 하든 데이터를 추가한다. 되돌리거나 데이터를 삭제할 방법이 없다. 다른 VCS 처럼 Git 도 커밋하지 않으면 변경사항을 잃어버릴 수 있다. 하지만, 일단 스냅샷을 커밋하고 나면 데이터를 잃어버리기 어렵다.

Git 을 사용하면 프로젝트가 심각하게 망가질 걱정 없이 매우 즐겁게 여러 가지 실험을 해 볼 수 있다. 9 장을 보면 Git 이 데이터를 어떻게 저장하고 손실을 어떻게 복구해야 할지 알 수 있다.

**5. 세 가지 상태**




Git 파일 관리 상태

- Committed : 데이터가 로컬 데이터베이스에 안전하게 저장됨(commit 명령어)

- Modified : 수정한 파일을 아직 로컬 데이터베이스에 커밋하지 않음

- Staged : 현재 수정한 파일을 곧 커밋할 것이라고 표시한 상태 (add 명령어)

Git 프로젝트의 세 가지 단계

- Git directory : Git 이 프로젝트의 메타데이터와 객체 데이터베이스를 저장하는 곳. 다른 컴퓨터에 있는 저장소를 Clone 할 때 Git 디렉토리가 만들어짐.

- Working directory : 프로젝트의 특정 버전을 Checkout 한 것. Git 디렉토리는 지금 작업하는 디스크에 있고 그 디렉토리에 압축된 데이터베이스에서 파일을 가져와서 워킹 디렉토리를 만든다.

- Staging Area : Git 디렉토리에 있다. 단순한 파일이고 곧 커밋할 파일에 대한 정보를 저장한다.

**정리**

1. 워킹 디렉토리에서 파일을 수정한다.

2. Staging Area 에 파일을 Stage 해서 커밋할 스냅샷을 만든다.

3. Staging Area 에 있는 파일들을 커밋해서 Git 디렉토리에 영구적인 스냅샷으로 저장한다.

(Git 디렉토리에 있는 파일들은 Committed 상태이다. 파일을 수정하고 Staging Area 에 추가했다면 Staged 이다. 그리고 Checkout 하고 나서 수정했지만, 아직 Staging Area 에 추가하지 않았으면 Modified 이다.)

## 2장 Git의 기초

### 1. Git 저장소 만들기

**- 기존 디렉토리를 Git 저장소로 만들기**

기존 프로젝트를 Git으로 관리하고 싶을 때, 프로젝트 디렉토리로 이동 후 **git init** 명령 실행

```
$ git init
```

Git이 파일을 관리하게 하려면 저장소에 파일을 추가하고 커밋해야 한다. **git add** 명령으로 파일을 추가하고 **git commit** 명령으로 커밋한다.

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

**- 다른 서버에 있는 저장소를 Clone**

Git이 Subversion과 다른 가장 큰 차이점은 서버에 있는 거의 모든 데이터를 복사한다는 것이다. **git clone [url]** 명령으로 저장소를 Clone 한다.

```
$ git clone git://github.com/schacon/grit.git
```

이 명령은 "grit"이라는 디렉토리를 만들고 그 안에 `.git` 디렉토리를 만든다. 그리고 저장소의 데이터를 모두 가져와서 자동으로 가장 최신 버전을 Checkout해 놓는다. `grit` 디렉토리로 이동하면 Checkout으로 생성한 파일을 볼 수 있고 당장 하고자 하는 일을 시작할 수 있다.

아래와 같은 명령을 사용하여 저장소를 Clone하면 "grit"이 아니라 다른 디렉토리 이름으로 Clone 할 수 있다

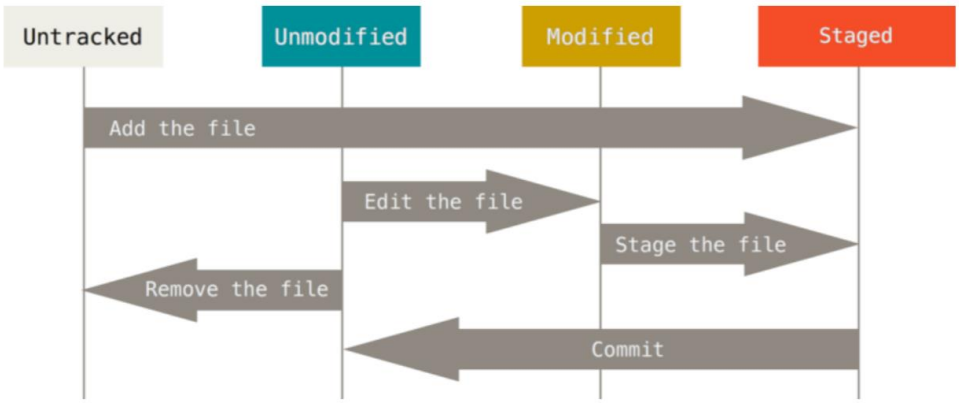
```
$ git clone git://github.com/schacon/grit.git mygrit
```

**cf. git clone -o mygrit** → mygrit/master라고 사용자가 정한 대로 리모트 이름을 생성해줌.

## 2. 수정하고 저장소에 저장하기

- 워킹 디렉토리 내 파일


- 파일의 라이프사이클



**- 파일의 상태 확인하기 : git status 명령**

**git status** 명령 실행

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

→ 파일을 하나도 수정하지 않음을 보여줌(Tracked, Modified 상태 파일이 없음)

→ Untracked 파일은 아직 없어서 목록에 나타나지 않음

→ 현재 브랜치를 알려줌

프로젝트에 README 파일을 생성. git status 실행 시 'Untracked files'에 들어감.

```
$ vim 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)
```

Untracked 파일 : 스냅샷(커밋)에 넣어지지 않은 파일. Tracked 상태가 되기 전까지는 Git 은 절대 그 파일을 커밋하지 않는다.

**- 파일을 새로 추적하기**

**git add** 명령으로 파일을 새로 추적 가능

```
$ git add README
```

**git status** 명령을 다시 실행시 README 파일이 Tracked 상태+Staged 상태인 것 확인 가능

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

        new file:   README
```

- "Changes to be committed" : Staged 상태라는 것을 의미한다.

**- Modified 상태의 파일을 stage 하기**

- "Changes not staged for commit" : 수정한 파일이 Tracked 상태이지만 아직 Staged 상태는 아님. (Modified 상태) **git add** 명령을 실행해야 staged 상태로 변경 가능.

- git add 명령을 실행한 후(Staged 상태로 변경) 또 파일을 수정하면 git add 명령을 다시 실행해야 한다. 그렇지 않으면 최신 버전은 Modified 상태. 기존 버전만 Staged 상태

**git add** : 파일을 새로 추적할 때도, 수정한 파일을 Staged 상태로 만들 때도 사용. Merge 할 때 충돌난 상태의 파일을 Resolve 상태로 만들 때도 사용한다. 프로젝트에 파일을 추가한다기보다는 다음 커밋에 추가한다고 받아들이는 게 좋다.

**- 파일 상태를 짤막하게 확인하기**

**git status -s** 또는 **git status --short** 옵션 사용

```
$ git status -s
   M README
  MM Rakefile
  A  lib/git.rb
  M  lib/simplegit.rb
  ?? LICENSE.txt
```

- ?? : Untracked

- A : Staged 상태로 추가한 파일 중 새로 생성한 파일

- M : 수정한 파일

# 상태정보 컬럼 : 두 가지 정보(왼쪽 : Staging Area 상태 / 오른쪽 : WorkingTree상태)

```
   M README
```

왼쪽 : _ / 오른쪽 : M → 내용을 변경했지만 아직 Staged 상태 X

```
  M  lib/simplegit.rb
```

왼쪽 : M / 오른쪽 : _ → 내용을 변경하고 Staged 상태로 추가까지 한 상태

```
  MM Rakefile
```

왼쪽 : M / 오른쪽 : M → 내용을 변경하고 Staged 상태로 추가한 후 또 내용을 변경해서 Staged 이면서 Unstaged 상태

- **파일 무시하기**

.gitignore 파일을 만들고 그 안에 무시할 파일 패턴을 적는다. 아래는 .gitignore 파일의 예이다:

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

https://github.com/github/gitignore 예제 확인

- **Staged와 Unstaged 상태의 변경 내용을 보기**

**git status** : Staged파일 / Unstaged파일 구분

**git diff** : 어떤 라인을 추가하고 삭제했는지 궁금할 때 사용

**git diff** : Staging Area에 있는 파일(Staged)과 워킹 디렉토리에 있는 파일(Unstaged) 비교

```
$ git diff
diff --git a/benchmarks.rb b/benchmarks.rb
index 3cb747f..da65585 100644
--- a/benchmarks.rb
+++ b/benchmarks.rb
@@ -36,6 +36,10 @@ def main
           @commit.parents[0].parents[0].parents[0]
         end

+        run_code(x, 'commits 1') do
+          git.commits.size
+        end
+
         run_code(x, 'commits 2') do
           log = git.commits('master', 15)
           log.size
```

→ git diff는 Unstaged 상태인 것들만 보여준다. 수정한 파일을 모두 Staging Area에 넣었다면 git diff 명령은 아무것도 출력하지 않는다.

**git diff --staged** 옵션 사용 : 저장소에 커밋한 것과 Staging Area에 있는 것을 비교한다.

```
$ git diff --cached
diff --git a/README b/README
new file mode 100644
index 0000000..03902a1
--- /dev/null
+++ b/README2
@@ -0,0 +1,5 @@
+grit
+ by Tom Preston-Werner, Chris Wanstrath
+ http://github.com/mojombo/grit
+
+Grit is a Ruby library for extracting information from a Git repository
```

**- 변경사항 커밋하기**

**git commit** 명령 : Staging Area 에 정리한 파일(staged 상태)을 커밋한다.

```
$ git commit
```

Git 설정에 지정된 편집기가 실행되고, 아래와 같은 텍스트가 자동으로 포함된다.

```
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
# Changes to be committed:
#       new file:   README
#       modified:   benchmarks.rb
#
~
~
~
".git/COMMIT_EDITMSG" 10L, 283C
```

(수정한 내용을 좀 더 구체적으로 남겨 둘 수 있다. **git commit 에 -v 옵션** 추가하면 편집기에 diff 메시지도 추가된다).

메시지를 인라인으로 첨부할 수도 있다. -m **옵션을** 사용한다:

```
$ git commit -m "Story 182: Fix benchmarks for speed"
[master 463dc4f] Story 182: Fix benchmarks for speed
 2 files changed, 3 insertions(+)
 create mode 100644 README
```

master 브랜치에 커밋, 체크섬은 (463dc4f). 수정한 파일 수, 삭제/추가된 라인의 수도 알려준다.

**# 커밋 :** Git 은 Staging Area 에 속한 스냅샷을 커밋한다. 수정을 했더라도 아직 Staging Area 에 넣지 않은 것은 다음에 커밋 가능. 커밋 시 프로젝트의 스냅샷을 기록하기 때문에 추후 스냅샷끼리 비교하거나 예전 스냅샷으로 되돌릴 수 있다.

**- Staging Area 생략하기**

`git commit -a` : Tracked 상태의 파일을 자동으로 Staging Area 에 넣음. Git add 명령을 실행하지 않아도 된다.

```
$ 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:   benchmarks.rb

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 files changed, 5 insertions(+)
```

**- 파일을 삭제하기**

**`git rm`** : Tracked(Staging Area) 상태의 파일을 삭제. 실제 파일도 지워짐.

- Git 에서 git rm 명령으로 삭제한 파일은 Staged 상태

- Git 없이 삭제한 후 git status 로 명령을 상태를 확인하면 Changes not staged for commit(Unstaged) 상태

- 커밋 후 파일은 삭제되고 Untracked 상태. 이미 파일을 수정했거나 Staging Area 에 추가했다면 **-f** 옵션으로 강제 삭제해야 함. (안전장치)

**`git rm -cached`** **:** Staging Area 에서만 제거하고 워킹디렉토리에 있는 파일은 삭제하지 않음.

```
$ git rm --cached readme.txt
```

- file-glob 패턴 사용도 가능(\ 필요)

```
$ git rm log/\*.log
```
: log/ 디렉토리에 있는 .log 파일 모두 삭제

```
$ git rm \*~
```
: ~로 끝나는 파일 모두 삭제

- **파일 이름 변경하기**

  **git mv file_from file_to**

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

        renamed:    README.txt -> README
```

  **git mv** 명령은 아래 명령어들을 수행한 것과 같다.

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

### 3. 커밋 히스토리 조회하기

**git log** : 히스토리를 조회

예제 Clone 후

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

```
$ 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 code

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

    first commit
```

→ 가장 최근의 커밋이 가장 먼저 나온다.

→ 각 커밋의 SHA-1 체크섬, 저자 이름, 이메일, 커밋한 날짜, 커밋 메시지를 보여줌.

**git log -p** 옵션 : 각 커밋의 diff 결과를 보여줌. / **-2** 옵션 : 최근 두 개의 결과만 보여줌

```
$ 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 +5,5 @@ require 'rake/gempackagetask'
 spec = Gem::Specification.new do |s|
     s.name      =   "simplegit"
-    s.version   =   "0.1.0"
+    s.version   =   "0.1.1"
     s.author    =   "Scott Chacon"
     s.email     =   "schacon@gee-mail.com"

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

    removed unnecessary test code

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
\ No newline at end of file
```

→ 직접 diff 를 실행한 것과 같은 결과를 출력하기 때문에 동료가 무엇을 커밋했는지 리뷰하고 빨리 조회하는데 유용하다.

**git log -U1 –word-diff** 옵션 : 줄 대신 단어 단위로, 해당 줄만 보여줌. 소스코드보다 책/에세이 적합.

```
$ git log -U1 --word-diff
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
@@ -7,3 +7,3 @@ spec = Gem::Specification.new do |s|
    s.name      =   "simplegit"
    s.version   =   [-"0.1.0"-]{+"0.1.1"+}
    s.author    =   "Scott Chacon"
```

diff: 기본적으로 해당 줄 + 위 아래 줄 = 3 줄을 보여준다. 단어 단위로 비교해서 볼 때는 굳이 3 줄을 다 볼 필요가 없으므로 -U1 옵션 사용.

**git log --stat** 옵션 : 각 커밋의 통계 정보 조회 가능. 요약 정보는 가장 뒤쪽에 보여줌

```
$ 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 code

 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(+)
```

**git log --pretty** 옵션 : **oneline**(각 커밋을 한 줄로 보여줌), **short, full, fuller** 사용 가능.

```
$ git log --pretty=oneline
ca82a6dff817ec66f44342007202690a93763949 changed the version number
085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7 removed unnecessary test code
a11bef06a3f659402fe7563abf99ad00de2209e6 first commit
```

**git log --pretty=format** 옵션 : 나만의 포맷으로 결과를 출력함.

```
$ git log --pretty=format:"%h - %an, %ar : %s"
ca82a6d - Scott Chacon, 11 months ago : changed the version number
085bb3b - Scott Chacon, 11 months ago : removed unnecessary test code
a11bef0 - Scott Chacon, 11 months ago : first commit
```

git log pretty=format 에 쓸 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 |

**--graph** 옵션 : oneline, format 옵션과 사용해 아스키 그래프 출력

```
$ git log --pretty=format:"%h %s" --graph
* 2d3acf9 ignore errors from SIGCHLD on trap
*  5e3ee11 Merge branch 'master' of git://github.com/dustin/grit
|\
| * 420eac9 Added a method for getting the current branch.
* | 30e367c timeout code and tests
* | 5a09431 add timeout protection to grit
* | e1193f8 support for heads with slashes in them
|/
* d6016bc require time for xmlschema
*  11d191e Merge branch 'defunkt' into local
```

→ 브랜치, Merge 결과 히스토리 살펴보면 흥미로움.

*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. |

- 조회 제한 조건

*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. |

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

지난 2 주 동안 만들어진 커밋들만 조회하는 명령, "2008-01-15" 정확한 기간 형태나 "2 years 1 day 3 minutes ago"의 상대적 기간 형태도 사용 가능

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

어떤 함수가 추가되거나 제거된 커밋만을 찾아보기 위해 사용.

디렉토리나 파일 이름을 사용하여 그 파일이 변경된 log 의 결과를 검색할 수 있다. 이 옵션은 -- 와 함께 경로 이름을 사용하는데 명령어 끝 부분에 쓴다(역주, `git log -- path1 path2`).

```
$ git log --pretty="%h - %s" --author=gitster --since="2008-10-01" \
   --before="2008-11-01" --no-merges -- t/
5610e3b - Fix testcase failure when extended attribute
acd3b9e - Enhance hold_lock_file_for_{update,append}()
f563754 - demonstrate breakage of detached checkout wi
d1a43f2 - reset --hard/read-tree --reset -u: remove un
51a94af - Fix "checkout --track -b newbranch" on detac
b0ad11e - pull: allow "git pull origin $something:$cur
```

2008 년 10 월에 Junio Hamano 가 커밋한 히스토리를 조회, 그 중에서 테스트 파일을 수정한 커밋 중에서 머지 커밋이 아닌 것들만 조회한다

## 4. 되돌리기 ※ 주의 : 한 번 되돌리면 복구할 수 없음

**- 커밋 수정하기**

**git commit --amend** 옵션 : 다시 커밋하고 싶을 때 사용

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

Staging Area 를 사용하여 커밋함, 편집기가 실행되면 이전 커밋 메시지가 자동으로 포함됨.

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

→ 커밋 시 Stage 할 파일을 빠트렸을 때 위와 같이 고칠 수 있다. 두 번째 커밋은 첫 번째 커밋을 덮어쓴다.

**- 파일 상태를 Unstaged 로 변경하기**

**git reset HEAD <file>** : Unstage 상태로 변경

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

        modified:   README.txt
        modified:   benchmarks.rb
```

→ 실수로 모든 파일을 Staged 상태로 만듦.

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

        modified:   README.txt

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:   benchmarks.rb
```

→ git reset HEAD benchmarks.rb 명령을 통해 해당 파일을 Unstage 상태로 변경.

**※ git reset 명령은 매우 위험함. --hard 옵션과 함께 사용하면 더욱 위험. 하지만 옵션 없이 사용하면 워킹 디렉토리의 파일은 건드리지 않는다.**

**- Modified 파일 되돌리기**

**git checkout -- <file>** : 최근 커밋된 버전 / 처음 Checkout 한 내용으로 되돌리는 방법

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

        modified:   README.txt
```

**※ git checkout 명령은 위험함. 수정한 내용은 전부 사라짐. 대신 Stash 와 Branch 사용 권장**

## 5. 리모트 저장소

리모트 저장소 : 인터넷/네트워크에 있는 저장소

**- 리모트 저장소 확인하기**

**git remote** 명령 : 현재 프로젝트에 등록된 리모트 저장소 확인.

```
$ git clone git://github.com/schacon/ticgit.git
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 | 193.00 KiB/s, done.
Resolving deltas: 100% (772/772), done.
Checking connectivity... done.
$ cd ticgit
$ git remote
origin
```

**git remote -v 옵션** : 단축이름과 URL 함께 보기

```
$ git remote -v
bakkdoor  git://github.com/bakkdoor/grit.git
cho45     git://github.com/cho45/grit.git
defunkt   git://github.com/defunkt/grit.git
koke      git://github.com/koke/grit.git
origin    git@github.com:mojombo/grit.git
```

cf. origin 만 SSH URL 이기 때문에 origin 에만 Push 할 수 있다(4 장에서 좀 더 자세히 다룬다).

**- 리모트 저장소 추가하기**

**git remote add [단축이름] [url]** : 새 리모트 저장소 추가

```
$ git remote
origin
$ git remote add pb git://github.com/paulboone/ticgit.git
$ git remote -v
origin  git://github.com/schacon/ticgit.git
pb      git://github.com/paulboone/ticgit.git
```

→ URL 대신 pb 사용 가능.

**git fetch pb** : 로컬 저장소에 없지만, Paul 의 저장소에 있는 것을 가져올 때 실행

```
$ git fetch pb
remote: Counting objects: 58, done.
remote: Compressing objects: 100% (41/41), done.
remote: Total 44 (delta 24), reused 1 (delta 0)
Unpacking objects: 100% (44/44), done.
From git://github.com/paulboone/ticgit
 * [new branch]      master     -> pb/master
 * [new branch]      ticgit     -> pb/ticgit
```

**- 리모트 저장소를 Pull 하거나 Fetch 하기**

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

**git fetch** : 로컬에는 없지만, 리모트 저장소에 있는 데이터를 모두 가져옴. 이후 리모트 저장소의 모든 브랜치를 로컬에서 저장할 수 있어서 언제든지 머지를 하거나 내용을 살펴볼 수 있다.

**git pull** : 데이터를 가져올 뿐 아니라 자동으로 로컬 브랜치와 머지시킴.

**git clone** : 자동으로 로컬의 master 브랜치가 리모트 저장소의 master 브랜치를 추적하도록 한다. (remote 저장소에 master 브랜치 있다는 가정 하에)

이후 **git pull**→ clone 서버에서 데이터를 가져오고, 자동으로 현재 작업하는 코드와 머지시킴.

이후 **git fetch origin**→ clone 실행 이후 수정된 것을 모두 가져옴. 자동으로 머지하지 않기 때문에 로컬 작업을 정리하고 수동으로 머지해야 함.

**- 리모트 저장소에 Push 하기**

**git push [리모트 저장소 이름] [브랜치 이름]** : clone 한 리모트에 저장소 쓰기 권한이 있고, clone 한 이후 아무도 리모트 저장소에 push 하지 않았을 때만 사용 가능. 여러명이 Clone 했다면 다른 사람이 작업한 것을 가져와서 머지한 후 Push 가능.

**- 리모트 저장소 살펴보기**

**git remote show [리모트 저장소 이름]** : 저장소 구체적인 정보 확인 가능

```
$ 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 URL 과 추적하는 브랜치를 출력. git pull 명령 실행 시 master 브랜치와 Merge 할 브랜치가 무엇인지 보여줌.

```
$ git remote show origin
* remote origin
   URL: https://github.com/my-org/complex-project
   Fetch URL: https://github.com/my-org/complex-project
   Push URL: https://github.com/my-org/complex-project
   HEAD branch: master
   Remote branches:
            master                tracked
            dev-branch            tracked
            markdown-strip        tracked
            issue-43              new (next fetch will store in remotes/origin)
            issue-45              new (next fetch will store in remotes/origin)
            refs/remotes/origin/issue-11          stale (use 'git remote prune' to
remove)
   Local branches configured for 'git pull':
         dev-branch merges with remote dev-branch
         master     merges with remote master
   Local refs configured for 'git push':
       dev-branch                pushes to dev-branch (up to date)
       markdown-strip            pushes to markdown-strip (up to date)
       master                    pushes to master (up to date)
```

→ 아직 로컬로 가져오지 않은 리모트 저장소의 브랜치와, 삭제되었지만 가지고있는 브랜치 표시

→ push, pull 명령 시 자동으로 연결되는 브랜치를 보여줌.

**- 리모트 저장소 이름을 바꾸거나 리모트 저장소를 삭제하기**

**git remote rename**

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

**git remote remove**

```
$ git remote rm paul
$ git remote
origin
```

## 6. 태그

**- 태그 조회하기**

**git tag** : 이미 만들어진 태그 있는지 확인

```
$ git tag -l 'v1.4.2.*'
v1.4.2.1
v1.4.2.2
v1.4.2.3
v1.4.2.4
```

→ 1.4.2 버전의 태그들만 검색

**- 태그 붙이기**

-Lightweight 태그 / Annotated 태그

-Lightweight Tag : 브랜치와 비슷하지만 단순히 특정 커밋에 대한 포인터일 뿐.

-Annotated 태그 : Git 데이터베이스에 태그 만든 사람 이름, 이메일, 태그 날짜, 메시지 저장. GPG(GNU Privacy Guard)로 서명 가능.

**- Annotated 태그**

**git tag -a**

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

→m 옵션으로 메시지도 함께 저장 가능.

**git show** 명령 : 태그 정보와 커밋 정보 모두 확인 가능

```
$ git show v1.4
tag v1.4
Tagger: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Feb 9 14:45:11 2009 -0800

my version 1.4

commit 15027957951b64cf874c3557a0f3547bd83b3ff6
Merge: 4a447f7... a6b4c97...
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sun Feb 8 19:02:46 2009 -0800

    Merge branch 'experiment'
```

**- 태그에 서명하기**

**git tag -s 옵션** (GPG 개인키가 있으면 사용 가능)

```
$ git tag -s v1.5 -m 'my signed 1.5 tag'
You need a passphrase to unlock the secret key for
user: "Scott Chacon <schacon@gee-mail.com>"
1024-bit DSA key, ID F721C45A, created 2009-02-09
```

**git show** - GPG 서명 보기 가능

```
$ git show v1.5
tag v1.5
Tagger: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Feb 9 15:22:20 2009 -0800

my signed 1.5 tag
-----BEGIN PGP SIGNATURE-----
Version: GnuPG v1.4.8 (Darwin)

iEYEABECAAYFAkmQurIACgkQON3DxfchxFr5cACeIMN+ZxLKggJQf0QYiQBwgySN
Ki0An2JeAVUCAiJ7Ox6ZEtK+NvZAj82/
=WryJ
-----END PGP SIGNATURE-----
commit 15027957951b64cf874c3557a0f3547bd83b3ff6
Merge: 4a447f7... a6b4c97...
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sun Feb 8 19:02:46 2009 -0800

    Merge branch 'experiment'
```

**-Lightweight 태그**

**git tag** : 파일에 커밋 체크섬을 저장할 뿐, 다른 정보는 저장하지 않는다. -a, -s, -m 옵션은 사용하지 않는다.

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

**git show**를 실행해도 별도의 태그 정보를 확인할 수 없고, 단순히 커밋 정보만 보여줌.

```
$ git show v1.4-lw
commit 15027957951b64cf874c3557a0f3547bd83b3ff6
Merge: 4a447f7... a6b4c97...
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sun Feb 8 19:02:46 2009 -0800

    Merge branch 'experiment'
```

**- 태그 검증하기**

**git tag -v [태그이름]** : GPG 를 사용하여 서명을 검증. 서명자의 GPG 공개키가 필요함.

```
$ git tag -v v1.4.2.1
object 883653babd8ee7ea23e6a5c392bb739348b1eb61
type commit
tag v1.4.2.1
tagger Junio C Hamano <junkio@cox.net> 1158138501 -0700

GIT 1.4.2.1

Minor fixes since 1.4.2, including git-mv and git-http with alternates.
gpg: Signature made Wed Sep 13 02:08:25 2006 PDT using DSA key ID F3119B9A
gpg: Good signature from "Junio C Hamano <junkio@cox.net>"
gpg:                 aka "[jpeg image of size 1513]"
Primary key fingerprint: 3565 2A26 2040 E066 C9A7  4A7D C0C6 D9A4 F311 9B9A
```

만일 공개키가 없으면 아래와 같이 메시지 출력

```
gpg: Signature made Wed Sep 13 02:08:25 2006 PDT using DSA key ID F3119B9A
gpg: Can't check signature: public key not found
error: could not verify the tag 'v1.4.2.1'
```

**- 나중에 태그하기**

```
$ 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
```

명령 끝에 커밋 체크섬을 명시한다. (긴 체크섬을 모두 사용할 필요는 없음)

```
$ git tag -a v1.2 -m 'version 1.2' 9fceb02
```

**- 태그 공유하기**

**git push origin [태그이름]** : git push 명령은 자동으로 리모트 서버에 태그를 전송하지 않음.

```
$ git push origin v1.5
Counting objects: 50, done.
Compressing objects: 100% (38/38), done.
Writing objects: 100% (44/44), 4.56 KiB, done.
Total 44 (delta 18), reused 8 (delta 1)
To git@github.com:schacon/simplegit.git
* [new tag]         v1.5 -> v1.5
```

**git push --tags** : 태그를 여러 개 push 하고 싶을 때 실행. 리모트 서버에 없는 태그 모두 전송 가능

```
$ git push origin --tags
Counting objects: 50, done.
Compressing objects: 100% (38/38), done.
Writing objects: 100% (44/44), 4.56 KiB, done.
Total 44 (delta 18), reused 8 (delta 1)
To git@github.com:schacon/simplegit.git
 * [new tag]         v0.1 -> v0.1
 * [new tag]         v1.2 -> v1.2
 * [new tag]         v1.4 -> v1.4
 * [new tag]         v1.4-lw -> v1.4-lw
 * [new tag]         v1.5 -> v1.5
```

## 7. Git Alias

**git config** 사용 : 각 명령의 Alias 쉽게 만들 수 있음. 아래는 예시

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

unstage 상태로 변경하는 명령

```
$ git config --global alias.unstage 'reset HEAD --'
```

최근 커밋을 쉽게 확인할 수 있는 명령

```
$ git config --global alias.last 'log -1 HEAD'
```

아래 명령은 `git visual`이라고 입력하면 `gitk`가 실행된다:

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

2019-09-03

## 3장 Git 브랜치

- 브랜치 : 코드와 상관없이 독립적으로 개발하는 것

- Git 의 브랜치 : 커밋사이를 가볍게 이동할 수 있는 **포인터** 같은 것

### 1. 브랜치란 무엇인가

**파일을 Staging Area 에 저장하고 커밋하는 예제**

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

- 파일을 Stage 할 때

→ Blob 생성 : Git 저장소에 파일을 저장하고(**Blob**), Staging Area 에 해당 파일의 체크섬을 저장

- commit 할 때

→ 트리 개체 생성 : 루트 디렉토리와 각 하위 디렉토리의 트리 개체를 체크섬과 함께 저장소에 저장

→ 커밋 개체 생성 : 메타데이터와 루트 디렉토리 트리 개체를 가리키는 포인터 정보를 커밋 개체에 넣어 저장

**# 총 다섯 개 데이터 개체 생성**

- 각 파일에 대한 Blob 세 개
- 파일+디렉토리 구조가 들어있는 트리 개체 하나
- 메타데이터+루트 트리를 가리키는 포인터가 담긴 커밋 개체 하나,

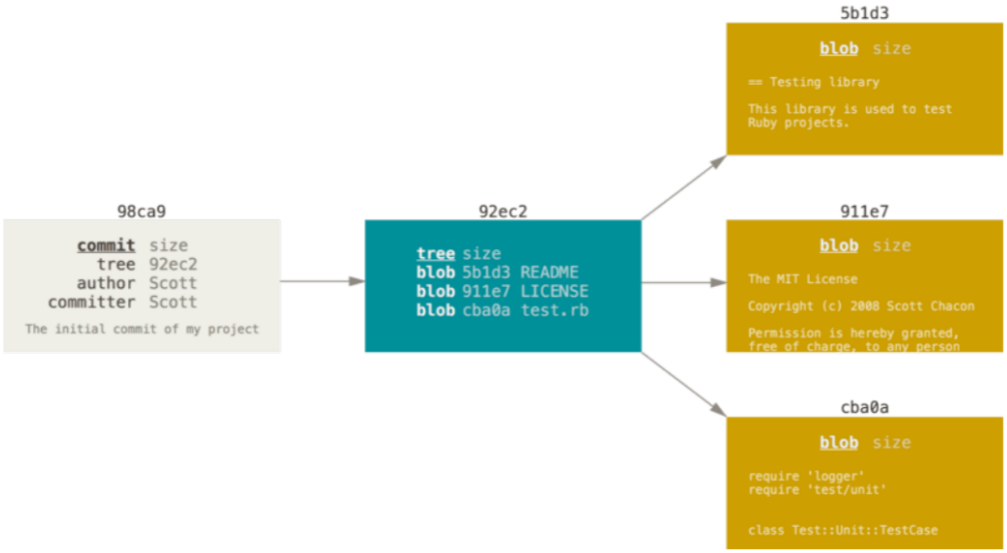



*Figure 9. 커밋과 트리 데이터*

다시 파일을 수정하고 커밋하면 이전 커밋이 무엇인지도 저장한다.

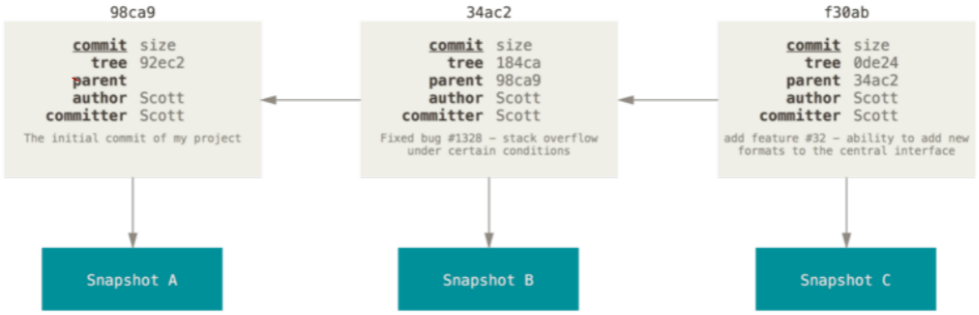



*Figure 10. 커밋과 이전 커밋*

# 최초로 커밋하면 Git 은 master 라는 이름의 브랜치를 만들어서 자동으로 가장 마지막 커밋을 가리키게 함.

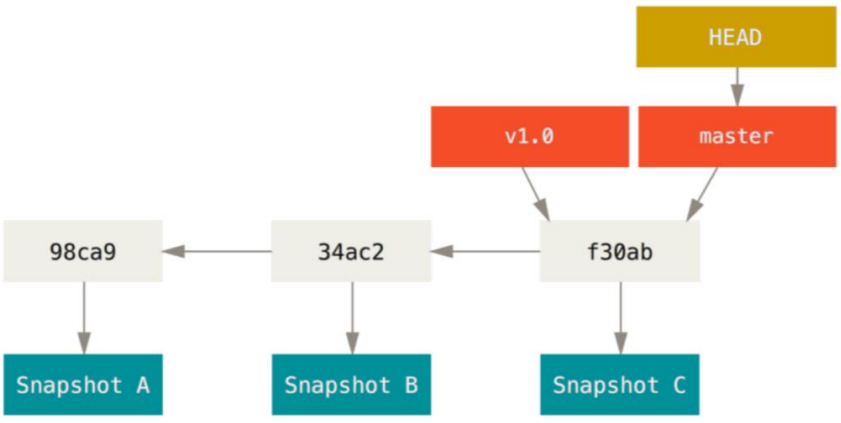



*Figure 11. 브랜치와 커밋 히스토리*

- 새 브런치 생성하기

**git branch [branch 이름] :** 새로 만든 브랜치도 지금 작업하던 가장 마지막 커밋을 가리킴

```
$ git branch testing
```

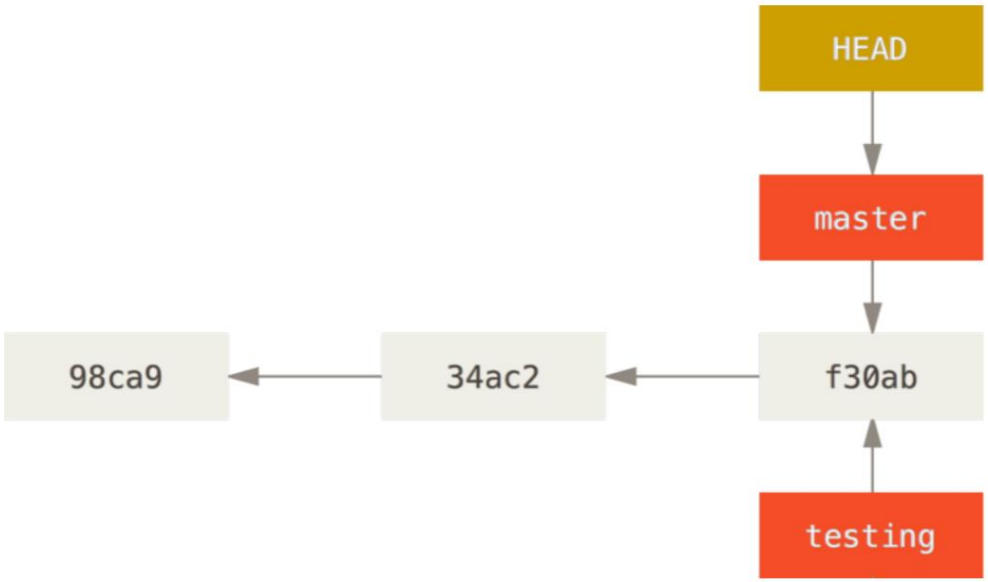



*Figure 13. 현재 작업 중인 브랜치를 가리키는 HEAD*

# **git branch** 명령은 브랜치를 만들기만 하고 브랜치를 옮기지 않는다.

# **HEAD 포인터** : 지금 작업하는 로컬 브랜치.

**git log --decorate 옵션** : 브랜치가 어떤 커밋을 가리키는지 확인할 수 있음

```
$ 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
```

- **브랜치 이동하기**

**git checkout [브랜치 이름] :** 다른 브랜치로 이동

```
$ git checkout testing
```

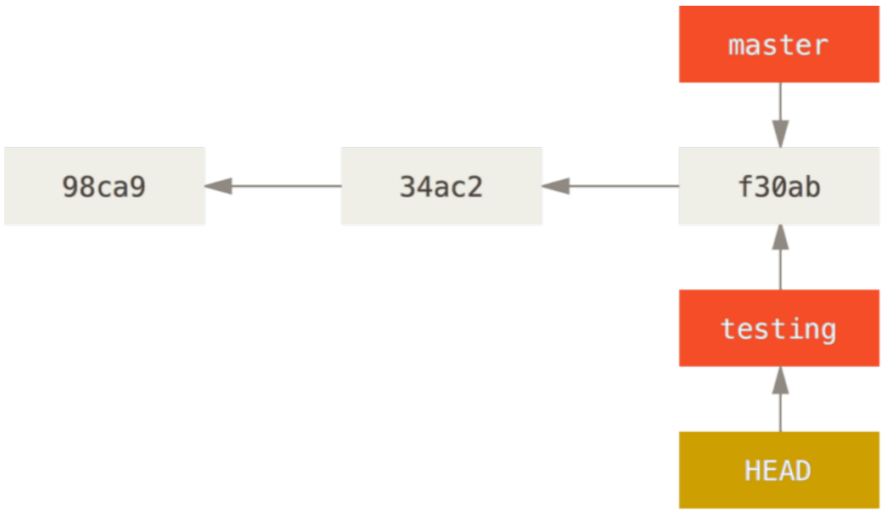



*Figure 14. HEAD는 testing 브랜치를 가리킴*

커밋 새로 실행 시 testing 브랜치는 앞으로 이동하지만 master 브랜치는 이전 커밋을 가리킴.

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

다시 master 브랜치로 돌아가기

```
$ git checkout master
```

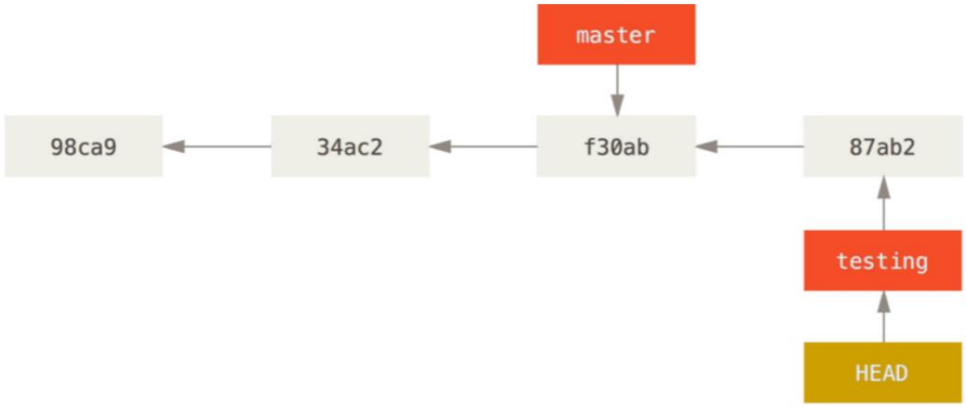


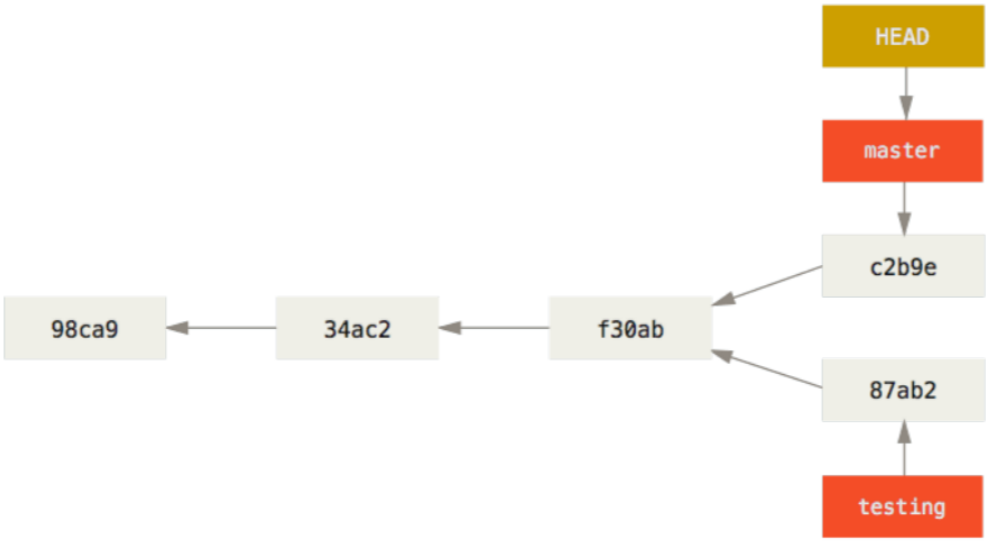

*Figure 15. HEAD가 가리키는 testing 브랜치가 새 커밋을 가리킴*

**→ master 브랜치가 가리키는 커밋을 HEAD 가 가리키게 하고, 워킹 디렉토리의 파일도 그 시점으로 되돌려 놓음.**

**※ 브랜치를 이동하면 워킹 디렉토리의 파일이 변경된다. 파일 변경시 문제가 있어 브랜치를 이동시키는게 불가능한 경우, Git 은 브랜치 이동 명령을 수행하지 않는다.**

파일 수정 후 다시 커밋

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




**→ 프로젝트 히스토리는 분리되어 진행한다. (갈라지는 브랜치)**

**git log** : 현재 가리키고 있는 브랜치의 히스토리와 어떻게 갈라져 나왔는지 보여준다.

**git log --oneline --decorate --graph --all** : 히스토리 출력

```
$ 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
```

## 2. 브랜치와 Merge의 기초

### - 브랜치의 기초

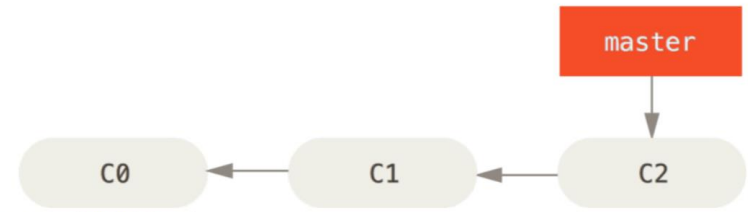



Figure 18. 현재 커밋 히스토리

**git checkout -b**: 브랜치를 만들면서 checkout까지 한 번에 하기**(=git branch + git checkout)**

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

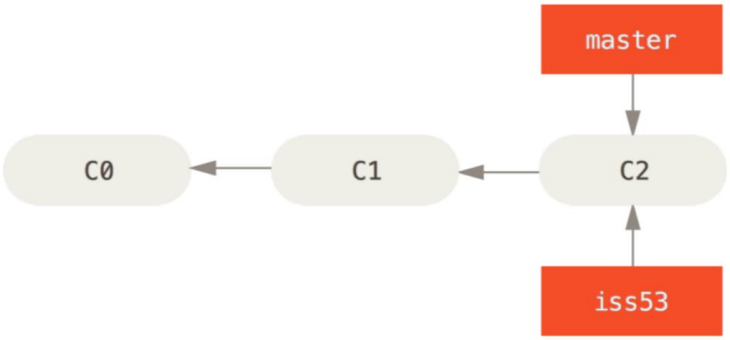



Figure 19. 브랜치 포인터를 새로 만듦

iss53 을 checkout 했기 때문에, 이후 commit 부분은 iss53 브랜치가 앞으로 나아감

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

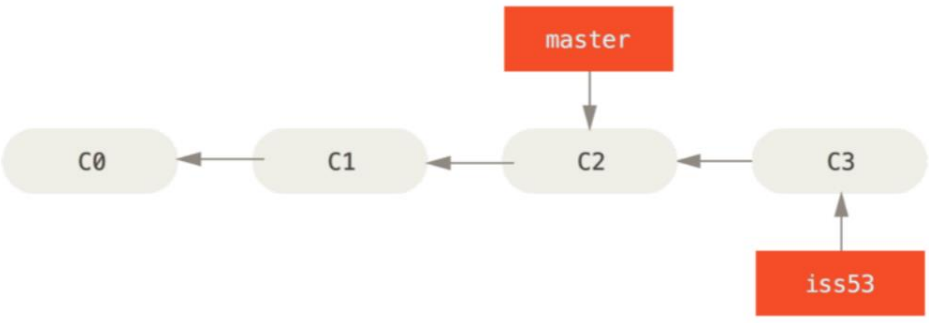



Figure 20. 진행 중인 iss53 브랜치

**※ 브랜치를 이동하기 전(master 브랜치로 돌아가기 전) 유의 :**

**아직 커밋하지 않은 파일이 Checkout할 브랜치와 충돌 나면 브랜치를 변경할 수 없다. 따라서 브랜치를 변경할 때에는 워킹 디렉토리를 정리하는 것이 좋다. (이후 Stashing과 Cleaning 다룸)**

→지금은 작업하던 것을 커밋하고 master 브랜치로 옮기는 것으로 가정

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

- 현재 사이트에 문제가 생겨 즉시 해결해야 하는데, iss53이 섞이는 것을 방지하기 위해 master 브랜치로 변경.

- Git은 자동으로 디렉토리에 파일 추가/삭제/수정하여 Checkout한 브랜치의 스냅샷으로 되돌려 놓음 유의.

```
$ git checkout -b hotfix
Switched to a new branch 'hotfix'
$ vim index.html
$ git commit -a -m 'fixed the broken email address'
[hotfix 3a0874c] fixed the broken email address
 1 files changed, 1 deletion(-)
```

- hotfix라는 브랜치를 만들고 새로운 이슈를 해결할 때까지 사용

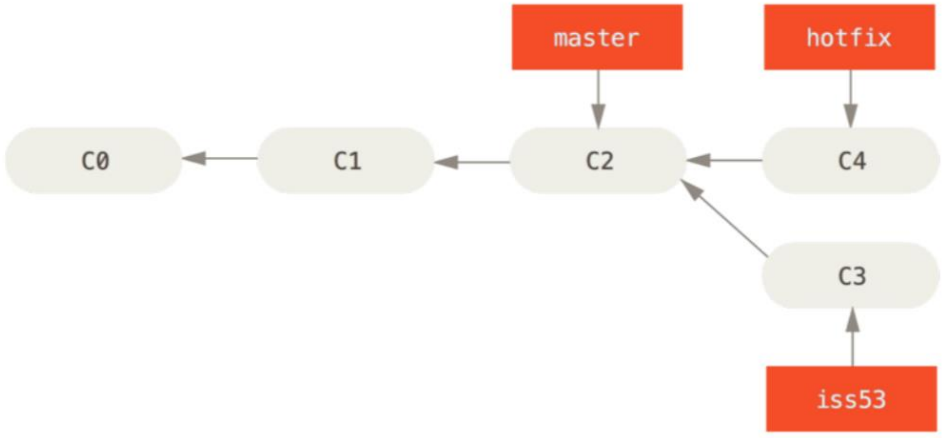



*Figure 21. master 브랜치에서 갈라져 나온 hotfix 브랜치*

**git merge [브랜치 이름]** : 현재 활성화된 브랜치에 해당 브랜치 merge

```
$ git checkout master
$ git merge hotfix
Updating f42c576..3a0874c
Fast-forward
 README | 1 -
 1 file changed, 1 deletion(-)
```

**-Fast-forward** : 현 브랜치의 앞으로 진행한 커밋일 경우, 브랜치 포인터를 최신 커밋으로 이동시킴

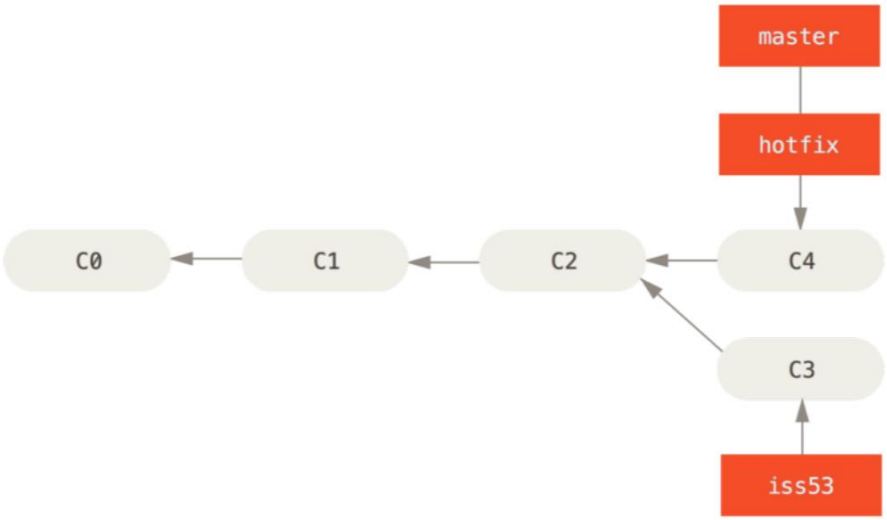



*Figure 22. Merge 후 hotfix 같은 것을 가리키는 master 브랜치*

**git branch -d [브랜치 이름]** : 해당 브랜치 삭제

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

다시 iss53 브랜치로 변경 후 커밋 진행

```
$ 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(+)
```

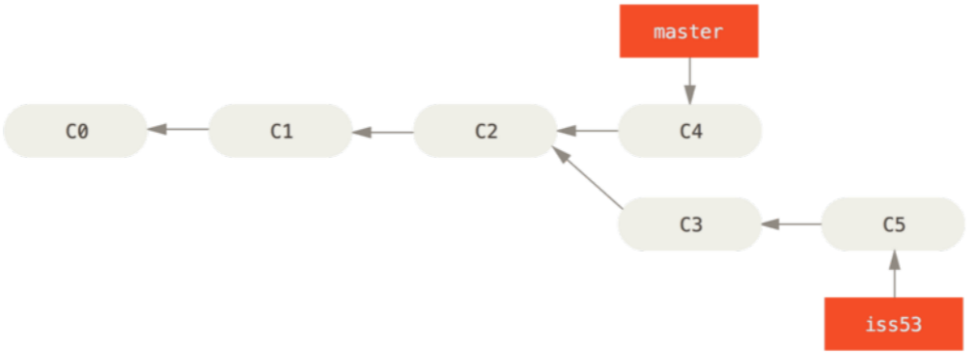



*Figure 23. master와 별개로 진행하는 iss53 브랜치*

**- Merge 의 기초**

master 브랜치로 변경 후 iss53 merge

```
$ git checkout master
$ git merge iss53
Auto-merging README
Merge made by the 'recursive' strategy.
 README | 1 +
 1 file changed, 1 insertion(+)
```

: 3-way-merge. 각 브랜치가 가리키는 커밋 두 개 + 공통 조상 하나 사용

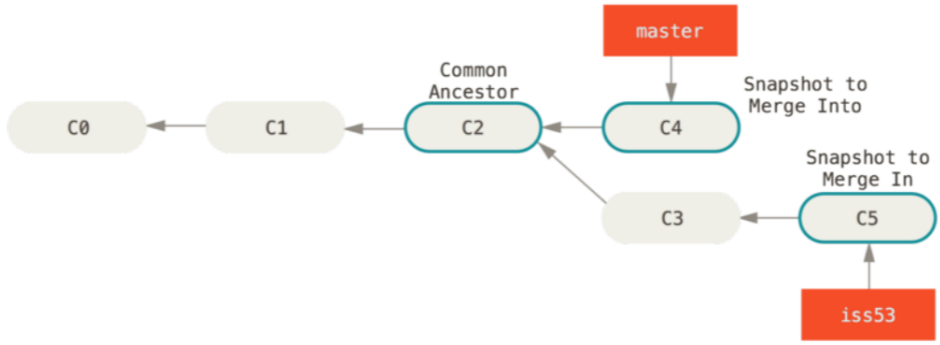



*Figure 24. 커밋 3개를 Merge*

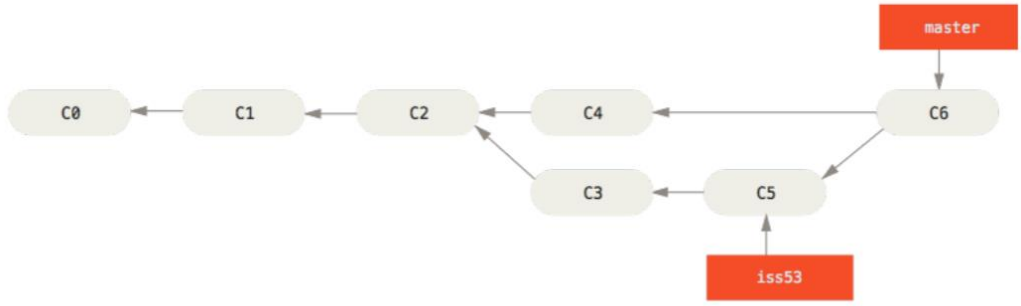



*Figure 25. Merge 커밋*

필요 없어진 iss53 브랜치 삭제, 이슈 상태를 처리 완료로 표시함.

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

# Git 은 Merge 하는데 필요한 최적의 공통 조상을 자동으로 찾는다.

**- 충돌의 기초**

- 3-way Merge 실패 : 두 브랜치에서 같은 파일의 한 부분을 동시에 수정하고 Merge 시

ex) iss53 과 hotfix 가 같은 부분을 수정했다 → 충돌 메시지 출력

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

→ 변경사항의 충돌을 개발자가 해결하지 않는 한 Merge 과정 진행 불가

**git status** 명령 이용 충돌 확인

```
$ 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 난 부분 표시

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

: HEAD-master 브랜치 // iss53 브랜치

→ 해결하려면 위쪽/ 아래쪽 내용에서 고르거나 새로 작성해서(충돌 부분 해결 후 git add) Merge

+ **git mergetool** 명령: 충돌 해결 Merge 도구

```
$ git mergetool

This message is displayed because 'merge.tool' is not configured.
See 'git mergetool --tool-help' or 'git help config' for more details.
'git mergetool' will now attempt to use one of the following tools:
```

```
opendiff kdiff3 tkdiff xxdiff meld tortoisemerge gvimdiff diffuse diffmerge
ecmerge p4merge araxis bc3 codecompare vimdiff emerge
Merging:
index.html

Normal merge conflict for 'index.html':
  {local}: modified file
  {remote}: modified file
Hit return to start merge resolution tool (opendiff):
```

**git commit** 명령으로 충돌 해결 후 Merge 한 것을 커밋한다. 커밋 메시지는 아래와 같다.

```
Merge branch 'iss53'

Conflicts:
  index.html
#
# It looks like you may be committing a merge.
# If this is not correct, please remove the file
#       .git/MERGE_HEAD
# and try again.
#
```

### 3. 브랜치 관리

**git branch :** 브랜치의 목록을 보여줌

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

* : 현재 Checkout 해서 작업하는 브랜치

**git branch -v** : 브랜치마다 마지막 커밋 메시지

```
$ git branch -v
  iss53   93b412c fix javascript issue
* master  7a98805 Merge branch 'iss53'
  testing 782fd34 add scott to the author list in the readmes
```

**git branch --merged / git branch --no-merged** : Merge 된 브랜치인지 아닌지 필터링

```
$ git branch --merged
  iss53
* master
```

* 기호가 붙어있지 않은 브랜치는 **git branch -d** 명령으로 삭제해도 되는 브랜치.

```
$ git branch --no-merged
  testing
```

아직 Merge 안 한 커밋이 있으므로 **git branch -d** 명령으로 삭제 불가. **-D** 옵션 강제 삭제 가능

## 4. 브랜치 워크플로

### - Long-Running 브랜치


*Figure 26. 안정적인 브랜치일수록 커밋 히스토리가 뒤쳐짐*

실험실에서 충분히 테스트하고 실전에 배치하는 과정으로 보면 이해하기 쉽다

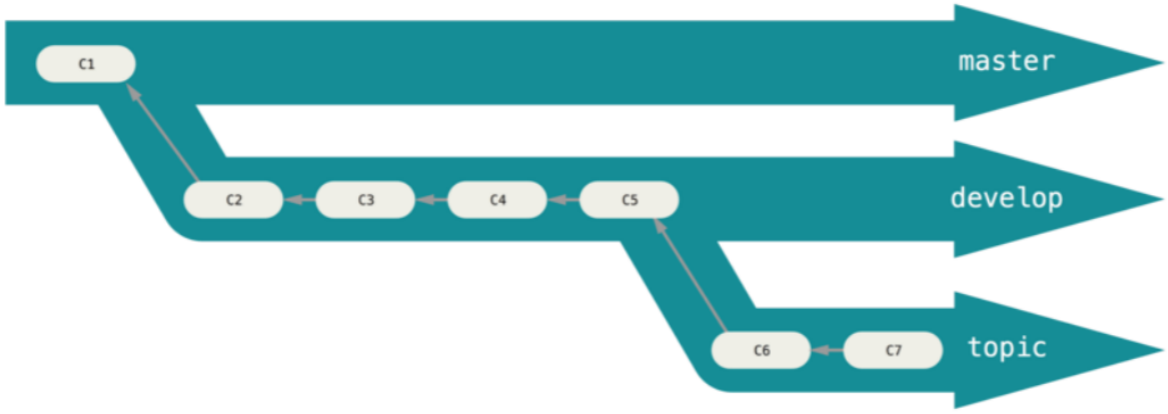



*Figure 27. 각 브랜치를 하나의 ‘`실험실''로 생각*

→ 코드를 여러 단계로 나누어 안정성을 높여가며 운영할 수 있음.

→ 규모가 크고 복잡한 프로젝트일수록 유용함.

**- 토픽 브랜치**

->한 가지 주제나 작업을 위해 만든 짧은 호흡의 브랜치.

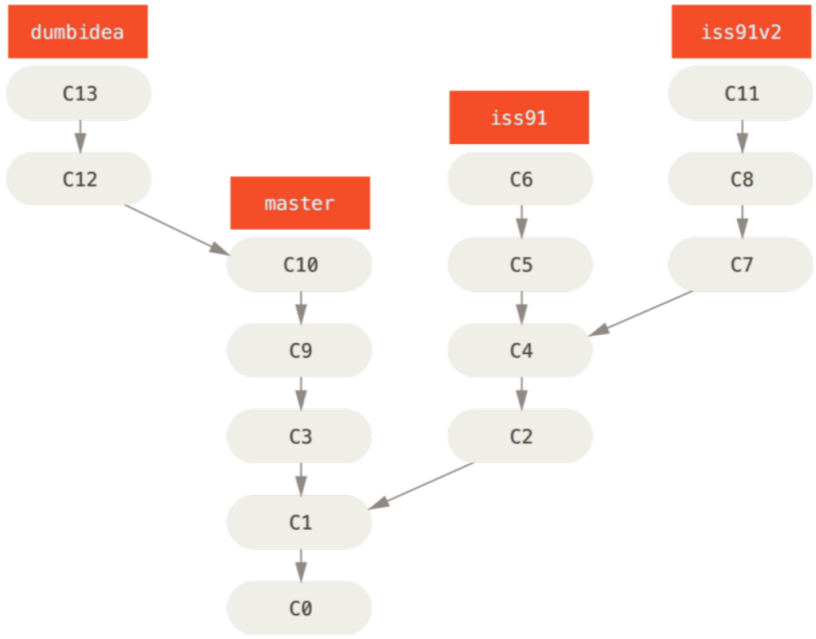



*Figure 28. 토픽 브랜치가 많음*

→ iss91v2 브랜치 + dumbidea 브랜치를 Merge

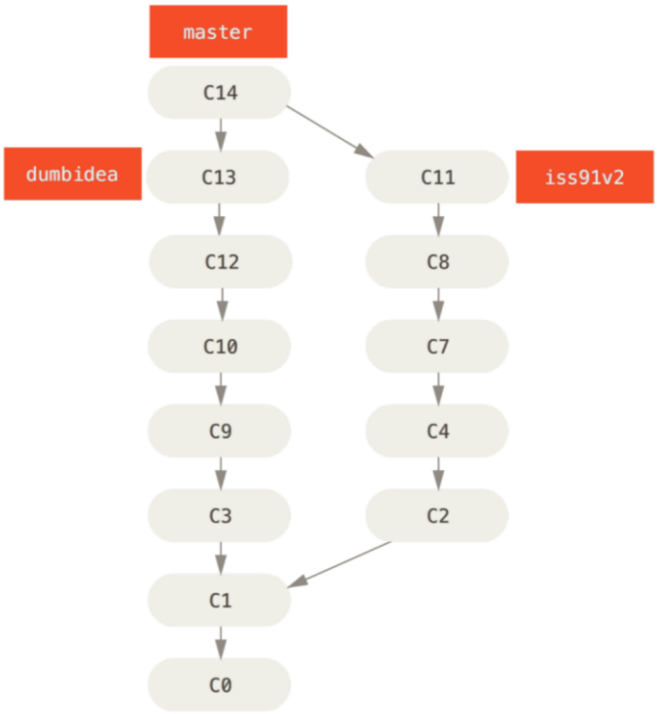



*Figure 29. dumbidea 와 iss91v2 브랜치를 Merge 하고 난 후의 모습*

**※ 여태까지의 작업은 전부 로컬에서만 처리했음. 서버와 통신을 주고 받지 않음**

## 5. 리모트 브랜치

**리모트 Refs** : 리모트 저장소에 있는 포인터인 레퍼런스(리모트 저장소에 있는 브랜치, 태그 등등)

**git ls -remote [remote]** 명령 : 모든 리모트 Refs를 조회할 수 있음

**git remote show [remote]** 명령 : 모든 리모트 브랜치와 그 정보를 보여줌

**리모트 트래킹 브랜치** : 리모트 브랜치를 추적하는 브랜치. (일종의 북마크)

→ 리모트 저장소에 마지막으로 연결했던 순간에 브랜치가 어떤 커밋을 가리키고 있었는지 나타냄.

일반적으로 리모트 브랜치 이름은 (remote) / (branch) 형식. Clone하면 자동으로 origin 이름. origin으로부터 저장소 데이터를 모두 내려받고, master 브랜치를 가리키는 포인터를 만듦. 이 포인터는 origin/master라고 부름. Git은 로컬의 master 브랜치가 origin/master를 가리키게 하고, 이 master 브랜치에서 작업을 시작할 수 있음.

**cf. git clone -o mygrit** → mygrit/master라고 사용자가 정한 대로 리모트 이름을 생성해줌.

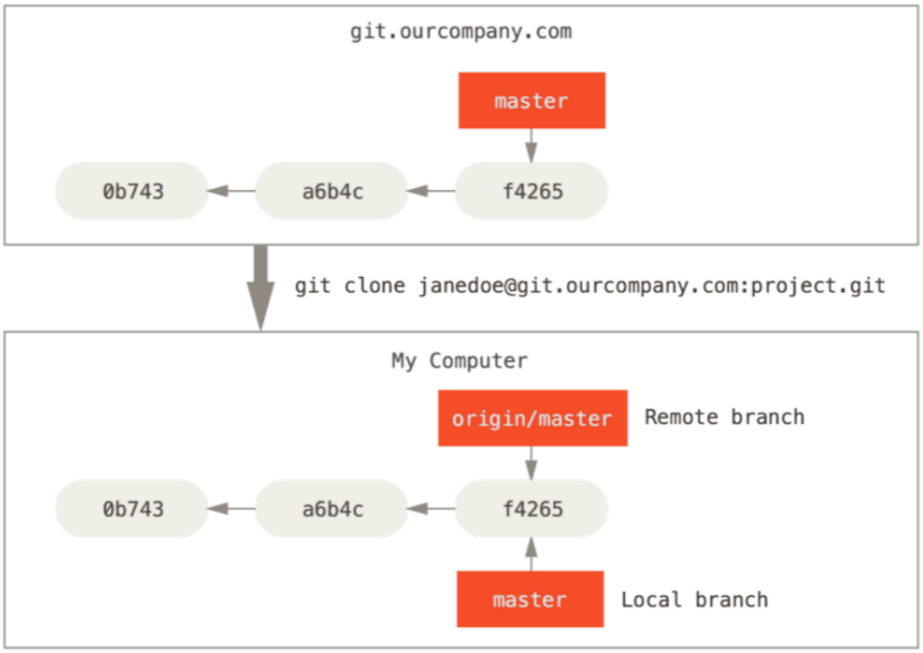



*Figure 30. Clone 이후 서버와 로컬의 master 브랜치*

다른 팀원이 서버에 Push 하고 master 브랜치를 업데이트하면 팀원 간 히스토리가 달라짐. but, origin/master 포인터는 그대로. (아직 업데이트 안 함)

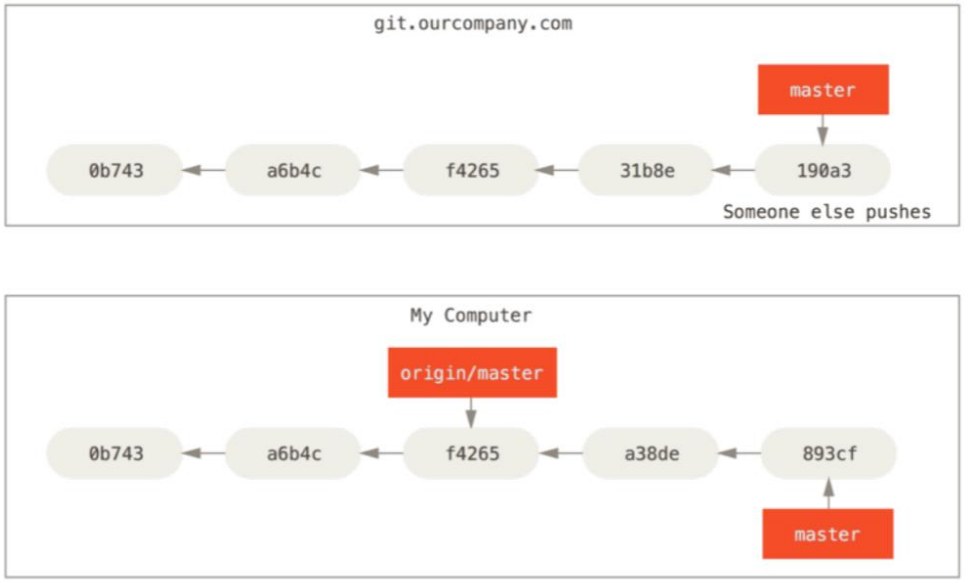



*Figure 31. 로컬과 서버의 커밋 히스토리는 독립적임*

git fetch origin : 리모트 서버로부터 저장소 정보를 동기화하기

6. Rebase 하기