[서식3] 교육자료

|  |
| --- |
| 원본 |

|  |
| --- |
| Git 교육 |

2017 SW공학기술 현장적용 지원사업

2017.07.20

에이블맥스 ㈜ 링크투 ㈜

목차

[I. 시작하기 5](#_Toc488851059)

[1. 버전 관리란? 5](#_Toc488851060)

[1.1. 로컬 버전 관리 5](#_Toc488851061)

[1.2. 중앙집중식 버전 관리(CVCS) 6](#_Toc488851062)

[1.3. 분산 버전 관리(DVCS) 6](#_Toc488851063)

[2. Git 기초 7](#_Toc488851064)

[3. CLI와 GUI 8](#_Toc488851067)

[4. Git 설치 8](#_Toc488851068)

[4.1. Linux에서 설치 8](#_Toc488851070)

[4.2. Mac에서 설치 9](#_Toc488851071)

[4.3. Windows에서 설치 9](#_Toc488851072)

[5. Git 최초 설정 9](#_Toc488851073)

[5.1. 사용자 정보 설정 10](#_Toc488851075)

[5.2. 편집기 설정 10](#_Toc488851076)

[5.3. 설정 확인 10](#_Toc488851077)

[6. 도움말 보기 11](#_Toc488851078)

[II. Git의 기초 12](#_Toc488851079)

[1. Git 저장소 만들기 12](#_Toc488851080)

[1.1. 기존디렉토리를 Git저장소로 만들기 12](#_Toc488851083)

[1.2. 기존 저장소를 Clone하기 12](#_Toc488851084)

[2. 수정하고 저장소에 저장하기 13](#_Toc488851085)

[2.1. 파일의 상태 확인하기 13](#_Toc488851088)

[2.2. 파일 새로 추적하기 14](#_Toc488851089)

[2.3. Modified 상태의 파일을 Stage하기 14](#_Toc488851090)

[2.4. 파일 상태를 짤막하게 확인하기 15](#_Toc488851091)

[2.5. 파일 무시하기 15](#_Toc488851092)

[2.6. Staged와 Unstaged 상태의 변경 내용을 보기 16](#_Toc488851093)

[2.7. 변경사항 커밋하기 16](#_Toc488851094)

[2.8. Staging Area 생략하기 16](#_Toc488851095)

[2.9. 파일 삭제하기 16](#_Toc488851096)

[2.10. 파일 이름 변경하기 17](#_Toc488851097)

[3. 커밋 히스토리 조회하기 17](#_Toc488851098)

[3.1. 조회 제한 조건 18](#_Toc488851100)

[4. 되돌리기(Undo) 18](#_Toc488851101)

[4.1. 파일 상태를 Unstage로 변경하기 18](#_Toc488851103)

[4.2. Modified 파일 되돌리기 19](#_Toc488851104)

[5. 리모트 저장소 20](#_Toc488851105)

[5.1. 리모트 저장소 확인하기 20](#_Toc488851107)

[5.2. 리모트 저장소 추가하기 20](#_Toc488851108)

[5.3. 리모트 저장소를 Pull하거나 Fetch 하기 21](#_Toc488851109)

[5.4. 리모트 저장소에 Push하기 21](#_Toc488851110)

[5.5. 리모트 저장소 살펴보기 21](#_Toc488851111)

[5.6. 리모트 저장소 이름 변경 및 삭제하기 22](#_Toc488851112)

[III. Git 브랜치 22](#_Toc488851113)

[1. 브랜치란 무엇인가 22](#_Toc488851114)

[1.1. 브랜치 생성하기 22](#_Toc488851117)

[1.2. 브랜치 이동하기 23](#_Toc488851118)

[2. 브랜치와 Merge 의 기초 24](#_Toc488851119)

[3. 브랜치 관리 26](#_Toc488851120)

[4. 리모트 브랜치 27](#_Toc488851121)

[4.1. Push하기 28](#_Toc488851125)

[4.2. Pull하기 28](#_Toc488851126)

[IV. Git 서버 28](#_Toc488851127)

[1. 프로토콜 28](#_Toc488851128)

[1.1. 로컬 프로토콜 28](#_Toc488851131)

[1.2. HTTP 프로토콜 29](#_Toc488851132)

[1.3. SSH 프로토콜 29](#_Toc488851133)

[1.4. Git 프로토콜 30](#_Toc488851134)

[2. SSH 공개키 만들기 30](#_Toc488851135)

[3. GitWeb 31](#_Toc488851136)

[4. GitLab 31](#_Toc488851137)

[5. 또 다른 선택지, 호스팅 31](#_Toc488851138)

[V. 분산 환경에서의 Git 32](#_Toc488851139)

[1. 분산 환경에서의 워크플로 32](#_Toc488851140)

[1.1. 중앙집중식 워크플로 32](#_Toc488851143)

[1.2. Intergration-Manager 워크플로 32](#_Toc488851144)

[1.3. Dictator and Lieutenants 워크플로 33](#_Toc488851145)

[2. 프로젝트에 기여하기 34](#_Toc488851146)

[3. 프로젝트 관리하기 35](#_Toc488851147)

[VI. Git 도구 35](#_Toc488851148)

[1. 리비전 조회하기 35](#_Toc488851149)

[1.1. SHA-1 줄여 쓰기 35](#_Toc488851152)

[1.2. 브랜치로 가리키기 37](#_Toc488851153)

[1.3. Reflog로 가리키기 38](#_Toc488851154)

[2. 대화형 명령 40](#_Toc488851155)

[3. Stashing과 Cleaning 42](#_Toc488851156)

[3.1. 하던일 Stash하기 42](#_Toc488851159)

[3.2. Stash를 새롭게 만들기 42](#_Toc488851160)

[3.3. Stash 적용한 브랜치 만들기 42](#_Toc488851161)

[3.4. 워킹디렉토리 청소하기 43](#_Toc488851162)

[4. 검색 43](#_Toc488851163)

[4.1. Git Grep 43](#_Toc488851165)

[4.2. Git Log 검색 44](#_Toc488851166)

[4.3. 라인 히스토리 검색 45](#_Toc488851167)

[5. 히스토리 단장하기 46](#_Toc488851168)

[5.1. 마지막커밋 수정하기 46](#_Toc488851170)

[5.2. 커밋 메시지 여러 개 수정하기 46](#_Toc488851171)

[5.3. 커밋 순서 바꾸기 49](#_Toc488851172)

[5.4. 커밋 합치기 49](#_Toc488851173)

[5.5. 커밋 분리하기 50](#_Toc488851174)

[6. Reset 명확히 알고 가기 51](#_Toc488851175)

[6.1. Reset의 역할 53](#_Toc488851177)

[7. Git으로 버그 찾기 53](#_Toc488851178)

[7.1. 파일 어노테이션 53](#_Toc488851180)

[7.2. 이진 탐색 54](#_Toc488851181)

[8. Bundle 55](#_Toc488851182)

[9. Replace 56](#_Toc488851183)

[VII. Git 따라하기 57](#_Toc488851184)

[1. Git 설치하기 57](#_Toc488851185)

[2. 저장소 만들기(init) 58](#_Toc488851186)

[3. Git이 관리할 대상으로 파일 등록하기(add & status) 59](#_Toc488851187)

[4. 버전 만들기(Commit) 60](#_Toc488851188)

[5. Stage area 62](#_Toc488851189)

[6. 변경사항 확인하기(log) 64](#_Toc488851190)

[7. 과거 버전으로 돌아가기(reset & revert) 65](#_Toc488851191)

[7.1. Reset 66](#_Toc488851200)

[7.2. Revert 67](#_Toc488851201)

[8. 브랜치 사용하기 68](#_Toc488851202)

[8.1. 브랜치 만들기 68](#_Toc488851204)

[8.2. 브랜치 정보확인하기 71](#_Toc488851205)

[8.3. 브랜치 병합하기 72](#_Toc488851206)

[8.4. 브랜치 병합 시 충돌이 발생할 경우 73](#_Toc488851207)

# 시작하기

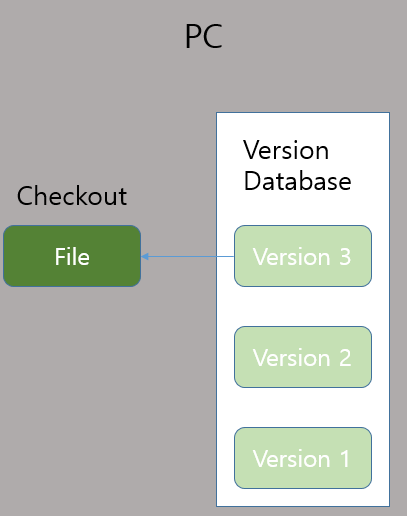
## 버전 관리란?

버전 관리란? 버전을 관리하는 것이다. 그렇다면 버전이란 무엇인가? 버전은 소프트웨어나 하드웨어 상품 개발 단계 및 순서를 번호로 표시한 것이다. 즉, 하나의 프로젝트 진행을 통하여 나온 모든 산출물 및 파일들을 모아 버전을 만들어 관리하는 것이다.

이러한 버전을 관리하는 시스템을 버전 관리 시스템(VCS – Version Control System)이라 하며, 버전 관리 시스템은 파일 변화를 시간에 따라 기록했다가 나중에 특정 시점의 버전을 다시 꺼내올 수 있는 시스템이다. 이러한 버전 관리 시스템을 사용하면 각 파일을 이전 상태로 되돌릴 수 있고, 프로젝트를 통째로 이전 상태로 되돌릴 수도 있다. 그리고 시간에 따라 수정 내용을 비교해 볼 수 있으며, ‘누가 문제를 일으켰거나 버전 생성 등’의 이력을 알 수 있다. 게다가 파일을 잃어버리거나 잘못 고쳤을 때에 이전 이력으로 되돌아가 쉽게 복구 할 수 있다. 버전 관리 방법에는 로컬 버전 관리, 중앙집중식 버전 관리, 분산 버전 관리 이와 같이 3가지 방법이 있다.

### 로컬 버전 관리

많은 사람들은 자신의 로컬 즉 PC에서 버전관리하기 위해 디렉토리로 파일을 복사하는 방법이다. 이 방법은 간단하여 자주 사용되지만, 그만큼 잘못되기 쉽다. 즉, 작업하던 디렉토리, 파일의 삭제, 잘못된 수정 및 복사 등이 일어났을 경우 원본의 안전을 위협할 수 있다. 이러한 이유로 VCS라는 것이 만들어 졌고, 이 VCS는 아주 간단한 데이터베이스를 사용하여 파일의 변경 정보를 관리한다.

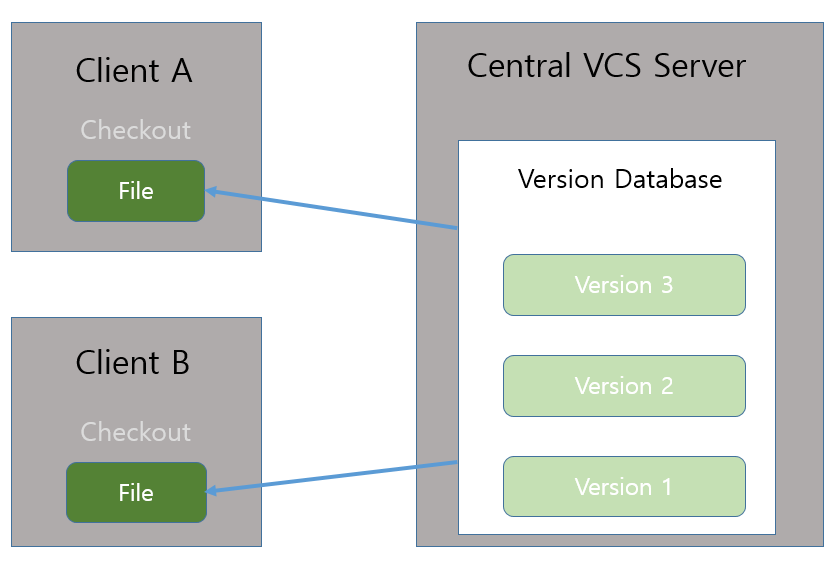


### 중앙집중식 버전 관리(CVCS)

중앙집중식 버전 관리는 다수의 사람들 즉, 클라이언트가 서버에서 관리되는 파일을 받아서 사용(Checkout)하는 것이다. 이러한 CVCS 환경은 로컬VCS에 비해 아래와 같은 장점을 가지고 있다..

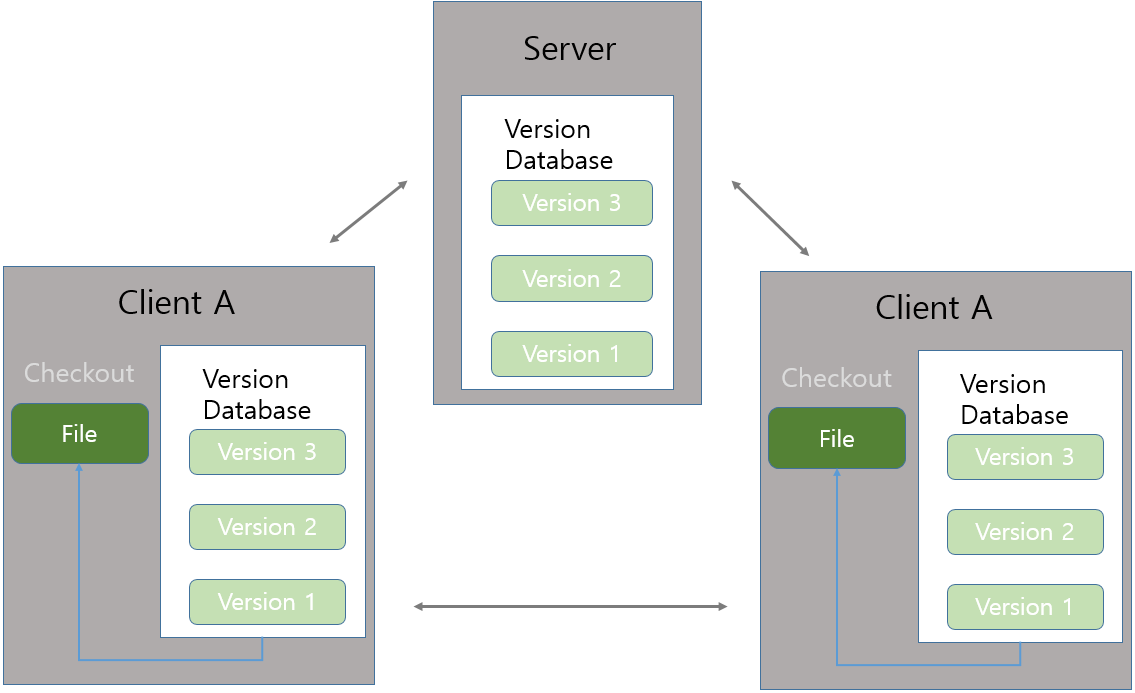
* 누가 무엇을 하고 있는지 알 수 있다.
* 관리자는 누가 무엇을 할지 관리할 수 있다. (업무 분담 및 관리 용이)
* 각 클라이언트의 로컬 데이터베이스를 관리 하는 것보다 VCS하나를 관리(서버PC) 하기 쉽다.

그러나 이러한 CVCS 환경에는 치명적인 결점이 있다. 대표적인 것이 바로 서버에 문제가 발생되었을 경우이다. 서버에 문제가 생길 경우 클라이언트간의 협업은 물론 프로젝트의 모든 작업내역을 잃을 수 있다.



### 분산 버전 관리(DVCS)

분산 버전 관리는 CVCS처럼 클라이언트가 단순히 파일의 마지막 스냅샷을 사용(Checkout)하지 않는다. 그냥 저장소를 전부 복제하여 작업에 들어간다. 그렇기 때문에 서버의 문제 발생에도 백업에 가능하다. 게다가 대부분의 DVCS 환경에서는 리모트 저장소가 존재한다. 이러한 리모트 저장소를 통해 다양한 그룹과 다양한 방법으로 협업을 할 수 있으며, 계층모델 같은 중앙집중식 시스템으로는 할 수 없는 워크플로를 다양하게 사용할 수 있다.



## Git 기초

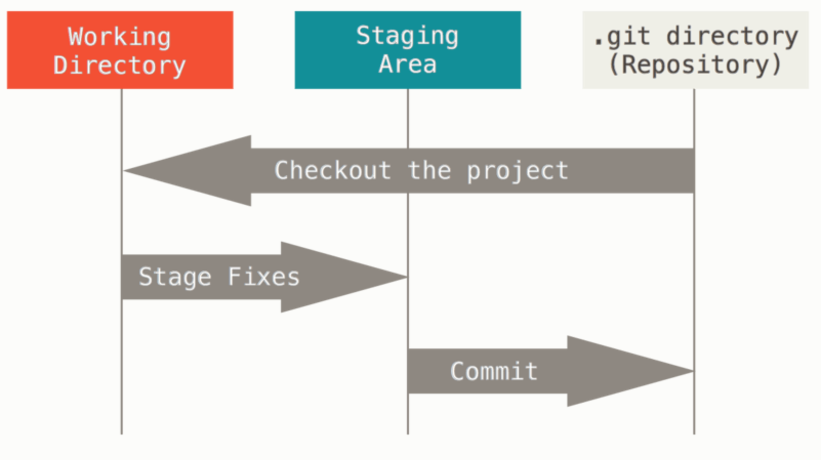
Git은 데이터를 파일 시스템 스냅샷으로 취급한다. Git은 커밋(Commit)을 하거나 프로젝트의 상태를 저장할 때마다 파일이 존재하는 그 순간을 중요하게 여긴다. 파일이 변경되지 않았으면 파일을 새로이 저장하지 않고, 이전 상태의 파일에 링크만 저장한다. 데이터를 스냅샷의 스트림처럼 취급한다. 기존 VCS는 각 파일의 변화를 시간순으로 관리하면서 파일 집합을 관리한것과 구분이 된다. 이와 같이 Git은 강력한 도구를 지원함과 동시에 작은 파일 시스템이다. 이와 같은 Git은 아래와 같은 특징을 갖는다.

* 거의 모든 명령을 로컬에서 실행한다.
* 데이터 저장하기 전에 체크섬을 구하고 그것으로 데이터를 관리하여 무결성을 갖는다.
* 데이터를 추가할 뿐이다.

또한 Git은 파일을 Committed, Modified, Staged 이렇게 세 가지 상태로 관리한다.

* Committed : 데이터가 로컬 데이터베이스에 안전하게 저장됐다는 것
* Modified : 수정한 파일을 아직 로컬 데이터베이스에 커밋하지 않은 것
* Staged : 현재 수정한 파일을 곧 커밋할 것이라고 표시한 상태

이 세 가지 상태는 Git 프로젝트의 세 가지 단계와 연결돼 있으므로 Git 디렉토리, 워킹 트리, Staging Area 이렇게 세 가지 단계에 대해 알아보자.



Git 디렉토리는 Git이 프로젝트의 메타데이터와 객체 데이터베이스를 저장하는 곳을 말한다. 이 Git 디렉토리가 Git의 핵심이다. 다른 컴퓨터에 있는 저장소를 Clone 할 때 Git 디렉토리가 만들어진다. 워킹 트리는 프로젝트의 특정 버전을 Checkout 한 것이다. Git 디렉토리는 지금 작업하는 디스크에 있고 그 디렉토리 안에 압축된 데이터베이스에서 파일을 가져와서 워킹 트리를 만든다. Staging Area는 Git 디렉토리에 있다. 단순한 파일이고 곧 커밋할 파일에 대한 정보를 저장한다. 종종 ``Index``라고 불리기도 하지만, Staging Area라는 명칭이 표준이 되어가고 있다.

Git으로 하는 일은 기본적으로 아래와 같다.

워킹 트리에서 파일을 수정한다.

Staging Area에 파일을 Stage 해서 커밋할 스냅샷을 만든다.

Staging Area에 있는 파일들을 커밋해서 Git 디렉토리에 영구적인 스냅샷으로 저장한다.

Git 디렉토리에 있는 파일들은 Committed 상태이다. 파일을 수정하고 Staging Area에 추가했다면 Staged이다. 그리고 Checkout 하고 나서 수정했지만, 아직 Staging Area에 추가하지 않았으면 Modified이다.

## CLI와 GUI

Git을 사용하는 방법에는 CLI와 GUI가 있다.

CLI는 Git의 모든 기능을 지원한다. 하지만 UI가 없으므로 Mac의 Terminal이나 Windows의 CMD나 Powershell을 통하여 명령어를 실행한다. 반면 GUI는 Git의 기능 일부를 UI로 만들어 사용하기 편하게 만들어져 있다.

## Git 설치



### Linux에서 설치

Linux에서 패키지로 Git을 설치할 때는 보통 각 배포판에서 사용하는 패키지 관리도구를 사용하여 설치한다. Fedora에서는 아래와 같이 한다.

|  |  |
| --- | --- |
| Fedora | Ubuntu등의 데비안 계열 배포판 |
| $ sudo yum install git-all | $ sudo apt-get install git-all |

다른 Unix 배포판에 설치하려면 http://git-scm.com/download/linux 에서 확인하자.

### Mac에서 설치

Mac에 Git을 설치하는 방법 중에는 Xcode Command Line Tools를 설치하는 방법이 가장 쉽다. Mavericks(10.9)부터는 Terminal에 단지 처음으로 'git’을 실행하는 것으로 설치가 시작되고, 'git’이 설치돼 있지 않으면 설치하라고 안내해준다.

좀 더 최신 버전이 필요하면 바이너리 인스톨러로 설치할 수 있다. OSX용 Git 인스톨러는 Git 웹사이트에서 관리하고 있으며 http://git-scm.com/download/mac 에서 내려받도록 하자.

### Windows에서 설치

Windows에 Git을 설치하는 방법은 여러 가지다. 공식 배포판은 Git 웹사이트에서 내려받을 수 있다. http://git-scm.com/download/win에 가면 자동으로 다운로드가 시작된다. 이 프로젝트가 'Git for Windows’인데, Git 자체와는 다른 별도의 프로젝트다. 자세한 정보는 https://git-for-windows.github.io/에서 확인하자.

## Git 최초 설정

Git의 사용 환경 설정은 Git을 업그레이드 해도 유지되므로 한번만 해주면 된다. 환경 설정은 ‘git config’라는 도구로 설정 내용을 확인 및 변경 할수 있는데, 이때 사용하는 설정 파일은 아래와 같다.

|  |  |
| --- | --- |
| 파일 | 내용 |
| /etc/gitconfig | 시스템의 모든 사용자와 모든 저장소에 적용되는 설정  git config --system 옵션으로 이 파일을 읽고 쓸 수 있다. |
| ~/.gitconfig  ~/.config/git/config | 특정 사용자에게만 적용되는 설정  git config --global 옵션으로 이 파일을 읽고 쓸 수 있다. |
| .git/config | Git 디렉토리에 있고 특정 저장소(혹은 현재 작업 중인 프로젝트)에만 적용된다. |

각 설정은 역순으로 우선시 된다.



### 사용자 정보 설정

Git을 설치하고 나서 가장 먼저 해야 하는 것이 사용자 이름과 이메일 주소를 설정하는 것이다. 이 정보는 Git이 커밋(Commit)할 때마다 사용되면 커밋된 후에는 정보를 변경할 수 없다.

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

만약 프로젝트마다 다른 이름과 이메일 주소를 사용하고 싶으면 –global 옵션을 빼고 위 명령어를 실행하면 된다.

### 편집기 설정

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

|  |  |
| --- | --- |
| Note | Vim과 Emacs, Notepad++은 꽤 인기 있는 편집기로 개발자들이 즐겨 사용한다. OS X나 Linux 같은 Unix 시스템이나 Windows 시스템에서 사용 가능하다. 이 편집기가 불편해서 다른 편집기를 사용해야 한다면 Git 편집기로 설정하는 방법을 찾아봐야 한다. |
| Warning | 자신의 편집기를 설정하지 않으면 갑자기 실행된 편집기에 당황할 수 있다. 그땐 당황하지 말고 편집기를 그냥 종료하면 Git 명령을 취소할 수 있다. |

### 설정 확인

git config --list 명령을 실행하면 설정한 모든 것을 보여준다. 그래서 바로 확인할 수 있다.

|  |
| --- |
| $ git config --list  user.name=John Doe  user.email=johndoe@example.com  color.status=auto  color.branch=auto  color.interactive=auto  color.diff=auto  ... |

Git은 같은 키를 여러 파일(/etc/gitconfig`와 `~/.gitconfig 같은)에서 읽기 때문에 같은 키가 여러 개 있을 수도 있다. 그러면 Git은 나중 값을 사용한다.

git config <key> 명령으로 Git이 특정 Key에 대해 어떤 값을 사용하는지 확인할 수 있다.

|  |
| --- |
| $ git config user.name  John Doe |

## 도움말 보기

명령어에 대한 도움말이 필요할 때 도움말을 보는 방법은 세 가지다.

|  |
| --- |
| $ git help <verb>  $ git <verb> --help  $ man git-<verb> |

예를 들어 아래와 같이 실행하면 config 명령에 대한 도움말을 볼 수 있다.

|  |
| --- |
| $ git help config |

# Git의 기초

## Git 저장소 만들기

Git의 저장소를 만드는 방법은 두 가지 있다. 첫 번째는 기존 프로젝트나 디렉토리를 Git 저장소로 만드는 방법이고, 두번째는 다른 서버에 있는 저장소를 Clone하는 방법이다.



### 기존디렉토리를 Git저장소로 만들기

기존 프로젝트를 Git으로 관리하고 싶을 때, 아래와 같이 해당 프로젝트의 디렉토리로 이동하고 ‘git init’이라는 명령을 실행한다.

|  |  |
| --- | --- |
| OS | 명령어 |
| Linux | $ cd /home/user/your\_repository(해당 디렉토리로 이동)  $ git init |
| Mac | $ cd /Users/user/your\_repository  $ git init |
| Windows | $ cd /c/user/your\_repository  $ git init |

‘git init’ 이 명령은 .git이라는 하위 디렉토리를 만드는데, 이곳에는 저장소에 필요한 파일(SKeleton)이 들어 있다. 실제로 Git이 파일을 관리하게 하려면 저장소에 파일을 추가하고 커밋을 해야 한다. 명령어는 아래와 같다.

|  |  |
| --- | --- |
| Git add | 파일을 추가한다. |
| Git commit | 추가된 파일을 등록한다. |
| $ git add \*.c  $ git add LICENSE  $ git commit -m 'initial project version' | |

### 기존 저장소를 Clone하기

다른 프로젝트에 참여하려거나(Contribute) Git 저장소를 복사하고 싶을 때에는 ‘git clone’ 명령을 사용한다. 이 명령어를 실행하면 해당 프로젝트의 히스토리를 전부 받아온다.

예를 들어 ibgit2 라이브러리 소스코드를 Clone 하려면 아래과 같이 실행한다.

|  |
| --- |
| $ git clone https://github.com/libgit2/libgit2 |

이 명령은 ‘`libgit2’이라는 디렉토리를 만들고 그 안에 .git 디렉토리를 만든다. 그리고 저장소의 데이터를 모두 가져와서 자동으로 가장 최신 버전을 Checkout 해 놓는다. libgit2 디렉토리로 이동하면 Checkout으로 생성한 파일을 볼 수 있고 당장 하고자 하는 일을 시작할 수 있다.

만약 Clone시 다른 이름의 디렉토리로 하려면 가장 뒤에 지정할 디렉토리명을 적어주면 된다.

|  |
| --- |
| $ git clone https://github.com/libgit2/libgit2 mylibgit |

## 수정하고 저장소에 저장하기

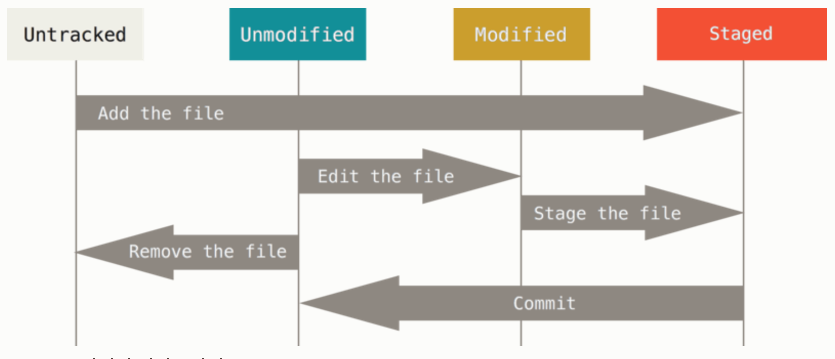
여기서는 파일을 수행하고 파일의 스냅샷을 커밋하는 것을 알아보자.

워킹 디렉토리의 모든 파일은 크게 Tracked(관리대상)와 Untracked(비관리대상)로 나뉜다.

Tracked 파일은 이미 스냅샷에 포함된 파일이다. 이러한 Tracked파일은 또 Unmodified(수정하지 않음), Modified(수정함), Staged(커밋으로 저장소에 기록할 상태)와 같이 3가지 상태중 하나로 되어있다.

Untracked 파일은 워킹 디렉토리에 있는 파일중 스냅샷에도 Staging Area에도 포함되지 않은 파일다.

* [참고] 파일의 라이플 사이클





### 파일의 상태 확인하기

파일의 상태를 확인하기 위해선 ‘git status’ 명령어를 사용한다. 예를 들어 아래와 같이 test 파일을 만들고 ‘git status’를 실행하여 보자.

|  |
| --- |
| $ echo 'My Project' > TEST  $ git status  On branch master  Your branch is up-to-date with 'origin/master'.  Untracked files:  (use "git add <file>..." to include in what will be committed)  TEST  nothing added to commit but untracked files present (use "git add" to track) |

위와 같이 처음 생성된 파일은 Untracked files 부분에 속하며, 이 파일들은 아직 스냅샷(커밋)에 넣어지지 않은 파일이다. 이러한 파일들은 직접 Tracked 상태로 변경해 줘야 하는데 그것은 다음단계에서 알아보자.

### 파일 새로 추적하기

위와 같이 Untracked 파일들은 ‘git add’ 명령어를 통하여 파일을 새로 추적(Tracked)할 수 있다. 위 생성된 TEST 파일을 ‘git add TEST’하여 주고 ‘git status’명령을 다시 실행하면 아래와 같이 Tracked 상태이면서 Staged상태라는 것을 확인할 수 있다.

|  |
| --- |
| $ git add TEST  $ 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: TEST |

‘git add’ 명령은 파일 또는 디렉토리의 경로를 아규먼트로 받는다. 만약 아규먼트가 디렉토리면 해당 디렉토리 아래에 있는 모든 파일들까지 재귀적으로 추가한다.

### Modified 상태의 파일을 Stage하기

Tracked의 Stage상태의 파일을 수정하면 해당 파일은 Modified 상태로 된다. 이러한 상태에서는 커밋시에는 해당 파일이 마지막으로 Stage상태로 있었을 때의 상태로 커밋이 된다. 즉, 수정한 부분이 업데이트가 되지 않는 것이다. 그렇기 때문에 수정한 후에는 Staged 상태로 만들어야 한다.

수정된 파일을 Staged상태로 바꾸는 방법은 위와 같이 ‘git add’명령어를 통하여 하면 된다.

|  |
| --- |
| $ vim TEST  $ 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: TEST  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: TEST |

### 파일 상태를 짤막하게 확인하기

‘git status –s’ 또는 ‘git status --short’ 와 같이 옵션을 주면 좀더 간단하게 상태 내용을 확인할 수 있다.

### 파일 무시하기

모든 파일이 git이 관리할 필요는 없을 것이다. 보통 로그 파일이나 자동으로 생성된 파일들이 그러할 것이다. 이러한 파일들을 무시하기 위해서는 ‘.gitignore’파일을 만들고 그 안에 무시할 파일의 패턴을 적는다. 예를 들어 확장자가 “.o”나 “.a”인 파일이나 “~”으로 끝나는 모든 파일을 무시하게 하려면 아래와 같이 하면 된다.

|  |
| --- |
| $ cat .gitignore  \*.[oa]  \*~ |

/gitignore 파일에 입력하는 패턴은 아래와 같은 규칙을 따른다.

* 아무것도 없는 라인이나, `#`로 시작하는 라인은 무시한다.
* 표준 Glob 패턴을 사용한다.
* 슬래시(/)로 시작하면 하위 디렉토리에 적용되지(Recursivity) 않는다.
* 디렉토리는 슬래시(/)를 끝에 사용하는 것으로 표현한다.
* 느낌표(!)로 시작하는 패턴의 파일은 무시하지 않는다.

|  |
| --- |
| # 확장자가 .a인 파일 무시  \*.a  # 윗 라인에서 확장자가 .a인 파일은 무시하게 했지만 lib.a는 무시하지 않음  !lib.a  # 현재 디렉토리에 있는 TODO파일은 무시하고 subdir/TODO처럼 하위디렉토리에 있는 파일은 무시하지 않음  /TODO  # build/ 디렉토리에 있는 모든 파일은 무시  build/  # doc/notes.txt 파일은 무시하고 doc/server/arch.txt 파일은 무시하지 않음  doc/\*.txt  # doc 디렉토리 아래의 모든 .pdf 파일을 무시  doc/\*\*/\*.pdf |

### Staged와 Unstaged 상태의 변경 내용을 보기

단순히 파일이 변경되었다는 사실이 아니라 어떤 내용이 변경되었는지 살펴보려면 ‘git diff’ 명령을 사용한다. 이 명령은 Patch처럼 어떤 라인을 추가했고 삭제했는지 알려준다. 해당 명령은 Unstaged 상태인 것들만 보여주므로, Staged 상태인 파일을 확인할 때에는 ‘—staged’와 ‘—cached’ 같은 옵션을 이용한다.

### 변경사항 커밋하기

변경사항을 커밋하기 위해서는 해당 파일의 상태를 Staged상태로 변경 시켜 준 뒤 ‘git commit’명령어를 이용하여 커밋한다.

### Staging Area 생략하기

Staging Area는 커밋할 파일을 정리한다는 점에서 매우 유용하지만 복잡하기만 하고 필요하지 않은 때도 있다. 아주 쉽게 Staging Area를 생략할 수 있다. 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.

### 파일 삭제하기

Git에서 파일을 제거하려면 ‘git rm’ 명령어를 통하여

Tracked 상태의 파일을 삭제한 후에(정확하게는 Staging Area에서 삭제하는 것) 커밋해야 한다.

|  |  |
| --- | --- |
| Git 명령을 사용하지 않고 단순히 워킹 디렉터리에서 파일을 삭제 | git rm 명령을 실행하여 파일 삭제 |
| $ 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 grit.gemspec  rm 'grit.gemspec'  $ git status  On branch master  **Changes to be committed:**  (use "git reset HEAD <file>..." to unstage)  deleted: grit.gemspec |

이 명령은 워킹 디렉토리에 있는 파일도 삭제하기 때문에 실제로 파일도 지워진다.

### 파일 이름 변경하기

파일 이름을 변경하는 명령어는 ‘git mv’ 이다. 이 명령은 일종의 단축 명령어이다. 이 명령으로 파일이름을 바꿔도 되고 mv 명령으로 파일이름을 직접 바꿔도 된다. 하지만 mv 명령으로 이름을 변경하고 나서는 꼭 rm/add 명령을 실행해야 한다.

|  |  |
| --- | --- |
| $ git mv README.md README | $ mv README.md README  $ git rm README.md  $ git add README |

## 커밋 히스토리 조회하기

해당 저장소의 히스토리를 보고 싶을때가 있을 것이다. 그럴때 사용하는 명령어가 바로 ‘git log’이다. 해당 명령어를 실행하면 저장소의 커밋 히스토리를 최근 시간순으로 정보들을 보여준다. 정보들에는 커밋의 SHA-1 체크섬, 사용자 정보(이름, 이메일), 커밋한 날짜, 커밋 메시지가 있다. 그리고 ‘git log’ 명령에는 다양한 옵션들이 아래와 같이 있다.

|  |  |  |
| --- | --- | --- |
| 옵션 | 내용 | 사용방법 |
| -p | 각 커밋의 diff 결과를 보여줌 | $git log –p |
| -2 | 최근 두개의 결과만 보여줌 | $git log –p -2 |
| --stat | 각 커밋의 통계 정보를 조회  (어떤 파일이 수정되었는지, 얼마나 많은 파일이 변경 되었는지, 얼마나 많은 라인을 추가 삭제 했는지를 보여줌) | $git log --stat |
| --shortstat | --stat 명령의 결과중에서 수정한 파일, 추가된 라인, 삭제된 라인만 보여준다. | $git log --shortstat |
| --pretty | 히스토리 내용을 보여줄 때 기본 형식 이외의 형식으로 보여줌   * Oneline – 커밋을 한줄로 보여줌 * Short, full, fuller – 정보를 조금씩 가감해서 보여줌 | $git log –pretty=online |
| --graph | 브랜치와 머지 히스토리를 보여주는 아스키 그래프를 출력 | $git log --graph |



### 조회 제한 조건

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

## 되돌리기(Undo)

Git을 사용하면 어떠한 것을 되돌리고 싶을 때 되돌릴 수 있다. 하지만 되돌리면 복구할 수 없기에 주의해야 한다. 커밋의 옵션중 하나로 –amend 옵션을 확인해 보자.

--amend 옵션은 너무 일찍 커밋했거나 어떠한 파일을 빼먹었을 때, 혹은 커밋 메시지를 잘못 적었을때 사용한다. 즉, --amend를 사용한 커밋이 이전 커밋을 덮어쓰는 것이다.



### 파일 상태를 Unstage로 변경하기

파일을 여러 개 수정하고서 따로따로 커밋하려고 했지만, 한번에 커밋을 했을 경우가 있을 것이다. 이러할 때 Staging Area에 들어가 있는 파일 중 일부는 Unstage 하여야 할 것이다. 이때 바로 ‘git reset’명령을 사용한다. 하지만 git reset 명령은 매우 위험하기에 정확히 알아보고 사용해야 한다.

Tip) Git status 명령어를 사용하면 되돌릴수 있는 방법을 알려준다.

|  |
| --- |
| $ 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 |

### Modified 파일 되돌리기

파일을 수정하다가 잘못 수정했을 때, 최근 커밋한 상태로 돌아가고 싶을 경우가 있을 것이다. 이럴때 역시 git status 명령을 사용하면 친철하게 알려준다.

|  |
| --- |
| $ 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 |

‘git checkout --<file>’을 하면 수정한 파일을 되돌려 준다. 하지만 이 명령은 꽤 위험한 명령이다. 원래 파일로 덮어쓰기 때문에 수정한 내용이 전부 사라진다. 그러므로 수정한 내용이 정말 필요 없을 경우에만 사용하도록 하자.

## 리모트 저장소

리모트 저장소란? 인터넷이나 네트워크 어딘가에 있는 저장소, 즉 외부 저장소를 뜻한다. 이 리모트 저장소가 있어야 다른 사람과의 협업이 원활할 것이다. 그렇다면 리모트 저장소를 관리한다는 것은 무엇인가? 바로 저장소를 추가, 삭제하는 것뿐만 아니라 브랜치를 관리하고 추적할지 말지 등을 관리하는 것이다.

아래에서 리모트 저장소 과니 방법에 대해 알아보자.



### 리모트 저장소 확인하기

‘git remote’ 명령을 통해 현재 프로젝트에 등록된 리모트 저장소를 확인할 수 있다. 이 명령은 리모트 저장소의 단축 이름을 보여준다.

|  |
| --- |
| $git remote  origin |

만약 URL을 함께 보고 싶을 경우에는 ‘-v’옵션을 주면 된다.

|  |
| --- |
| $ git remote -v  origin https://github.com/test(fetch)  origin https://github.com/test (push) |

### 리모트 저장소 추가하기

기존 워킹디렉토리에 새 리모트 저장소는 ‘git remote add [단축이름] [URL]’ 명령을 통하여 쉽게 추가할 수 있다.

|  |
| --- |
| $ git remote  origin  $ git remote add test2 https://github.com/test2  $ git remote -v  origin https://github.com/schacon/ticgit (fetch)  origin https://github.com/schacon/ticgit (push)  test2 https://github.com/test2 (fetch)  test2 https://github.com/test2 (push) |

### 리모트 저장소를 Pull하거나 Fetch 하기

리모트 저장소에서 데이터를 가져오려만 ‘git fetch [remote-name]’ 명령을 사용하면 된다. 그러면 리모트 저장소의 모든 브랜치를 로컬에서 접근할 수 있기 때문에 언제든 Merge를 하거나 내용을 살펴볼 수 있다. 하지만 이 명령은 데이터를 가져오지만 Merge는 하지 않는다. 즉, 작업을 하고 수동으로 Merge해야한다.

반면 ‘git pull’명령은 리모트 저장소 브랜치에서 데이터를 가져올 뿐만 아니라 자동으로 로컬 브랜치와 Merge 시킨다.

### 리모트 저장소에 Push하기

‘git push [리모트 저장소 이름] [브랜치 이름]’ 명령은 프로젝트에서 작업한 내역을 리모트 저장소에 공유 즉, 업로드 할 때 사용한다. 이때 주의 해야 할 점은 리모트 저장소의 내역이 바뀌지 않았을때 가능하다는 것이다. 만약 다른사람이 Push를 하였다면 Push하기전 다른 사람이 작업한 것을 가져와서 Merge 한 후에 Push를 하여야 가능하다.

### 리모트 저장소 살펴보기

‘git remote show [리모트 저장소 이름]’ 명령은 리모트 저장소의 구체적인 정보를 확인 할 수 있게 해준다.

|  |
| --- |
| $ 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 저장소 이름 변경 및 삭제하기

‘git remote rename’ 명령을 통해 리모트 저장소 이름을 변경할 수 있다.

예로 test를 test2로 변경하여 보자.

|  |
| --- |
| $ git remote  origin  test  $ git remote rename test test2  $ git remote  origin  test2 |

만약 리모트 저장소를 삭제해야 한다면 ‘git remote remove’나 ‘git remote rm’ 명령을 사용하면 된다. 위에서 변경한 test2를 삭제하여 보자

|  |
| --- |
| $git remote remove test2  $git remote  origin |

# Git 브랜치

## 브랜치란 무엇인가

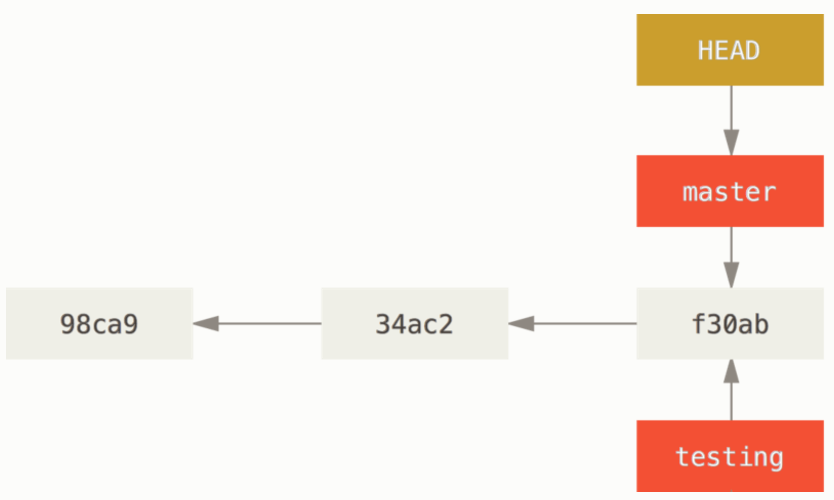
브랜치(Branch)란 가지라는 뜻으로 작업을 분기해서 처리하는 경우를 뜻한다. 이러한 브랜치는 Git에서 커밋 사이를 가볍게 이동할 수 있는 어떤 포인터 같은 것으로 이해하면 된다.



### 브랜치 생성하기

‘git branch’명령을 통해 브랜치를 생성할 수 있다.

|  |
| --- |
| $git branch testing |

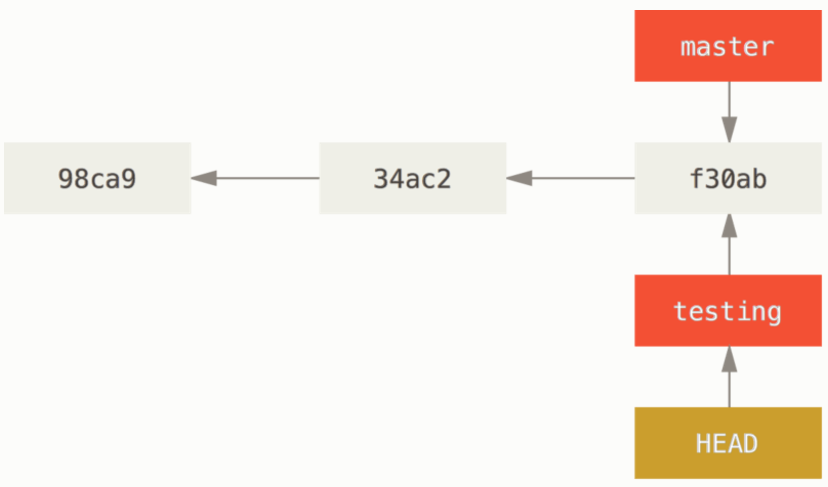


Git은 위 그림과 같이 ‘HEAD’라는 특수 포인터를 통해 현재 작업하고 있는 로컬 브랜치를 가리킨다.

### 브랜치 이동하기

‘git checkout [브랜치명]’ 명령을 통해 지정한 브랜치로 이동할 수 있다.

|  |
| --- |
| $git checkout testing |



## 브랜치와 Merge 의 기초

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

작업 중인 프로젝트가 있다.

새로운 이슈를 처리할 새 Branch를 하나 생성한다.

새로 만든 Branch에서 작업을 진행한다.

이때 중요한 문제가 생겨서 그것을 해결하려면 새로운 브랜치를 먼저 만들어야 한다. 그러면 아래와 같이 할 수 있다.

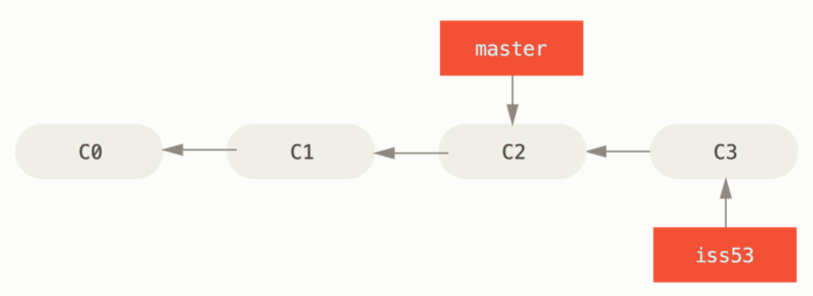
새로운 이슈를 처리하기 이전의 운영(Production) 브랜치로 이동한다.

브랜치를 새로 하나 생성한다.

수정한 브랜치 테스트를 마치고 운영 브랜치로 Merge 한다.

다시 작업하던 브랜치로 옮겨가서 하던 일 진행한다.

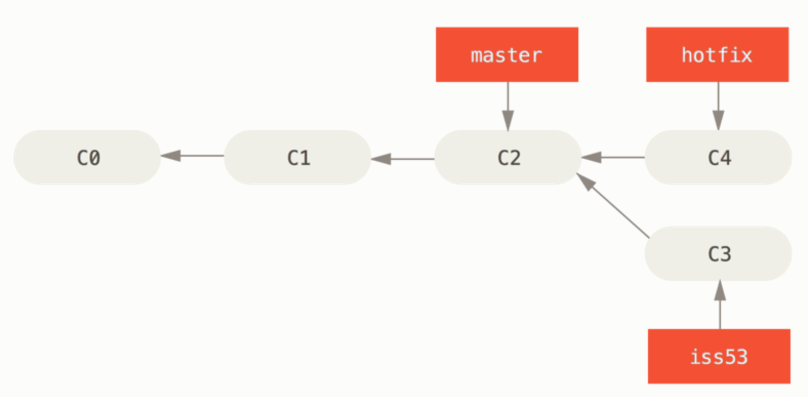
이러한 것을 다음 그림을 통해서 알아보자.



위와 같은 상태에서 master브랜치 작업 중 새로운 이슈가 생겼을 경우 이것을 해결하기 위해 hotfix브랜치를 생성한다.

|  |
| --- |
| $ 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(+) |

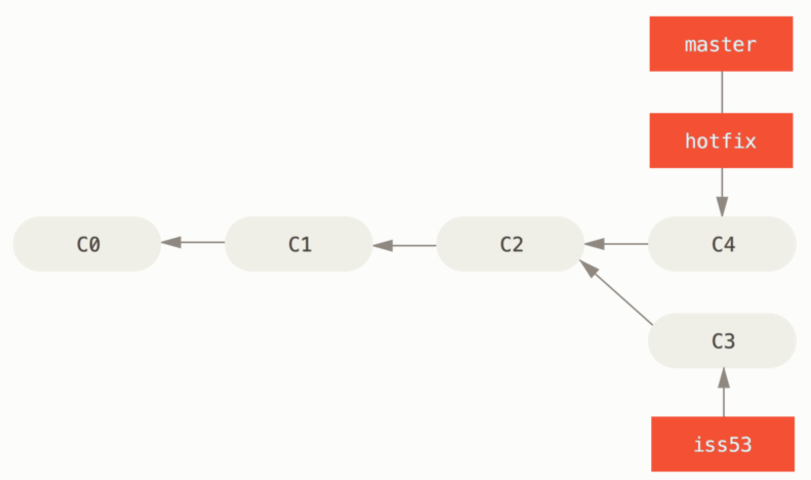
위처럼 index.html을 수정하여 이슈 해결 후 커밋을 하면 아래와 같이 되었을 것이다.



위처럼 이슈 문제를 제대로 고치고 테스트를 하면 해당 브랜치는 master 브랜치에 합쳐야 할 것이다. 이때 사용하는 명령어가 바로 ‘git merge [대상 브랜치 명]’ 이다.

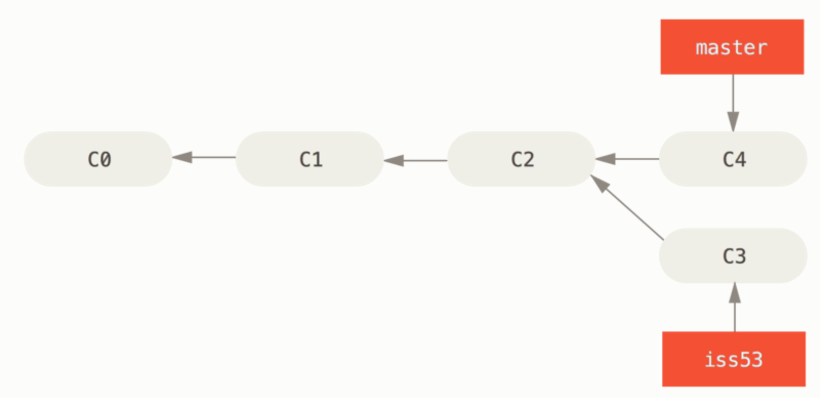
|  |
| --- |
| $ git checkout master  $ git merge hotfix  Updating f42c576..3a0874c  Fast-forward  index.html | 2 ++  1 file changed, 2 insertions(+) |

위처럼 merge를 하면 아내와 같은 그림 상태가 될것이다.



이제 이슈를 수정한 horfix브랜치는 더 이상 필요가 없음으로 삭제를 하도록 하자. 이때 사용되는 명령이 ‘git branch –d [브랜치 명]’ 이다. 이 명령을 사용하면 아래와 같이 해당 브랜치는 삭제가 되고 그림과 같은 상태가 된다.

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



## 브랜치 관리

브랜치를 관리하는데 필요한 명령에 대해 알아보자.

‘git branch’ 명령을 옵션 없이 사용하면 아래와 같이 브랜치의 목록을 보여준다.

|  |
| --- |
| $ git branch  tester  \* master  testing |

위처럼 \*가 앞에 붙은 브랜치는 현재 checkout하여 작업중인 브랜치를 나타낸다.

만약 ‘git branch’에 ‘-v’옵션을 실행하면 아래와 같이 브랜치마다 마지막 커밋 메시지도 함께 보여준다.

|  |
| --- |
| $ git branch -v  Tester 93b412c fix javascript issue  \* master 7a98805 Merge branch ‘Tester’  testing 782fd34 add scott to the author list in the readmes |

거기에 checkout한 브랜치 기준으로 각 브랜치들이 어떤 상태인지 확인할 수 있는 옵션이 있다.

‘—merged’ 와 ‘—no-merged’이다. ‘—merged’ 는 이미 Merge한 브랜치 목록을, ‘—no-merged’는 Merge하지 않은 브랜치 목록을 보여준다.

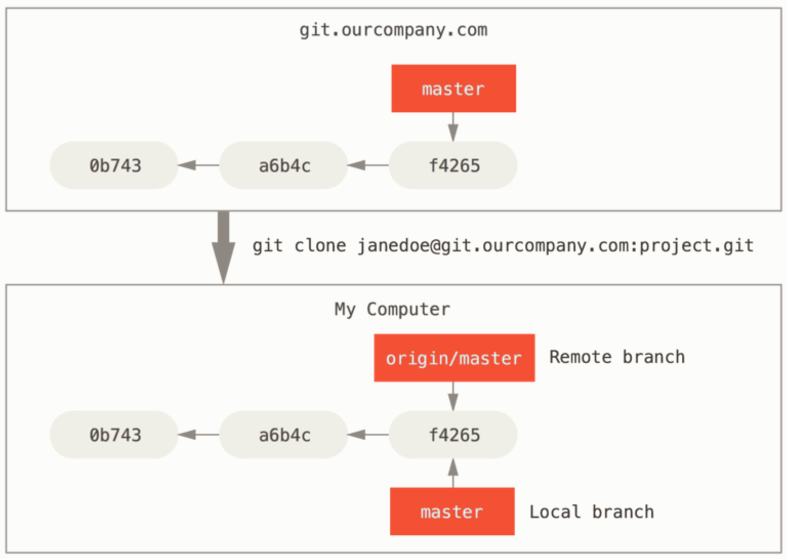
## 리모트 브랜치

리모트 Refs는 리모트 저장소에 있는 포인터인 레퍼런스다. 리모트 저장소에 있는 브랜치, 태그, 등등을 의미한다. ‘git ls-remote [remote]’ 명령으로 모든 리모트 Refs를 조회할 수 있다. ‘git remote show [remote]’ 명령은 모든 리모트 브랜치와 그 정보를 보여준다. 리모트 Refs가 있지만 보통은 리모트 트래킹 브랜치를 사용한다.

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

리모트 브랜치의 이름은 (remote)/(branch) 형식으로 되어 있다. 예를 들어 리모트 저장소 origin`의 `master 브랜치를 보고 싶다면 origin/master`라는 이름으로 브랜치를 확인하면 된다. 다른 팀원과 함께 어떤 이슈를 구현할 때 그 팀원이 `iss53 브랜치를 서버로 Push 했고 당신도 로컬에 iss53 브랜치가 있다고 가정하자. 이때 서버의 iss53 브랜치가 가리키는 커밋은 로컬에서 `origin/iss53`이 가리키는 커밋이다.

다소 헷갈릴 수 있으니 예제를 좀 더 살펴보자. git.ourcompany.com이라는 Git 서버가 있고 이 서버의 저장소를 하나 Clone 하면 Git은 자동으로 origin이라는 이름을 붙인다. origin으로부터 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.





### Push하기

로컬의 브랜치를 서버로 전송하려면 쓰기 권한이 있는 리모트 저장소에 Push 해야한다.

|  |
| --- |
| $git push <remote> <branch> |

### Pull하기

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 것이다. 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.

일반적으로 fetch`와 `merge 명령을 명시적으로 사용하는 것이 pull 명령으로 한번에 두 작업을 하는 것보다 낫다.

# Git 서버

## 프로토콜

Git은 로컬(Local), HTTP, SSH, Git 4가지의 프로토콜을 사용할 수 있다.

[참고 : Bare 저장소는 일반 프로젝트에서 .git 디렉토리만 있는 저장소다.]



### 로컬 프로토콜

가장 기본적인 것이 바로 로컬 프로토콜이다. 로컬 프로토콜은 리모트 저장소가 단순히 다스크의 다른 디렉토리에 있을 때 사용되는데, 이러한 경우는 팀원들이 전부 한 시스템에 접속하여 개발하거나 NFS같은 파일시스템 공유를 하고 있을 때 사용한다.

**장점**

파일 기반 저장소의 장점은 간단하다는 것이다. 기존에 있던 네트워크나 파일의 권한을 그대로 사용하기 때문에 설정하기 쉽다. 이미 팀 전체가 접근할 수 있는 파일시스템을 가지고 있다면 저장소를 아주 쉽게 구성할 수 있다. 다른 디렉토리를 공유할 때처럼 모든 동료가 읽고 쓸 수 있는 공유 디렉토리에 Bare 저장소를 만들면 된다.

**단점**

다양한 상황에서 접근할 수 있도록 디렉토리를 공유하는 것 자체가 일반적으로 어렵다. 집에 있을 때 Push 해야 하면 리모트 저장소가 있는 디스크를 마운트해야 하는데 이것은 다른 프로토콜을 이용하는 방법보다 느리고 어렵다.

그리고 보안에 취약하다. 이 프로토콜은 저장소에 우발적인 사고가 발생하지 않도록 보호해주지 않는다. 모든 사용자는 쉘에서 "리모트" 디렉토리에 무슨 짓이든지 할 수 있다. 누군가 저장소에 침범해서 Git 내부 파일을 삭제하고 변경하지 못하도록 하는 장치가 없다.

### HTTP 프로토콜

Git은 HTTP로 통신할 때, 서로 다른 두 방법으로 HTTP를 사용할 수 있다. 1.6.6 이전 버전에서는 읽기만 가능한 단순한 방법밖에 사용할 수 없는 멍청한 HTTP프로토콜과 1.6.6 버전부터 Git 데이터를 전송할 때 SSH처럼 서로 협상하는 스마트 HTTP 프로토콜이 있다. 스마트 HTTP 프로토콜은 사용이 쉽고 기능도 좋아서 많은 사람들이 사용하고 있다.

**장점**

읽기와 쓰기에 하나의 URL만 사용하며 사용자에게 익숙한 사용자이름과 암호 방식의 인증을 사용하기때문에 사용하기 간단하다.

**단점**

HTTP나 HTTPS를 사용하도록 설정하는 것이 SSH로 설정하는 것보다 까다로운 서버가 있다. 그것 말고는 스마트 HTTP 프로토콜이 다른 프로토콜보다 못한 단점은 별로 없다.

### SSH 프로토콜

Git의 대표 프로토콜이다. SSH를 이용하면 아무런 외부 도구 없이 Git 서버를 구축할 수 있다.

SSH를 통해 Git 저장소를 Clone 하려면 `ssh://`로 시작하는 URL을 사용한다:

|  |
| --- |
| $ git clone ssh://user@server/project.git |

아래와 같은 SCP 형태의 구문으로 줄여 쓸 수도 있다.

|  |
| --- |
| $ git clone user@server:project.git |

사용자 계정을 생략할 수도 있는데 계정을 생략하면 Git은 현재 로그인한 사용자의 계정을 사용한다.

**장점**

SSH는 상대적으로 설정하기 쉬우면 보안에 안전하다. 그리고 전송시 데이터를 가능한 압축하기 때문에 효율적이다.

**단점**

익명으로 접근을 할 수 없기때문에 오픈소스 프로젝트 같은 경우에 알맞지 않다.

### Git 프로토콜

Git 프로토콜은 Git에 포함된 데몬을 사용하는 것이다. 포트는 9418이며 SSH 프로토콜과 비슷한 서비스를 제공하지만, 인증 메커니즘이 없다. 즉, 저장소는 누구나 Clone 할 수 있거나 아무도 Clone 할 수 없거나 둘 중의 하나만 선택할 수 있다.

**장점**

전송 속도가 가장 빠르다.

**단점**

인증 메커니즘이 없다.

## SSH 공개키 만들기

Git 서버들은 SSH 공개키로 인증한다. 즉, 공개키가 존재하여야 한다. 기본적으로 사용자의 SSH키들은 사용자의 ‘~/.ssh’ 디렉토리에 저장한다. 이 디렉토리를 살펴보면 id\_dsa나 id\_rsa라는 파일 이름이 보일 것이고 같은 이름에 확장자가. ‘.pub’인 파일이 있을 것이다. ‘.pub’ 확장자 파일이 공개키이고 다른 파일은 개인키 이다. 만약 파일이 없거나 ‘.ssh’ 디렉토리가 없으면 ssh-keygen이라는 프로그램으로 키를 생성하여야 한다.

|  |
| --- |
| 공개키 생성법 : ssh-keygen 실행 |
| $ ssh-keygen  Generating public/private rsa key pair.  Enter file in which to save the key (/home/schacon/.ssh/id\_rsa):  Created directory '/home/schacon/.ssh'.  Enter passphrase (empty for no passphrase):  Enter same passphrase again:  Your identification has been saved in /home/schacon/.ssh/id\_rsa.  Your public key has been saved in /home/schacon/.ssh/id\_rsa.pub.  The key fingerprint is:  d0:82:24:8e:d7:f1:bb:9b:33:53:96:93:49:da:9b:e3 schacon@mylaptop.local |
| .ssh/id\_rsa 키를 저장하고 싶은 디렉토리를 입력하고 암호를 두 번 입력한다. 이때 암호를 비워두면 키를 사용할 때 암호를 묻지 않는다. |

[참고: SSH 키를 만드는 방법이 궁금하면 https://help.github.com/articles/generating-ssh-keys에 있는 GitHub 설명서를 찾아보자]

## GitWeb

프로젝트 저장소를 단순히 읽거나 쓰는 것에 대한 설정은 다뤘다. 이제는 웹 기반 인터페이스를 설정해 보자. Git은 웹에서 저장소를 조회할 수 있는 GitWeb이라는 CGI 스크립트를 제공한다.

## GitLab

간단하게 쓰기엔 GitWeb이 꽤 좋다. 그런데 좀 더 기능이 많은 Git 서버를 쓰려면 다른 서버를 찾아 설치해야 한다. GitLab은 널리 사용하는 서버 중 하나이다.

## 또 다른 선택지, 호스팅

Git 서버를 직접 운영하기가 부담스러울 수 있다. 그런 사람들을 위해 Git 호스팅 서비스가 몇 가지 있다. 외부 Git 호스팅 서비스를 사용하면 좋은 점이 있다. 설정이 쉽고 서버 관리 비용을 아낄 수 있다. 내부적으로 Git 서버를 운영하더라도 소스 코드를 공개하기 위해 호스팅 서비스를 이용해야 할 수 있다.

# 분산 환경에서의 Git

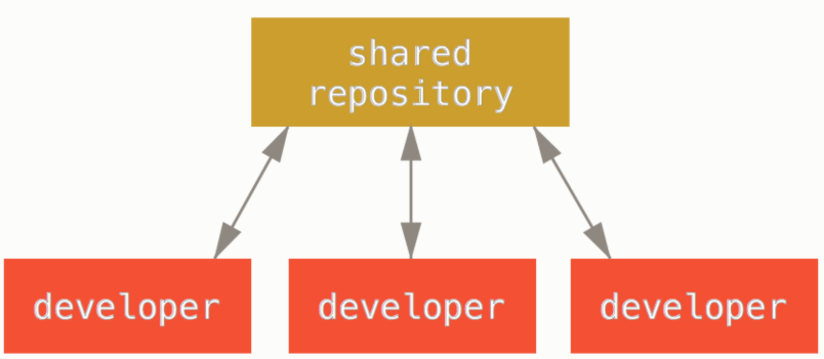
## 분산 환경에서의 워크플로

중앙집중형 버전 관리시스템과 달리 Git은 분산형이다. 그렇기에 구조가 매우 유연하며 다양한 방식으로 작업을 구성할 수 있다. 중앙집중형 버전 관리 시스템에서 개발자는 중앙 저장소를 중심으로 한 하나의 노드일 뿐이지만, git에서는 노드이며 중앙 저장소 같은 역할도 한다. 즉, 모든 개발자들이 다른 개발자으 저장소에 일한 내용을 전송하거나, 내려 받을 수 있도록 공유해준다. 그렇기에 다양한 워크플로가 존재하는데 아래에서 몇가지 방식을 보도록 하자.



### 중앙집중식 워크플로

중앙집중식 시스템에서는 보통 중앙집중식 협업 모델이라는 한 가지 방식밖에 없다. 중앙 저장소는 딱 하나 있고 변경 사항은 모두 이 중앙 저장소에 집중된다. 즉, 중앙 저장소를 하나 만들고 개발자 모두에게 Push 권한을 부여 사용하는 것이다. 모두에게 Push 권한을 부여했지만 Git은 한 개발자가 다른 개발자의 작업 내용을 덮어쓰도록 허용하지 않기 때문에 안전하다.



### Intergration-Manager 워크플로

Git을 사용하면 리모트 저장소를 여러 개 운영할 수 있다. 다른 개발자는 읽기만 가능하고 자신은 쓰기도 가능한 공개 저장소를 만드는 워크플로도 된다. 이 Worlflow에는 보통 프로젝트를 대표하는 공식 저장소가 있다. 기여자는 우선 공식 저장소를 하나 Clone 하고 수정하고 나서 자신의 저장소에 Push 한다. 그 다음에 프로젝트 Integration-Manager에게 새 저장소에서 Pull 하라고 요청한다. 그러면 그 Integration-Manager는 기여자의 저장소를 리모트 저장소로 등록하고, 로컬에서 기여물을 테스트하고, 프로젝트 메인 브랜치에 Merge 하고, 그 내용을 다시 프로젝트 메인 저장소에 Push 한다. 이런 과정은 아래와 같다.

프로젝트 Integration-Manager는 프로젝트 메인 저장소에 Push를 한다.

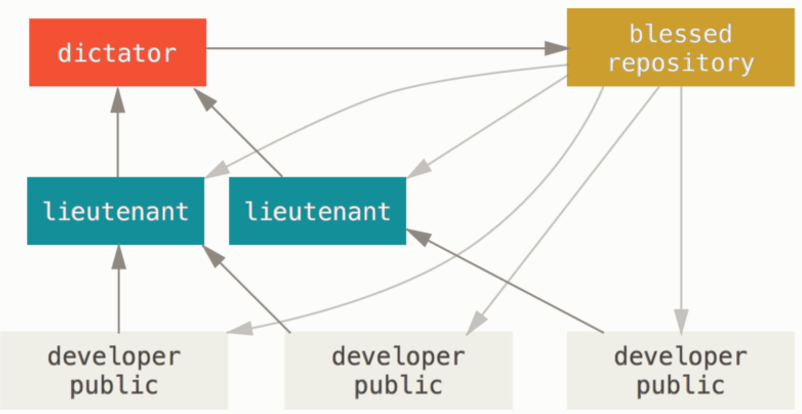
프로젝트 기여자는 메인 저장소를 Clone 하고 수정한다.

기여자는 자신의 저장소에 Push 하고 Integration-Manager가 접근할 수 있도록 공개해 놓는다.

기여자는 Integration-Manager에게 변경사항을 적용해 줄 것을 이메일로 요청한다.

Integration-Manager는 기여자의 저장소를 리모트 저장소로 등록하고 수정사항을 Merge 하여 테스트한다.

Integration-Manager는 Merge 한 사항을 메인 저장소에 Push 한다.



이 방식은 GitHub나 GitLab 같은 Hub 사이트를 통해 주로 사용하는 방식이다.

### Dictator and Lieutenants 워크플로

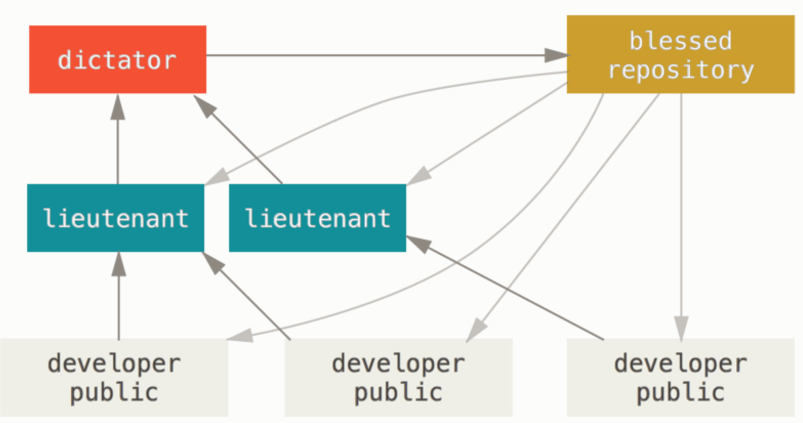
저장소를 여러개 운영하는 방식을 변형한 구조이다. 보통 수백 명의 개발자가 참여하는 아주 큰 프로젝트를 운영할 때 이 방식을 사용한다. Linux 커널 프로젝트가 대표적이다. 여러 명의 Integration-Manager가 저장소에서 자신이 맡은 부분만을 담당하는데 이들을 Lieutenants라고 부른다. 모든 Lieutenant는 최종 관리자 아래에 있으며 이 최종 관리자를 Benevolent Dictator라고 부른다. Benevolent Dictator는 Lieutenant의 저장소를 가져와 공식 저장소에 Push 하고 모든 프로젝트 참여자는 이 공식 저장소에서 반드시 Pull 해야 한다. 이러한 워크플로는 아래와 같다.

개발자는 코드를 수정하고 master 브랜치를 기준으로 자신의 토픽 브랜치를 Rebase 한다. 여기서 master 브랜치란 공식 저장소의 브랜치를 말한다.

Lieutenant들은 개발자들의 수정사항을 자신이 관리하는 master 브랜치에 Merge 한다.

Dictator는 Lieutenant의 master 브랜치를 자신의 master 브랜치로 Merge 한다.

Dictator는 자신의 master 브랜치를 Push 하며 다른 모든 개발자는 Dictator의 master 브랜치를 기준으로 Rebase 한다.



이 방식이 일반적이지 않지만 깊은 계층 구조를 가지는 환경이나 규모가 큰 프로젝트에서는 매우 쓸모 있다

## 프로젝트에 기여하기

프로젝트에 기여하는 방식은 매우 다양하다. Git이 워낙 유연하게 설계됐기 때문에 사람들은 여러 가지 방식으로 사용할 수 있다. 게다가 프로젝트마다 환경이 달라서 프로젝트에 기여하는 방식이 다양해 진다. 프로젝트에 기여하는 방식에 영향을 끼치는 몇 가지 변수(활발히 기여하는 개발자의 수, 선택한 워크플로의 종류, 접근 권한, 프로젝트의 외부 공개 및 비공개 등)가 있다.

개발자의 수 : 큰 프로젝트와 같이 개발자가 많으면 많을수록 코드를 깔끔하게 적용하거나 Merge 하기 어려워진다. 어떤 커밋은 다른 개발자가 이미 기여한 것으로 불필요해지기도 하고 때론 서로 충돌이 일어날 것이다. 어떻게 해야 코드를 최신으로 유지하면서 원하는 대로 수정할 수 있을지 고민해 보아야 한다.

프로젝트에서 선택한 워크플로 : 개발자 모두가 메인 저장소에 쓰기 권한을 갖는 중앙집중형 방식인가? 프로젝트에 모든 Patch를 검사하고 통합하는 관리자가 따로 있는가? 모든 수정사항을 개발자끼리 검토하고 승인하는가? 자신이 그저 돕는게 아니라 어떤 책임을 맡고 있는지? 중간 관리자가 있어서 그들에게 먼저 알려야 하는가? 여러 방법을 고민해야 한다.

접근 권한 : '프로젝트에 쓰기 권한이 있어서 직접 쓸 수 있는가? 아니면 읽기만 가능한가?'에 따라서 프로젝트에 기여하는 방식이 매우 달라진다. 쓰기 권한이 없다면 어떻게 수정 사항을 프로젝트에 반영할 수 있을지, 수정사항을 적용하는 정책이 프로젝트에 있는지, 얼마나 많은 시간을 프로젝트에 할애 및 기여하는지 등 여러 상황을 고민하고 적용해야 한다.

이런 변수에 따라 프로젝트에 기여하는 방법과 워크플로가 달라진다.

## 프로젝트 관리하기

프로젝트의 완성도를 높이려면 효율적으로 기여하는 방법뿐만 아니라 운영하는 방법도 알아야 한다. 프로젝트를 운영하는 것은 크게 두 가지로 이루어진다.

format-patch 명령으로 생성한 Patch를 이메일로 받아서 프로젝트에 Patch를 적용하는 것

프로젝트의 다른 리모트 저장소로부터 변경 내용을 Merge 하는 것

저장소를 아주 깔끔하고 정돈된 상태로 운영하고 Patch를 적용하거나 수정사항을 확인하기 쉬운 상태로 유지하려면 좋은 운영 방식을 터득해야 한다. 좋은 운영 방식은 다른 사람들이 이해하기 쉽고 프로젝트가 오랫동안 운영돼도 흐트러짐이 없는 것이다..

# Git 도구

## 리비전 조회하기

리비전은 하나 또는 범위를 주어 여러 개를 조회 할 수 있다. 아래에서 조회 방법에 대해 알아보자.



### SHA-1 줄여 쓰기

Git에서 커밋시 생성된 버전에는 고유한 SHA-1 해시값이 부여된다. 즉, Git은 해시값으로 어떤 커밋인지 식별할 수 있다. 이러한 해시 값은 중복되지 않으면 앞의 몇(4) 글자만으로도 식별이 가능하다. 아래의 예를 통해 알아보자.

‘git log’ 명령을 통해 어떤 커밋이 있는지 조회해보자.

|  |
| --- |
| $git log  commit 1334cc23c1780cf0e88d0ddbfca3a1159107ea9b  Author: hhlee <hhlee@linkto.co.kr>  Date: Tue May 23 11:36:17 2017 +0900  ver9  i Please enter the commit message for your changes. Lines starting  commit 38e5ca7b621d72997073891fa378c10b40d0fabc  Merge: ed06861 81ad5e9  Author: hhlee <hhlee@linkto.co.kr>  Date: Tue May 23 10:59:20 2017 +0900  Merge branch 'exp'  commit ed068615a1e36cf25f40b060bb68f473b7be7a6f  Author: hhlee <hhlee@linkto.co.kr>  Date: Tue May 23 10:38:44 2017 +0900  ver7 |

위처럼 커밋이 있다면, ‘git show’ 명령으로 특정한 커밋을 조회 할 수 있다. 다음에서는 ‘ed068615a1e36cf25f40b060bb68f473b7be7a6f’의 해쉬를 통해 조회를 해볼것이다.

|  |  |
| --- | --- |
| SHA-1 해쉬값 전체를 검색어로 사용한 방법 | USER@LG-15Z950-GT30K MINGW64 ~/git\_tests/gitfth (exp)  $ git show ed068615a1e36cf25f40b060bb68f473b7be7a6f  commit ed068615a1e36cf25f40b060bb68f473b7be7a6f  Author: hhlee <hhlee@linkto.co.kr>  Date: Tue May 23 10:38:44 2017 +0900  ver7  diff --git a/f3.txt b/f3.txt  new file mode 100644  index 0000000..e8d09e2  --- /dev/null  +++ b/f3.txt  @@ -0,0 +1 @@  +git branch test 3 |
| SHA-1 해쉬값의 일부를 검색어로 사용한 방법 | USER@LG-15Z950-GT30K MINGW64 ~/git\_tests/gitfth (exp)  $ git show ed06  commit ed068615a1e36cf25f40b060bb68f473b7be7a6f  Author: hhlee <hhlee@linkto.co.kr>  Date: Tue May 23 10:38:44 2017 +0900  ver7  diff --git a/f3.txt b/f3.txt  new file mode 100644  index 0000000..e8d09e2  --- /dev/null  +++ b/f3.txt  @@ -0,0 +1 @@  +git branch test 3 |

이와 같이 중복되지 않은 해쉬값을 통하여 서로 같은 커밋을 조회 할 수 있는 것을 볼 수 있다.

하지만 ‘git log’명령을 통해 사용자가 일일이 보고 해쉬값의 중복을 확인하기에는 어려운 점이 있다. 그럴때 ‘—abbrev-commit –pretty=online’이라는 옵션을 추가하면 중복되지 않는 짧은 해시값을 보여준다.

|  |
| --- |
| USER@LG-15Z950-GT30K MINGW64 ~/git\_tests/gitfth (exp)  $ git log --abbrev-commit --pretty=oneline  cc5c8f8 (HEAD -> exp, master) ver13  9000411 ver12  ae6ed5f ver10  1334cc2 ver9  38e5ca7 Merge branch 'exp'  ed06861 ver7  81ad5e9 ver8  313ee85 ver6  e5136d5 ver5 i Please enter the commit message for your changes. Lines starting  63f0c73 ver 4  bf95ce0 ver3 3 Please enter the commit message for your changes. Lines starting  fe00d1c ver2  5a5e42e ver1 i Please enter the commit message for your changes. Lines starting |

### 브랜치로 가리키기

‘git show <해쉬값>’을 통해 해당 커밋을 확인할 수 있었다면, 브랜치를 통해서도 커밋을 확인할 수 있다. 아래와 같이 ‘git show <브랜치>’ 명령을 사용하면 해당 브랜치의 최근 커밋을 확인할 수 있다.

|  |
| --- |
| USER@LG-15Z950-GT30K MINGW64 ~/git\_tests/gitfth (exp)  $ git show exp  commit cc5c8f8cd0889af106e54046221c33831b4678ce (HEAD -> exp, master)  Merge: 1334cc2 9000411  Author: hhlee <hhlee@linkto.co.kr>  Date: Tue May 23 16:17:42 2017 +0900  ver13  diff --cc master.txt  index 442406a,86c1454..8b13789  --- a/master.txt  +++ b/master.txt  @@@ -1,2 -1,1 +1,1 @@@  - a  -exp master  + |

이렇게 브랜치 명은 최근 SHA-1 해쉬값을 가리키고 있는 것을 확인할 수 있다. 이와 같이 브랜치로 해쉬값을 알아보려면 ‘git rev-parse <브랜치 명>’ 명령을 통해 알 수 있다.

|  |
| --- |
| USER@LG-15Z950-GT30K MINGW64 ~/git\_tests/gitfth (exp)  $ git rev-parse exp  cc5c8f8cd0889af106e54046221c33831b4678ce |

### Reflog로 가리키기

Reflog란, Git이 자동으로 브랜치와 HEAD가 지난일정기간동안 가리켰었던 기록을 의미한다.

‘git reflog’ 명령을 실행하면 아래와 같이 Reflog를 볼 수 있다.

|  |
| --- |
| USER@LG-15Z950-GT30K MINGW64 ~/git\_tests/gitfth (exp)  $ git reflog  cc5c8f8 (HEAD -> exp, master) HEAD@{0}: checkout: moving from master to exp  cc5c8f8 (HEAD -> exp, master) HEAD@{1}: checkout: moving from exp to master  cc5c8f8 (HEAD -> exp, master) HEAD@{2}: checkout: moving from master to exp  cc5c8f8 (HEAD -> exp, master) HEAD@{3}: checkout: moving from exp to master  cc5c8f8 (HEAD -> exp, master) HEAD@{4}: reset: moving to HEAD  cc5c8f8 (HEAD -> exp, master) HEAD@{5}: checkout: moving from master to exp  cc5c8f8 (HEAD -> exp, master) HEAD@{6}: commit (merge): ver13  1334cc2 HEAD@{7}: checkout: moving from exp to master  9000411 HEAD@{8}: commit: ver12  ae6ed5f HEAD@{9}: commit: ver10  38e5ca7 HEAD@{10}: checkout: moving from master to exp  1334cc2 HEAD@{11}: commit: ver9  38e5ca7 HEAD@{12}: merge exp: Merge made by the 'recursive' strategy.  ed06861 HEAD@{13}: commit: ver7  313ee85 HEAD@{14}: checkout: moving from exp to master  81ad5e9 HEAD@{15}: checkout: moving from master to exp  313ee85 HEAD@{16}: checkout: moving from exp to master  81ad5e9 HEAD@{17}: commit: ver8  313ee85 HEAD@{18}: checkout: moving from master to exp  313ee85 HEAD@{19}: commit: ver6  e5136d5 HEAD@{20}: commit: ver5  63f0c73 HEAD@{21}: commit: ver 4  bf95ce0 HEAD@{22}: commit: ver3  fe00d1c HEAD@{23}: commit: ver2  5a5e42e HEAD@{24}: commit (initial): ver1 |

위와 같이 Git은 브랜치가 가리키는 것이 달라질 때마다 그 정보를 임시 영역에 저장한다. 그래서 예전에 가리키던 것을 확인해 볼 때는 @{n} 규식을 사용하면 된다. 아래에서 ver7인 13번째 기록을 확인해 보자.

|  |
| --- |
| USER@LG-15Z950-GT30K MINGW64 ~/git\_tests/gitfth (exp)  $ git show head@{13}  commit ed068615a1e36cf25f40b060bb68f473b7be7a6f  Author: hhlee <hhlee@linkto.co.kr>  Date: Tue May 23 10:38:44 2017 +0900  ver7  diff --git a/f3.txt b/f3.txt  new file mode 100644  index 0000000..e8d09e2  --- /dev/null  +++ b/f3.txt  @@ -0,0 +1 @@  +git branch test 3 |

또한 git reflog 결과를 git log 명령과 같은 형태로 볼 수 있는데, ‘git log –g’ 옵션을 사용하면 된다.

|  |
| --- |
| USER@LG-15Z950-GT30K MINGW64 ~/git\_tests/gitfth (exp)  $ git log -g  commit cc5c8f8cd0889af106e54046221c33831b4678ce (HEAD -> exp, master)  Reflog: HEAD@{0} (hhlee <hhlee@linkto.co.kr>)  Reflog message: checkout: moving from master to exp  Author: hhlee <hhlee@linkto.co.kr>  Date: Tue May 23 16:17:42 2017 +0900  ver13  commit cc5c8f8cd0889af106e54046221c33831b4678ce (HEAD -> exp, master)  Reflog: HEAD@{1} (hhlee <hhlee@linkto.co.kr>)  Reflog message: checkout: moving from exp to master  Author: hhlee <hhlee@linkto.co.kr>  Date: Tue May 23 16:17:42 2017 +0900  ver13  commit cc5c8f8cd0889af106e54046221c33831b4678ce (HEAD -> exp, master)  Reflog: HEAD@{2} (hhlee <hhlee@linkto.co.kr>)  Reflog message: checkout: moving from master to exp  Author: hhlee <hhlee@linkto.co.kr>  Date: Tue May 23 16:17:42 2017 +0900  ver13 |

## 대화형 명령

Git에서는 대화형 스크립트를 제공해 주는데, 이것을 통해서 전문가처럼 능숙하게 커밋을 할 수 있다. 이 스크립트를 통해 커밋할 파일을 고르고 수정된 파일의 일부를 커밋할 수도 있다.

‘git add’명령에 ‘–I’ 나 ‘—interactive’ 옵션을 주고 실행하면 아래와 같이 대화형 모드가 실행된다.

|  |
| --- |
| $ git add -i  staged unstaged path  1: unchanged +0/-1 TEST  2: unchanged +1/-1 index.html  3: unchanged +5/-1 lib/simplegit.rb  \*\*\* Commands \*\*\*  1: status 2: update 3: revert 4: add untracked  5: patch 6: diff 7: quit 8: help  What now> |

이 명령은 Staging Area의 현재 상태와 할 수 있는 것들을 보여준다. 왼쪽에는 Staged 상태인 파일들이 오른쪽에는 Unstaged상태인 파일들을 보여준다. 그리고 commands 부분에서 할 수 있는 것들을 보여주는데 아래에서 자세히 알아보자.

|  |  |  |
| --- | --- | --- |
| Commands | 기능 | 예제 |
| 1 or s | Staging Area의 현재 상태를 보여줌 | \*\*\* Commands \*\*\*  1: status 2: update 3: revert 4: add untracked  5: patch 6: diff 7: quit 8: help  What now> 1  staged unstaged path  1: unchanged +0/-1 TEST  2: unchanged +1/-1 index.html  3: unchanged +5/-1 lib/simplegit.rb |
| 2 or u | Staging Area에 파일을 추가 할수 있다.  \* 표시가 붙은 파일은 Stage 하도록 선택한 것이다. 선택하고 Update>> 프롬프트에 아무것도 입력하지 않고 엔터를 치면 Git은 선택한 파일을 Staging Area로 추가한다. | What now> 2  staged unstaged path  1: unchanged +0/-1 TEST  2: unchanged +1/-1 index.html  3: unchanged +5/-1 lib/simplegit.rb  Update>>1,2  staged unstaged path  \* 1: unchanged +0/-1 TEST  \* 2: unchanged +1/-1 index.html  3: unchanged +5/-1 lib/simplegit.rb  Update>>  updated 2 paths  \*\*\* Commands \*\*\*  1: status 2: update 3: revert 4: add untracked  5: patch 6: diff 7: quit 8: help |
| 3 or r | Staging Area에 추가된 파일 취소할 수 있다. | What now> 3  staged unstaged path  1: +0/-1 nothing TEST  2: +1/-1 nothing index.html  3: unchanged +5/-1 lib/simplegit.rb  Revert>> 1  staged unstaged path  \* 1: +0/-1 nothing TEST  2: +1/-1 nothing index.html  3: unchanged +5/-1 lib/simplegit.rb  Revert>> [enter]  reverted one path |

## Stashing과 Cleaning

어떠한 프로젝트를 진행할 때, 다른요청으로 인해 작업중이던 브랜치를 변경해야 할 일이 생겼을 경우 완료되지 않은 작업을 커밋하기에는 꺼림직하다. 커밋을 하지 않고 나중에 다시 작업을 하고 싶을것이다. 이러한 문제는 ‘git stash’ 명령을 통해 해결할 수 있다.

Stash 명령은 워킹디렉토리에서 수정한 파일들만을 저장한다. 즉, Stash는 Modified이면서 Tracked 상태인 파일과 Staging Area에 있는 파일들을 보관해두는 장소다. 아직 끝내지 않은 수정사항을 스택에 잠시 저장했다가 나중에 다시 적용할 수 있다.



### 하던일 Stash하기

`git stash`나 `git stash save`를 실행하면 스택에 새로운 Stash가 만들어진다. 대신 워킹 디렉토리는 Clean된다. 이렇게 만들어진 Stash는 스택(Stack)에 저장되고 이것을 확인하려면 ‘git stash list’명령을 사용하면 된다. Stash를 재적용 하는 방법은 ‘git stash apply’ 명령을 사용한다. 하지만 이때는 staged 상태였던 파일을 자동으로 다시 Staged 상태로 만들어 주지 않는다. 그렇기 때문에 ‘—index’ 옵션을 주어 Staged상태까지 적용하여야 한다.

### Stash를 새롭게 만들기

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

많은 파일을 변경했지만 몇몇 파일만 커밋하고 나머지 파일은 나중에 처리하고 싶을 때 유용하다.

### Stash 적용한 브랜치 만들기

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

### 워킹디렉토리 청소하기

작업하고 있던 파일을 Stash 하지 않고 단순히 그 파일들을 치워버리고 싶을 때가 있다. git clean 명령이 그 일을 한다. 보통은 Merge나 외부 도구가 만들어낸 파일을 지우거나 이전 빌드 작업으로 생성된 각종 파일을 지우는 데 필요하다. 이 명령을 사용할 때는 신중해야 한다. 이 명령을 사용하면 워킹 디렉토리 안의 추적하고 있지 않은 모든 파일이 지워지기 때문이다. 명령을 실행하고 나서 후회해도 소용없다. 지워진 파일은 돌아오지 않는다. ‘git stash –all’ 명령을 이용하면 지우는 건 똑같지만, 먼저 모든 파일을 Stash 하므로 좀 더 안전하다.

워킹 디렉토리의 불필요한 파일들을 전부 지우려면 ‘git clean`을 사용한다. 추적 중이지 않은 모든 정보를 워킹 디렉토리에서 지우고 싶다면 ‘git clean -f –d’ 명령을 사용하자. 이 명령은 하위 디렉토리까지 모두 지워버린다. ‘-f’ 옵션은 강제(force)의 의미이며 "진짜로 그냥 해라"라는 뜻이다. 이 명령을 실행했을 때 어떤 일이 일어날지 미리 보고 싶다면 ‘-n’ 옵션을 사용한다. ‘–n’ 옵션은 ”가상으로 실행해보고 어떤 파일들이 지워질지 알려달라’'라는 뜻이다.

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

## 검색

프로젝트 진행시 함수의 정의나 함수가 호출되는 곳을 검색해야 하는 경우가 많다. 그리고 함수으 히스토리를 찾아보기도 한다. 이럴때 Git은 DB에 저장된 코드나 커밋에서 원하는 바를 검색하는 여러 도구들을 지원해준다. 아래에서 함께 살펴보도록 하자.



### Git Grep

Git의 grep 명령을 이용하면 커밋 트리의 내용이나 워킹 디렉토리의 내용을 문자열, 정규표현식을 통해 쉽게 검색 할 수 있다. Grep은 기본적으로 대상을 지정하지 않으면 워킹 디렉토리의 파일에서 찾는다. 그리고 ‘-n’ 옵션을 추가하면 찾을 문자열의 라인 번호도 같이 출력한다.

|  |
| --- |
| $ git grep -n gmtime\_r  compat/gmtime.c:3:#undef gmtime\_r  compat/gmtime.c:8: return git\_gmtime\_r(timep, &result);  compat/gmtime.c:11:struct tm \*git\_gmtime\_r(const time\_t \*timep, struct tm \*result)  compat/gmtime.c:16: ret = gmtime\_r(timep, result);  compat/mingw.c:606:struct tm \*gmtime\_r(const time\_t \*timep, struct tm \*result)  compat/mingw.h:162:struct tm \*gmtime\_r(const time\_t \*timep, struct tm \*result);  date.c:429: if (gmtime\_r(&now, &now\_tm))  date.c:492: if (gmtime\_r(&time, tm)) {  git-compat-util.h:721:struct tm \*git\_gmtime\_r(const time\_t \*, struct tm \*);  git-compat-util.h:723:#define gmtime\_r git\_gmtime\_r |

반면 위의 결과와 달리 어떤 파일에서 몇 번 쓰였는지만 나타내고 싶다면 ‘—count’ 옵션을 이용하면 된다.

|  |
| --- |
| $ git grep --count gmtime\_r  compat/gmtime.c:4  compat/mingw.c:1  compat/mingw.h:1  date.c:2  git-compat-util.h:2 |

그리고 단순 문자열이 아닌 매칭되는 라인이 있는 함수나 메서드를 찾고 싶다면 ‘-p’옵션을 주도록 하자.

|  |
| --- |
| $ git grep -p gmtime\_r \*.c  date.c=static int match\_multi\_number(unsigned long num, char c, const char \*date, char \*end, struct tm \*tm)  date.c: if (gmtime\_r(&now, &now\_tm))  date.c=static int match\_digit(const char \*date, struct tm \*tm, int \*offset, int \*tm\_gmt)  date.c: if (gmtime\_r(&time, tm)) { |

### Git Log 검색

Git에서는 히스토리에 언제 추가,변경 되었는지 찾아 볼수 있다. 예를 들어 ZLIB\_BUF\_MAX 라는 상수가 가장 처음 나타난 때를 찾는 문제라면 ‘-S’ 옵션을 이용해 해당 문자열이 추가된 커밋과 없어진 커밋만 검색할 수 있다.

|  |
| --- |
| $ git log -SZLIB\_BUF\_MAX --oneline  e01503b zlib: allow feeding more than 4GB in one go  ef49a7a zlib: zlib can only process 4GB at a time |

위 두 커밋의 변경 사항을 살펴보면 ef49a7a에서 `ZLIB\_BUF\_MAX 상수가 처음 나오고 e01503b 에서는 변경된 것을 알 수 있다.

더 세세한 조건을 걸어 찾고 싶다면 로그를 검색할 때 -G 옵션으로 정규표현식을 써서 검색하면 된다.

### 라인 히스토리 검색

git log`를 쓸 때 `-L 옵션을 붙이면 어떤 함수나 한 라인의 히스토리를 볼 수 있다.

예를 들어, zlib.c 파일에 있는 git\_deflate\_bound 함수의 모든 변경 사항들을 보길 원한다고 생각해보자. git log -L :git\_deflate\_bound:zlib.c 라고 명령 실행하면 된다. 이 명령을 실행하면 함수의 시작과 끝을 인식해서 함수에서 일어난 모든 히스토리를 함수가 처음 만들어진 때부터 Patch를 나열하여 보여준다.

|  |
| --- |
| $ git log -L :git\_deflate\_bound:zlib.c  commit ef49a7a0126d64359c974b4b3b71d7ad42ee3bca  Author: Junio C Hamano <gitster@pobox.com>  Date: Fri Jun 10 11:52:15 2011 -0700  zlib: zlib can only process 4GB at a time  diff --git a/zlib.c b/zlib.c  --- a/zlib.c  +++ b/zlib.c  @@ -85,5 +130,5 @@  -unsigned long git\_deflate\_bound(z\_streamp strm, unsigned long size)  +unsigned long git\_deflate\_bound(git\_zstream \*strm, unsigned long size)  {  - return deflateBound(strm, size);  + return deflateBound(&strm->z, size);  }  commit 225a6f1068f71723a910e8565db4e252b3ca21fa  Author: Junio C Hamano <gitster@pobox.com>  Date: Fri Jun 10 11:18:17 2011 -0700  zlib: wrap deflateBound() too  diff --git a/zlib.c b/zlib.c  --- a/zlib.c  +++ b/zlib.c  @@ -81,0 +85,5 @@  +unsigned long git\_deflate\_bound(z\_streamp strm, unsigned long size)  +{  + return deflateBound(strm, size);  +}  + |

Git이 함수의 처음과 끝을 인식하지 못할 때는 정규표현식으로 인식하게 할 수도 있다. ‘git log -L '/unsigned long git\_deflate\_bound/',/^}/:zlib.c’ 명령으로 위와 같은 결과를 볼 수 있다. 한 라인의 히스토리만 검색할 수도 있고 여러 라인에 걸친 히스토리를 검색할 수도 있다.

## 히스토리 단장하기

Git을 활용하면 커밋 히스토리를 수정해야 할 때가 있다. 커밋 히스토리로는 커밋의 순서, 메시지, 파일 등도 있고, 혹인 여러 개의 커밋을 하나도 합치거나 분리, 삭제 할 수도 있다. 이러한 결정들을 나중으로 미룰 수 있는 것이 장점이지만, 이것들 또한 다른 사람들과 공유하기 전까지는 해야 한다.



### 마지막커밋 수정하기

마지막 커밋을 수정하는 것은 자주 발생한다. 이것은 기본적으로 커밋 메시지 수정과 파일 목록 수정하는 것이 있다. 커밋 메시지를 수정하는 방법은 ‘git commit –amend’ 명령을 사용하면 된다. 이 명령은 자동으로 텍스트편집기를 실행시켜 마지막 커밋 메시지를 열어준다. 여기서 편집하고 저장하면 커밋 메시지가 수정되는 것이다.

그리고 커밋 파일 수정은 ‘git add’명령과 ‘git rm’명령을 통하여 추가/삭제를 한 후 ‘git commit --amend’ 명령으로 커밋하면 된다. 이때 SHA-1 해쉬값이 변경되기 때문에 과거의 커밋을 변경할때는 주의가 필요하다.

### 커밋 메시지 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. log 명령을 실행하면 아래와 같은 결과를 볼 수 있다.

|  |
| --- |
| $ git log --pretty=format:"%h %s" HEAD~3..HEAD  a5f4a0d added cat-file  310154e updated README formatting and added blame  f7f3f6d changed my name a bit |

위 결과의 역순임을 기억하자. 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.

### 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.

### 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 합쳐진다.

### 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.

## Reset 명확히 알고 가기

Git은 일반적으로 아래 세가지 트리를 관리하는 시스템이다.

|  |  |
| --- | --- |
| 트리 | 역할 |
| HEAD | * 마지막 커밋 스냅샷, 다름 커밋의 부모 커밋 * 현재 브랜치를 가리키는 포인터이며, 브랜치는 가장 마지막 커밋을 가리킨다. |
| Index | * 다음에 커밋할 스냅샷 * 사용자가 git commit 명령을 실행했을 때 Git이 처리할 것들이 있는 곳(Staging Area) |
| Working Directory | * 샌드박스 * 실제 파일이 존재하는곳으로 편집하기 수월하며, 이 파일들은 커밋 전에 Index에 올려놓고 얼마든지 변경가능하다. |

아래와 같이 Git의 동작과 트리 동작을 살펴 볼 수 있다.

|  |  |
| --- | --- |
| 동작 | 트리 동작 구현 이미지 |
| File 생성 |  |
| Git add |  |
| Git commit |  |
| 파일 수정 |  |
| Git add |  |
| Git commit |  |



### Reset의 역할

Reset의 동작은 3가지(HEAD이동, Index 업데이트, 워킹디렉토리 업데이트)의 단계로 이루어 진다.

Reset 명령은 HEAD 브랜치를 이동을 시키는데, checkout 명령처럼 HEAD가 가리키는 브랜치를 바꾸지는 않는다. Reset은 옵션에 따라서 지정한 곳까지 동작하고 멈춘다. ‘--soft’ 옵션은 HEAD는 현재 브랜치를 가리키고, 그 브랜치가 가리키는 커밋을 바꾸는 것, ‘—mixed’ 옵션은 Index를 현재 HEAD가 가리키는 스냅샷으로 업데이트까지, ‘--hard’ 옵션은 워킹 디렉토리의 내용까지 동작된다.

|  |  |  |  |
| --- | --- | --- | --- |
|  | HEAD 이동  (--soft) | Index 업데이트  (--mixed) | 워킹디렉토리 업데이트  (--hard) |
| 구현 동작 |  |  |  |

## Git으로 버그 찾기



### 파일 어노테이션

파일 어노테이션은 버그를 찾을 때 먼저 그 코드가 왜,언제 추가 되었는지 알고 싶을 때 사용한다. ‘git blame’ 명령으로 어떤 메소드에 버그가 있으면 그 메소드의 각 라인을 누가 언제 마지막으로 고쳤는지 알 수 있다. 아래에서 test.rb 파일을 통해 자세히 알아보자.

|  |
| --- |
| $ git blame -L 12,22 test.rb  ^4832fe2 (Scott Chacon 2008-03-15 10:31:28 -0700 12) def show(tree = 'master')  ^4832fe2 (Scott Chacon 2008-03-15 10:31:28 -0700 13) command("git show #{tree}")  ^4832fe2 (Scott Chacon 2008-03-15 10:31:28 -0700 14) end  ^4832fe2 (Scott Chacon 2008-03-15 10:31:28 -0700 15)  9f6560e4 (Scott Chacon 2008-03-17 21:52:20 -0700 16) def log(tree = 'master')  79eaf55d (Scott Chacon 2008-04-06 10:15:08 -0700 17) command("git log #{tree}")  9f6560e4 (Scott Chacon 2008-03-17 21:52:20 -0700 18) end  9f6560e4 (Scott Chacon 2008-03-17 21:52:20 -0700 19)  42cf2861 (Magnus Chacon 2008-04-13 10:45:01 -0700 20) def blame(path)  42cf2861 (Magnus Chacon 2008-04-13 10:45:01 -0700 21) command("git blame #{path}")  42cf2861 (Magnus Chacon 2008-04-13 10:45:01 -0700 22) end |

위처럼 첫 항목은 SHA-1 해쉬 값을 그 다음 항목은 누가, 언제 그 라인을 커밋했는지를 그 다음에는 라인 번호와 내용을 보여준다. 이와 같이 누가, 언제 커밋했는지 쉽게 찾을 수 있다. 거기에 ^4832fe2처럼 해당 라임이 처음 커밋되었다는 것도 알 수 있게 해준다.

### 이진 탐색

파일 어노테이션은 특정 이슈와 관련된 커밋을 찾는데 좋다. 하지만 문제가 발생했을 때 의심스러운 커밋이 수십, 수백 개에 이르는 경우 도대체 어디서부터 시작할지 모를 때 ‘git bisect’ 명령을 사용하면 유용하다. 이 명령은 커밋 히스토리를 이진 탐색 방법으로 좁혀 주기 때문에 이슈와 관련된 커밋을 최대한 빠르게 찾아낼 수 있도록 도와준다.

예를 들어 코드를 운용 환경에 배포하고 난 후에 개발할 때 발견하지 못한 버그가 있다고 보고받았다. 그런데 왜 그런 현상이 발생하는지 아직 이해하지 못하는 상황이라 해보자. 해당 이슈를 다시 만들고 작업하기 시작했는데 뭐가 잘못됐는지 알아낼 수 없다. 이때 bisect 명령을 사용하여 코드를 뒤져 보는 게 좋다. 먼저 git bisect start 명령으로 이진 탐색을 시작하고 git bisect bad`를 실행하여 현재 커밋에 문제가 있다고 표시를 남기고 나서 문제가 없는 마지막 커밋을 `git bisect good [good\_commit] 명령으로 표시한다.

|  |
| --- |
| $ git bisect start  $ git bisect bad  $ git bisect good v1.0  Bisecting: 6 revisions left to test after this  [ecb6e1bc347ccecc5f9350d878ce677feb13d3b2] error handling on repo |

이 예제에서 마지막으로 괜찮았던 커밋(v1.0)과 현재 문제가 있는 커밋 사이에 있는 커밋은 전부 12개이고 Git은 그 중간에 있는 커밋을 Checkout 해준다. 여기에서 해당 이슈가 구현됐는지 테스트해보고 만약 이슈가 있으면 그 중간 커밋 이전으로 범위를 좁히고 이슈가 없으면 그 중간 커밋 이후로 범위를 좁힌다. 이슈를 발견하지 못하면 `git bisect good`으로 이슈가 아직 없음을 알리고 계속 진행한다.

|  |
| --- |
| $ git bisect good  Bisecting: 3 revisions left to test after this  [b047b02ea83310a70fd603dc8cd7a6cd13d15c04] secure this thing |

현재 문제가 있는 커밋과 지금 테스트한 커밋 사이에서 중간에 있는 커밋이 Checkout 됐다. 다시 테스트해보고 이슈가 있으면 `git bisect bad`로 이슈가 있다고 알린다.

|  |
| --- |
| 현재 문제가 있는 커밋과 지금 테스트한 커밋 사이에서 중간에 있는 커밋이 Checkout 됐다. 다시 테스트해보고 이슈가 있으면 `git bisect bad`로 이슈가 있다고 알린다. |

이제 이슈를 처음 구현한 커밋을 찾았다. 이 SHA-1 값을 포함한 이 커밋의 정보를 확인하고 수정된 파일이 무엇인지 확인할 수 있다. 이 문제가 발생한 시점에 도대체 무슨 일이 있었는지 아래와 같이 살펴본다.

|  |
| --- |
| $ git bisect good  b047b02ea83310a70fd603dc8cd7a6cd13d15c04 is first bad commit  commit b047b02ea83310a70fd603dc8cd7a6cd13d15c04  Author: PJ Hyett <pjhyett@example.com>  Date: Tue Jan 27 14:48:32 2009 -0800  secure this thing  :040000 040000 40ee3e7821b895e52c1695092db9bdc4c61d1730  f24d3c6ebcfc639b1a3814550e62d60b8e68a8e4 M config |

이제 찾았으니까 git bisect reset 명령을 실행시켜서 이진 탐색을 시작하기 전으로 HEAD를 돌려놓는다.

|  |
| --- |
| $ git bisect reset |

수백 개의 커밋들 중에서 버그가 만들어진 커밋을 찾는 데 몇 분밖에 걸리지 않는다. 프로젝트가 정상적으로 수행되면 0을 반환하고 문제가 있으면 1을 반환하는 스크립트를 만든다면, 이 git bisect 과정을 완전히 자동으로 수행할 수 있다. 먼저 bisect start 명령으로 이진 탐색에 사용할 범위를 알려준다. 위에서 한 것처럼 문제가 있다고 아는 커밋과 문제가 없다고 아는 커밋을 넘기면 된다.

|  |
| --- |
| $ git bisect start HEAD v1.0  $ git bisect run test-error.sh |

문제가 생긴 첫 커밋을 찾을 때까지 Checkout 할 때마다 `test-error.sh`를 실행한다. `make`가 됐든지 `make tests`가 됐든지 어쨌든 이슈를 찾는 테스트를 실행하여 찾는다.

## Bundle

Git에는 Bundle이란 데이터를 한 파일에 몰아넣는 것이다. 이 방법은 다양한 경우 유용하게 사용할 수 있다. 예를 들어 네트워크가 불통인데 변경사항을 동료에게 보낼 때, 출장을 나갔는데 보안상의 이유로 로컬 네트워크에 접속하지 못할 때, 통신 인터페이스 장비가 고장났을 때, 갑자기 공용 서버에 접근하지 못할 때, 누군가에게 수정사항을 이메일로 보내야 하는데 40개 씩이나 되는 커밋을 `format-patch`로 보내고 싶지 않을 때를 예로 들 수 있다.

바로 이럴 때 ‘git bundle’이 유용하다. ‘git bundle’ 명령은 보통 `git push`명령으로 올려 보낼 모든 것을 감싸서 한 바이너리 파일로 만든다. 이 파일을 이메일로 보내거나 USB로 다른 사람에게 보내서 다른 저장소에 풀어서(Unbundle) 사용하면 된다.

## Replace

Git의 replace 명령은 “어떤 개체를 읽을 때 항상 다른 개체로 보이게 한다.” 이다. 즉, 히스토리에서 어떤 커밋이 다른 커밋처럼 보이도록 할 때 사용된다. Git의 개체는 기본적으로 히스토리(혹은 데이터 베이스)에 저장하면 변경 할 수 없기에 이 기능을 사용한다.

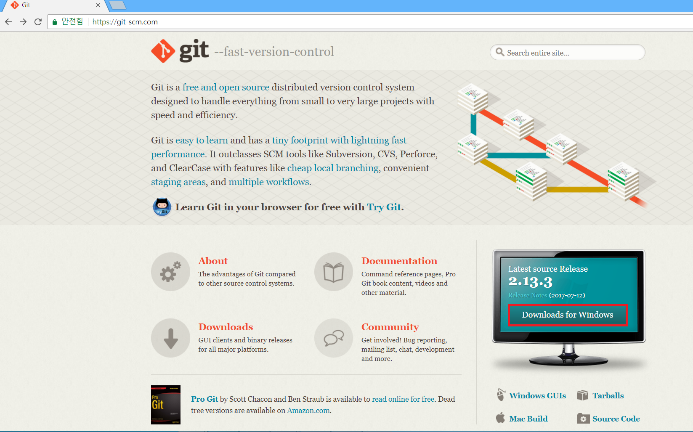
예를 들어 현 프로젝트의 히스토리가 매우 방대할 때, 새로 시작하는 개발자와 프로젝트 히스토리를 분석자가 있다고 하자. 이럴때 개발자에게는 분석자에게 처럼 복잡한 전체 히스토리를 제공하는 것보다는 아주 간단한 몇 개의 커밋으로 만들어 히스토리를 제공해 주는 것이 좋을것이다. 이렇게 간단히 만든 히스토리를 전체 히스토리의 마지막 부분에 연결할 때 replace 명령이 사용된다.

|  |  |
| --- | --- |
| $ git replace 81a708d c6e1e95 | |
|  |  |

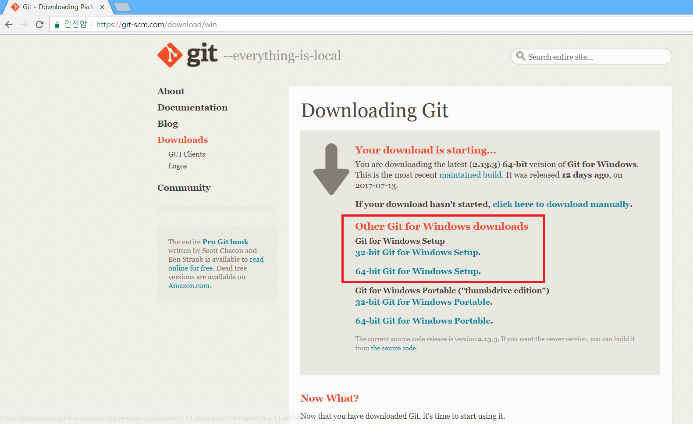
# Git 따라하기

## Git 설치하기

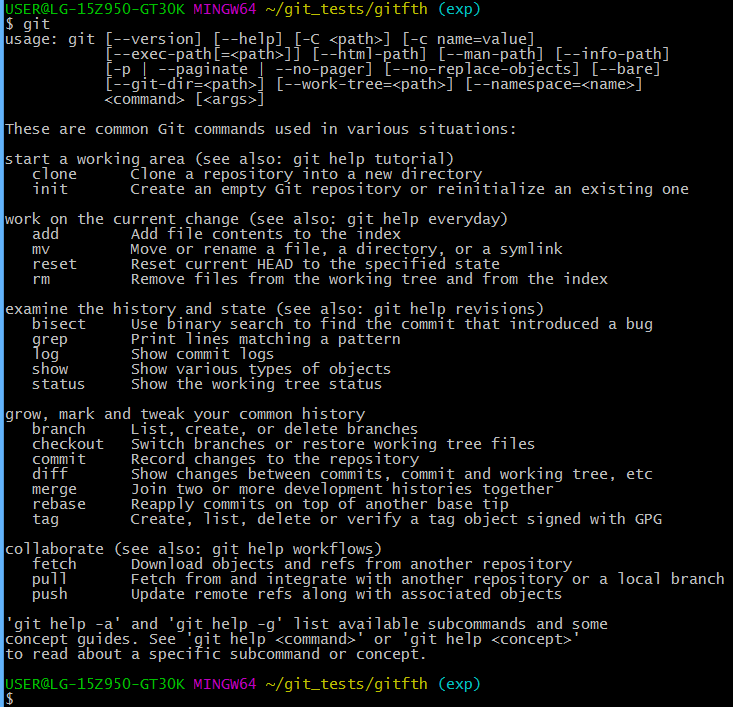
* <https://git-scm.com/> 으로 접속한다.
* 아래 그림과 같이 Downloads for Windows를 선택한다.



* 자동으로 설치파일이 다운 받아지지 않으면 해당 파일을 선택하여 다운 받는다.



* 다운 받은 파일을 실행하여 설치하면 된다. (특별한 설정이 없으므로 Next하면 된다.)
* 설치후 git bash를 실행 시켜 git이라 명령어를 치고 엔터를 쳐서 아래와 같이 나오면 정상적으로 설치된 것이다.



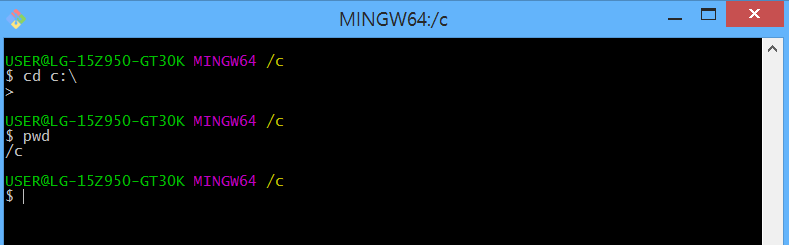
## 저장소 만들기(init)

사용할 프로젝트 파일들을 보관할 폴더를 만들어 보자. Window 탐색기를 통해서 만들어 접속해도 되고, 아님 Git bash 터미널을 통해 명령어로 만들어도 된다. 폴더 진입은 ‘cd <경로>’ 명령이고 생성은 ‘mkdir <폴더 명>’이다.

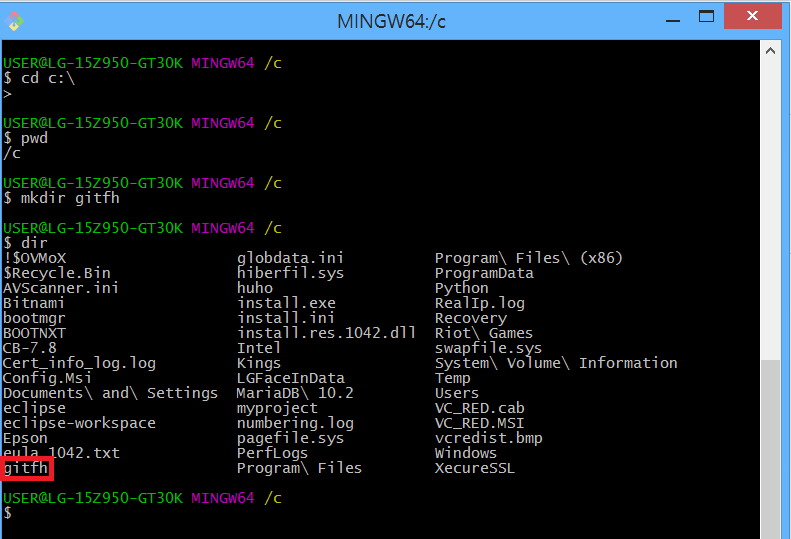
만든 폴더에 진입하였으면 ‘git init’ 명령을 통해 현재 폴더를 저장소로 등록(버전관리)하자.

아래의 예처럼 하면 된다.

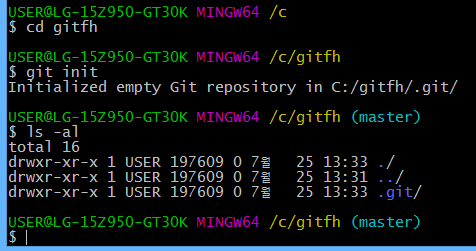
c 드라이브 경로로 이동한다.



gitfh라는 버전관리할 폴더를 생성한다.



gitfh 폴더로 이동하여 버전관리 할 폴더로 등록한다.



위처럼 .git이라는 폴더가 생성되면 제대로 등록 된것이다.

## Git이 관리할 대상으로 파일 등록하기(add & status)

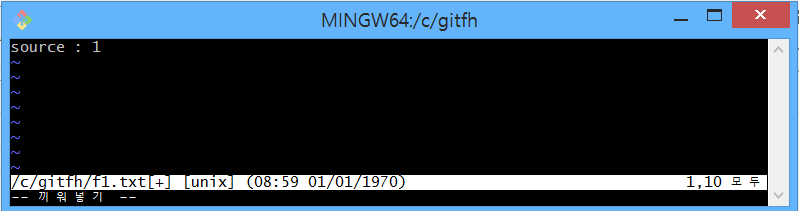
여기서는 Git이 관리할 대상으로 파일을 등록한다. vim이라는 텍스트편집기를 통해 .txt 파일을 생성 후 그 파일을 관리 대상으로 등록해보자.

Vim f1.txt<파일명> 명령을 통해 파일을 생성하여보자.

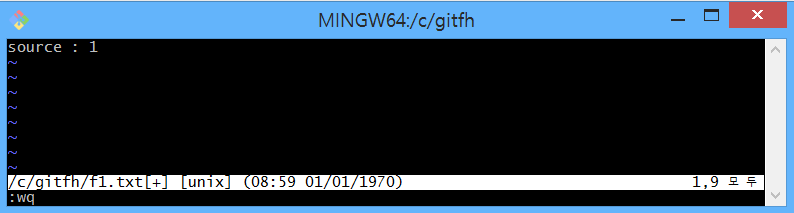


Vim 명령어를 실행하면 편집기 화면이 나오는데 여기에서는 I-insert (입력) w-write(쓰기,저장) q-quit(종료) 단축키로 동작한다. I를 눌러 입력 모드로 변경 후 ‘source : 1’이라는 텍스트를 입력후 esc키로 입력모드를 나온뒤 :wq 를 통하여 저장 및 종료를 하여보자.

i를 눌러 입력모드로 변경한다.



Esc 눌러 입력모드 종료 후 파일 저장 및 vim 편집기 종료한다.

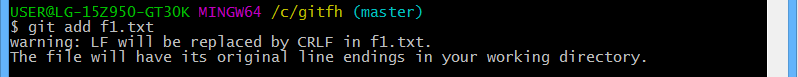


관리대상 폴더 상태 보자.(git status)

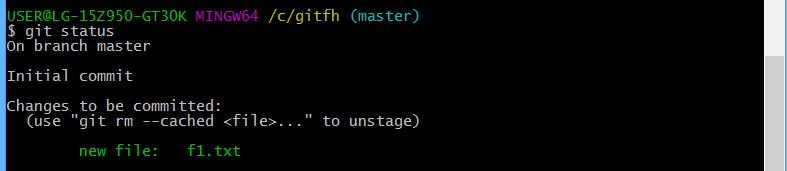


Untracked file – 추적 되지 않는 파일을 나타낸다.

git의 관리대상으로 추가하자.(git add f1.txt)



git의 관리대상 폴더 상태 확인한다.



## 버전 만들기(Commit)

여기서는 버전을 만들것이다. 그렇다면 버전이란 무엇일까? 버전이란, 어떠한 의미있는 작업이 완결된 상태를 나타낸다. 이러한 버전들을 git에서 관리할 탠데, 버전을 생성하기에 앞서서 git에 버버전 만들 때 누가 작업을 했는지 알려주기 위해 정보를 입력하자. 그리고 그후 버전을 만들어 보자.

버전에 포함될 버전을 만든 사람에 대한 정보를 설정한다.(처음에만 설정하면 된다.)



git config --global user.name "자신의 닉네임"

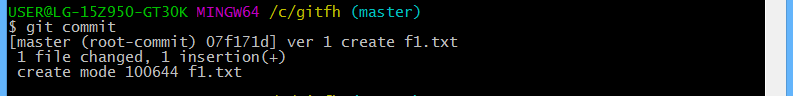
git config --global user.email "자신의 이메일"

버전을 만들어 보자.(git commit)



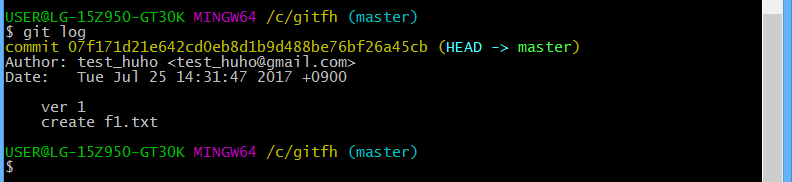
버전(commit) 메시지 입력하자.





무엇이 바뀌었는지, 해당 파일이 왜 변경되었는지에 대한 정보들을 입력한다.

버전 확인해 보자



## Stage area

여기에서는 Git이 버전관리 할 때, 파일들의 보관 영역에 대해 알아보자.

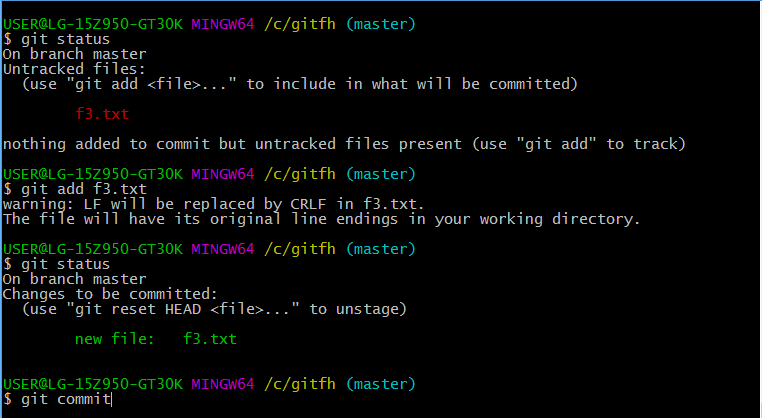
Git의 기초에서 보았듯이 Git은 워킹디렉토리 영역에 있는 파일들 중 Stage area영역에 올라온 파일들만은 버전(Commit)으로 만들어서 관리를 한다. 간단히 말하자면 Stage area란 commit 대기 파일들이 있는 영역을 뜻한다. 즉, git add 한 파일들이 있는 곳이다. 그리고 commit하는 순간 Stage area의 파일들은 저장소(repository)로 이동된다. 아래에서 이와 같은 동작을 실습해 보자.

현재 위 실습을 따라 했으면 워킹디렉토리에는 f1.txt 파일만 있을 것이다. 이제 ‘git add’의 동작에 따른 Stage area에 추가된 파일과 그렇지 않은 파일들을 구분하기 위해 f1.txt 파일을 복사할 것이다.

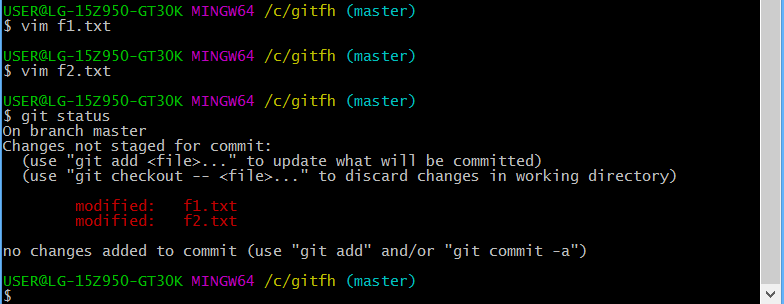


|  |  |
| --- | --- |
| 명령어 | 내용 |
| Ls –al | 현 디렉토리 현황을 보여줌 |
| Cp <복사할 파일명> <생성될 파일명> | 파일을 지정한 파일로 복사한다. |

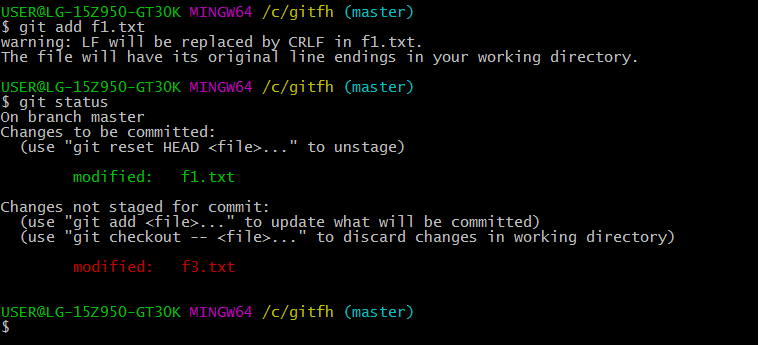
‘git status’ 명령으로 확인하면 ‘f2.txt’파일이 Untracked 상태 임을 알 수 있다. 이는 git의 관리 대상이 아니기 때문에 ‘git add’ 명령으로 Stage area에 추가한 후 버전을 생성 하자.



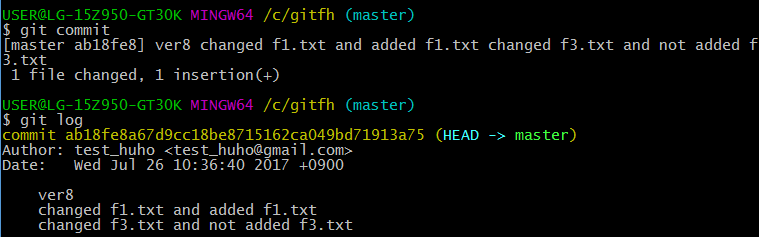
vim 편집기를 통하여 f1.txt와 f2.txt의 내용을 수정한 후 git의 상태(status)를 확인한다.



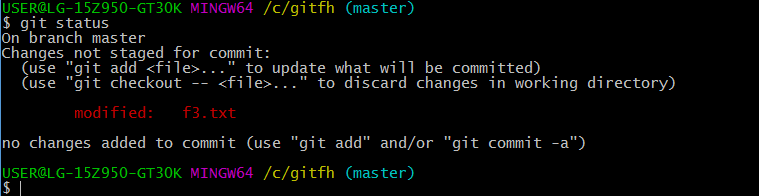
위와 같이 변경된 파일은 커밋에 대해 준비되지 않은 변경사항이다. 즉, 커밋시 해당 버전에 등록되지 않는 것이다. 이와 같이 변경이 있을 경우 ‘git add’ 명령을 통해 해당 사항들을 Stage are에 등록하여 알려주어야 한다. 그렇다면 ‘git add’ 명령으로 f1.txt 파일을 Stage area영역으로 추가해보자.



위와 같이 f1.txt 파일의 변경 사항이 등록되었음을 알 수 있다. 이번에는 버전을 만들어 보자.



위와 같이 새로운 버전이 생성됐음을 알 수 있다. 그러면 ‘git status’ 명령으로 상태를 확인해 보자.

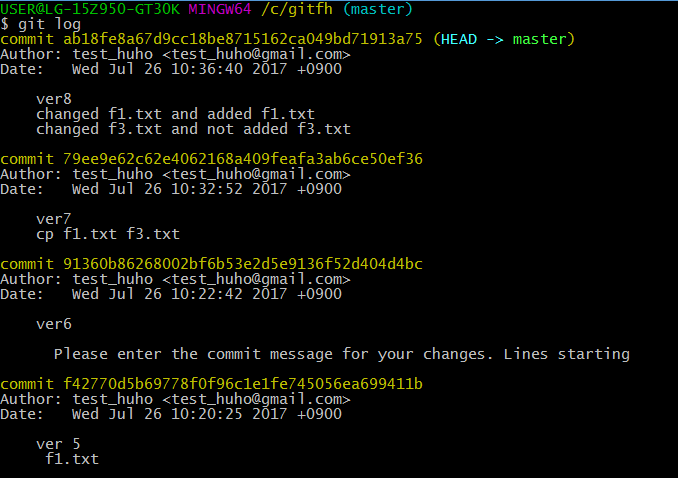


이처럼 Git은 ‘git add’명령을 통해 등록한 변경사항만을 버전으로 만들 수 있다.

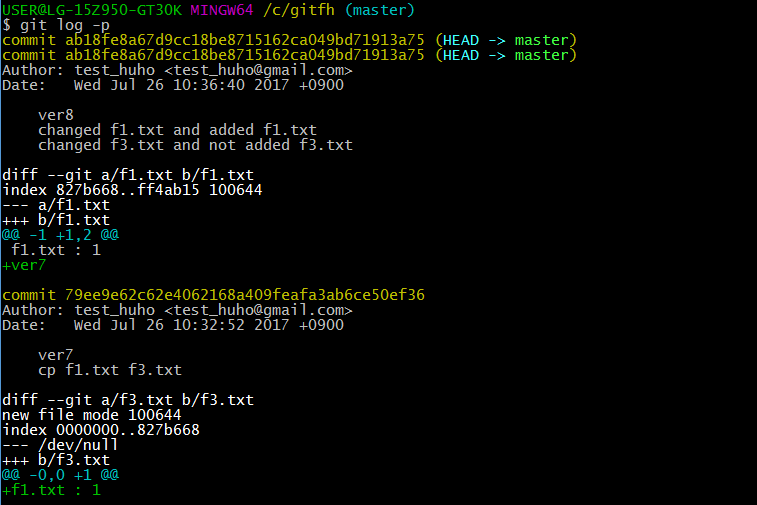
## 변경사항 확인하기(log)

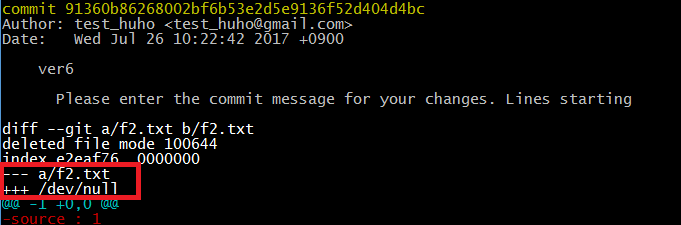
버전 관리의 유효점은 두가지가 있다. 버전의 차이점을 알 수 있다는 것과 과거로 돌아갈 수 있다는 것이다. 여기서는 차이점을 비교하는 것에 대해 알아보자.

‘git log’ 명령을 사용하면 버전의 목록이 출력된다.

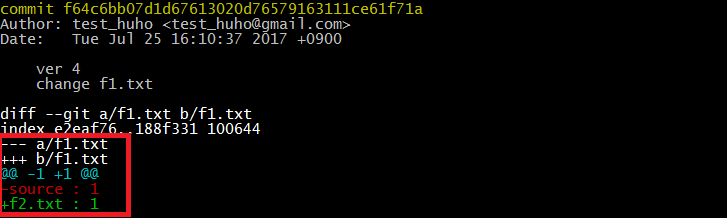


‘git log –p’ 명령을 통하면 로그에서 출력되는 버전 간의 차이점을 출력할 수 있다. 이때 내용은 크게 파일의 추가&삭제, 내용 변경 등이 있다.





위와 같이 ‘---a’(이전 버전)에서는 f2.txt파일이 존재하였는데 ‘+++’(현 버전)에서는 /dev/null(존재하지 않음)임을 알려준다.



위는 ‘---a’(이전 버전)에서 f1.txt파일의 내용이 source : 1 이였는데 ‘+++b’(현 버전)에서 f1.txt 파일의 내용이 f2.txt :1 으로 변경되었음을 알려준다.

위와 같이 로그를 통해 이전 버전과 차이를 알수도 있지만, 특정 버전간의 차이점을 비교할때가 더 많을 것이다. 이럴때 ‘git diff 버전id..버전id’ 명령을 사용하면 된다.



버전간의 차이점을 위와 같이 비교할수 있지만, git add하지 전과 add한 후의 파일 내용을 비교할 수도 있다. ‘git diff’ 명령을 사용하면 이전 ‘git add’한 상태의 파일들과 비교하여 준다.



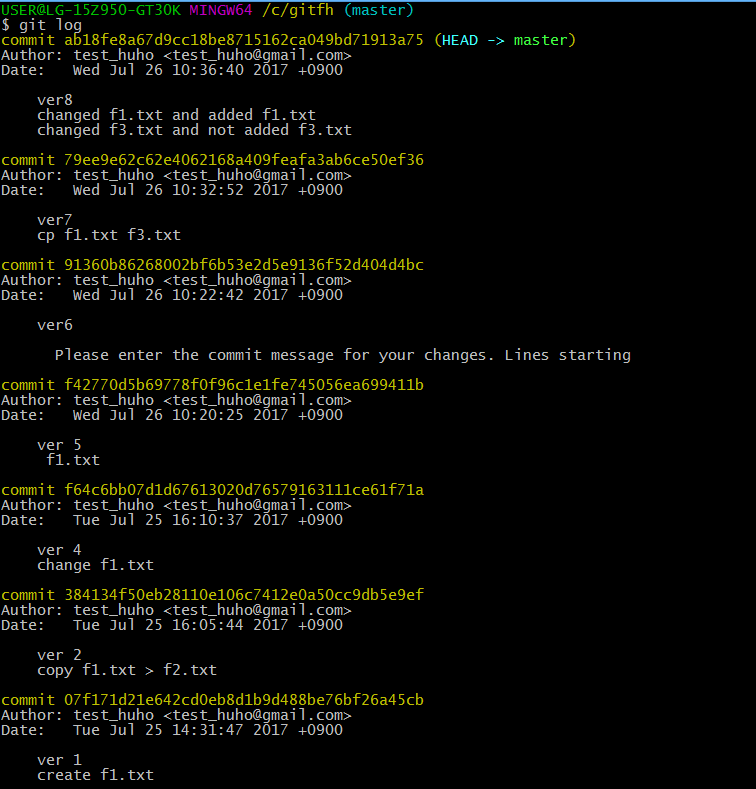
## 과거 버전으로 돌아가기(reset & revert)

과거 버전으로 돌아가는 방법에는 2가지가 있다. Reset 명령을 사용하는 것과 revert 명령을 사용하는 것이다. 그렇다면 reset과 revert의 차이점은 무엇일까. Reset은 해당 버전으로 돌아가면서 그 사이의 내역을 전부 없애는 것이고 revert는 해당 버전의 커밋을 취소하면서 새로운 버전으로 만드는 것이다.



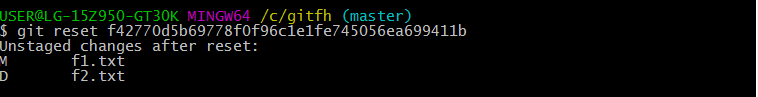
### Reset

‘git log’ 명령으로 현재 버전들을 확인해 보자.

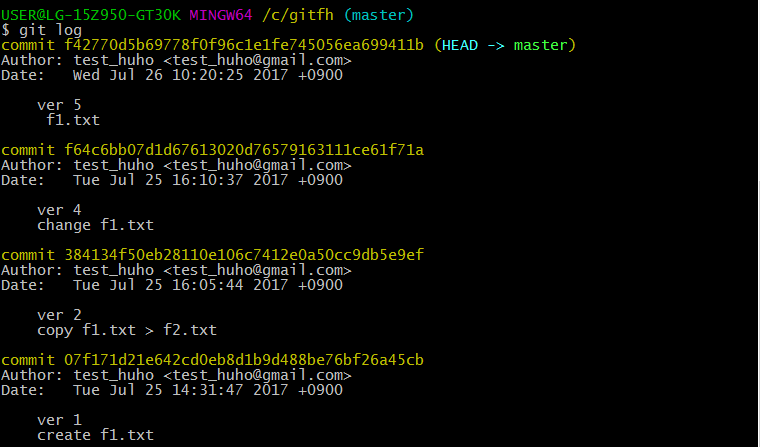


임의의 버전의 아이디를 선택하여 reset을 해보자

(예 : ver5 - f42770d5b69778f0f96c1e1fe745056ea699411b )



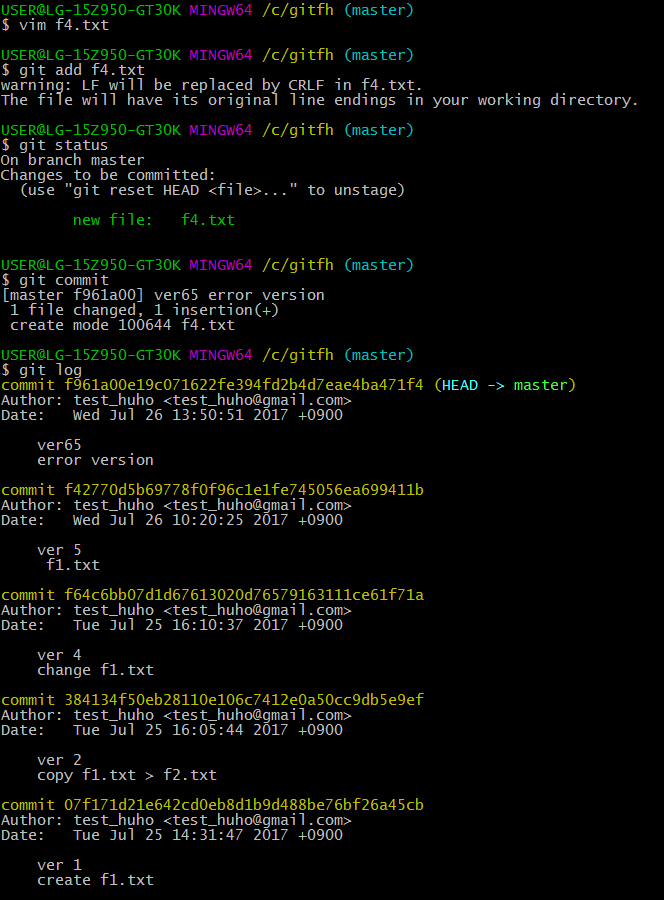
다시 ‘git log’ 명령으로 버전목록을 확인해보자



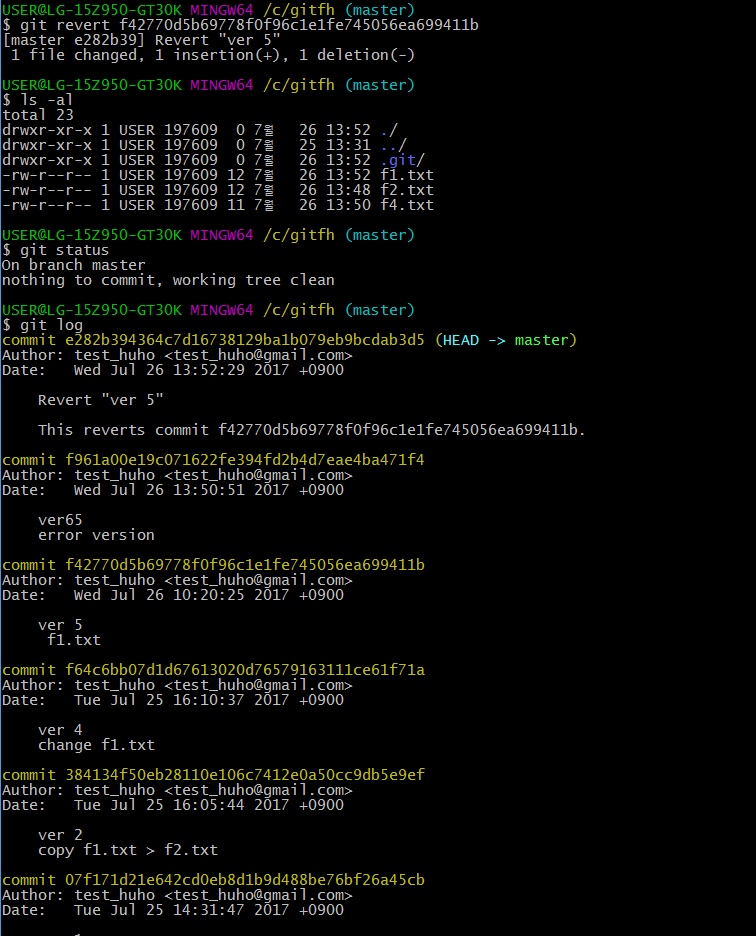
이처럼 ver5로 버전이 되돌아 가진걸 알수있다.

### Revert

현재에서 파일 f4.txt를 만들고 버전(ver65)을 만들어보자



위와 같이 버전 ver65가 문제가 있다고 할 때 문제가 생기기 전으로 돌아가야 할 것이다. 그럼 revert를 사용하여 보자.



이처럼 기존 버전의 내용을 지우지 않고 새로운 버전을 생성하는 것을 확인할 수 있다.

(참고 : reset과 revert의 개념을 잘 표현한 카툰이다. [http://www.popit.kr/](http://www.popit.kr/%EA%B0%9C%EB%B0%9C%EB%B0%94%EB%B3%B4%EB%93%A4-git-back-to-the-future/) )

## 브랜치 사용하기

브랜치란 작업의 분기라고 보면 된다. 예를 들어 기존의 프로젝트 진행중 고객사에 의해 새로운 기능을 추가 할 때 기존 프로젝트의 내용을 그대로 주고 새로운 작업을 해야 할 것이다. 그럴떄 브랜치를 나누어 작업을 시작하면 하나의 분기가 생성되어 진행되는 것이다.

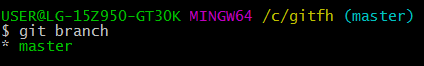


### 브랜치 만들기

브랜치에 대한 명령어는 다음과 같다.

|  |  |
| --- | --- |
| 명령어 | 내용 |
| Git branch | 브랜치 목록을 보여준다. |
| Git branch <브랜치 명> | 브랜치를 생성한다. |
| Git branch –d | 현재 사용중인 브랜치를 삭제한다. |
| Git branch -D  Git checkout <전환하려는 브랜치 명> | 병합하지 않은 브랜치를 강제 삭제한다.  브랜치를 전환한다. |
| Git checkout –b <생성하고 전환할 브랜치명> | 브랜치를 생성하고 전환한다. |

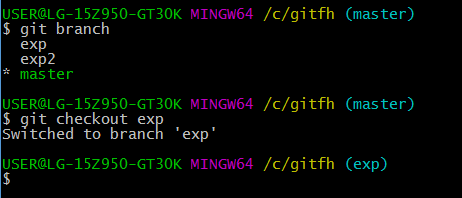
브랜치 목록 보기



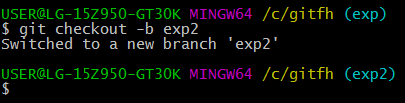
브랜치 생성하기



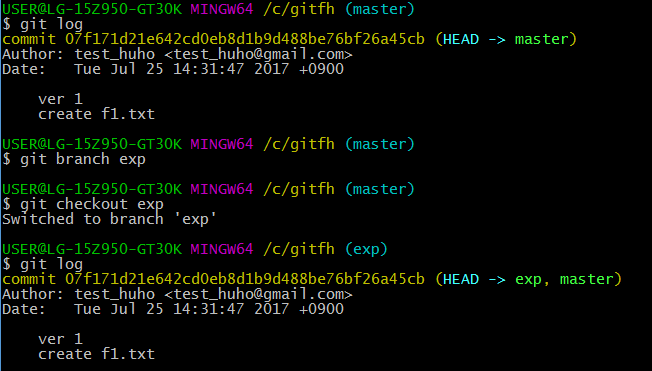
브랜치 전환하기



브랜치 생성하며 전환하기

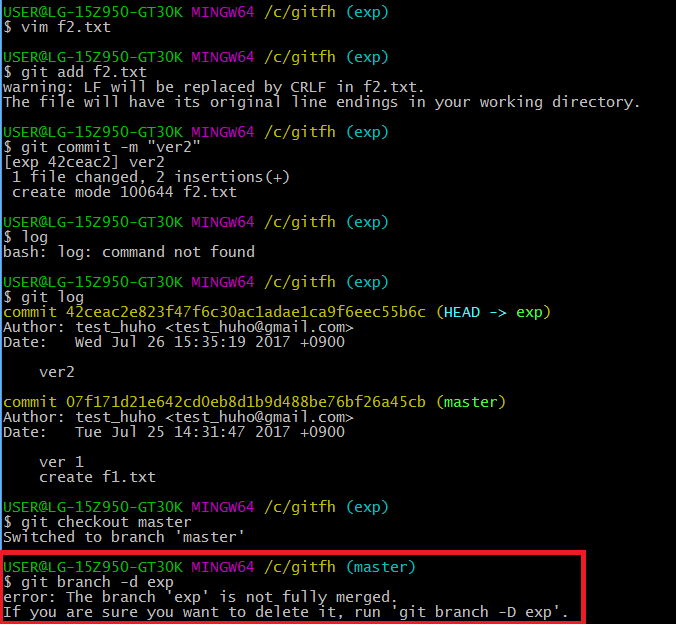


브랜치 삭제하기

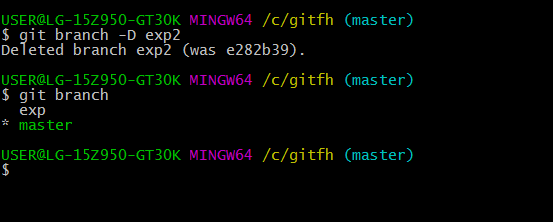


위 그림은 브랜치의 생성시 master와 exp가 같음을 볼 수 있다. 이때 새로운 브런치가 아무런 작업을 하지 않았으면 ‘-d’ 옵션으로 삭제가 가능하다. 만약 작업을 하였으면 두번째 그림과 같이 삭제되지 않을 것이다.

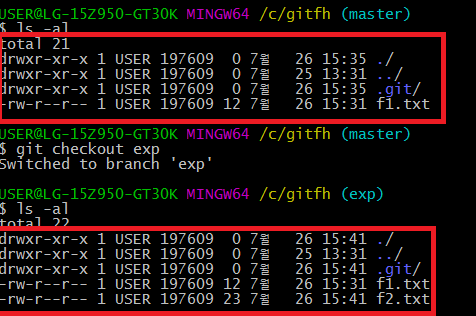




병합하지 않은 브랜치 삭제하기



이렇게 브랜치를 나누어 작업을 하는 이유는 아래 그림을 보면 쉽게 알 수 있다.



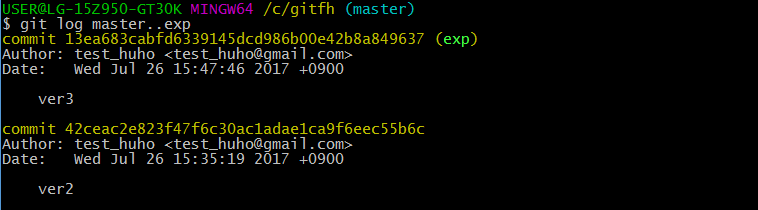
master에는 exp브랜치에서 작업한 f2.txt 파일이 보이지 않는다. 이처럼 작업이 분기 되었기에 해당 작업 파일들 관리도 따로 분기된 것이다.

### 브랜치 정보확인하기

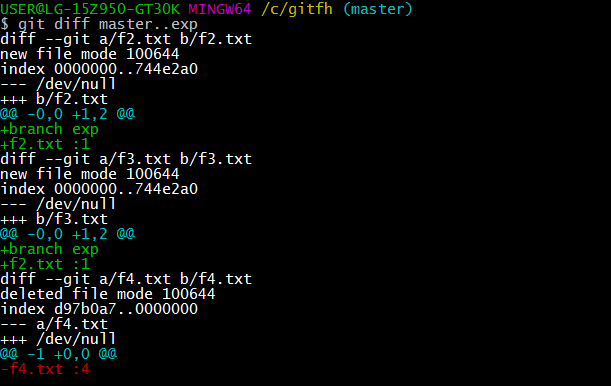
Git의 로컬에서 여러 브랜치가 존재할 경우 브랜치간의 정보를 비교, 확인이 필요할 경우가 있을있을 것이다. 이때 아래의 명령어를 통해 비교, 확인 가능하다.

|  |  |
| --- | --- |
| 명령어 | 내용 |
| Git log <브랜치명>..<브랜치명> | 브랜치간의 커밋 내용을 비교 |
| Git diff <브랜치명>..<브랜치명> | 브랜치 간의 파일 내용을 비교 |
| Git log –branches –decorate –graph --oneline | 로그의 모든 브랜치를 브랜치명, 그래프, 한줄로 표시 |

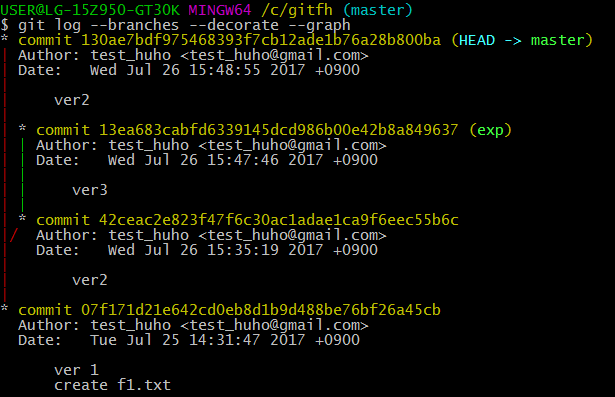
‘git log <브랜치1>..<브랜치2>’ 명령을 사용하면 브랜치1에 대한 브랜치2의 차이점을 보여준다.



‘git diff’ 명령을 사용하면 브랜치 간의 비교를 해주는데 아래그림을 보면 f2.txt와 f3.txt 파일은 exp브랜치에서 생성되었고, f4.txt는 없다는 것을 알려준다.



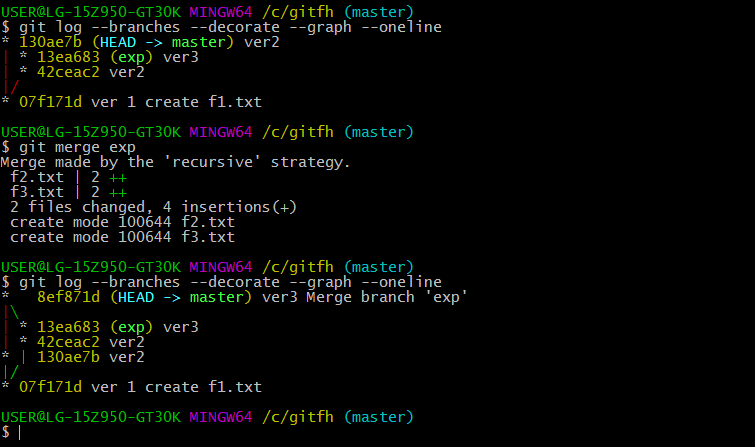
‘git log –branches –decorate –graph –oneline’ 명령을 이용하면 아래의 그림과 같이 한눈에 보기 편하다.



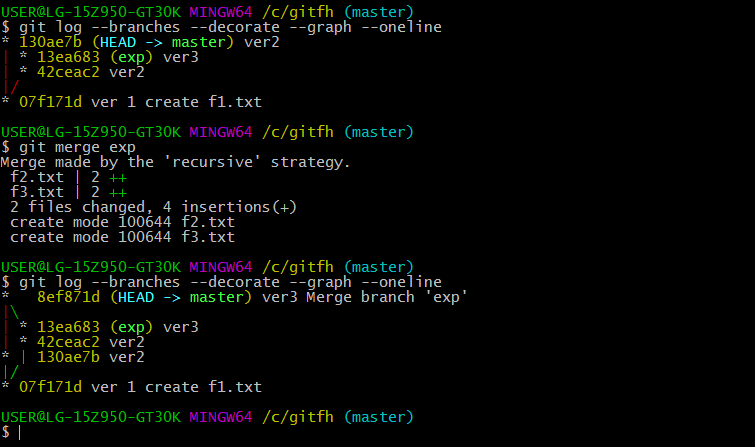
### 브랜치 병합하기

지금까지 각 분기(브랜치)에서 각자의 작업을 한 것을 알아보았다. 하지만 이렇게 분기된 작업을 하다 작업을 다시 합쳐야 하는 상황이 발생할수도 있다. 이럴때 두 작업을 병합(merge)을 하여야 한다. 여기서는 exp분기의 작업을 master로 병합하는 작업을 수행할 것이다.

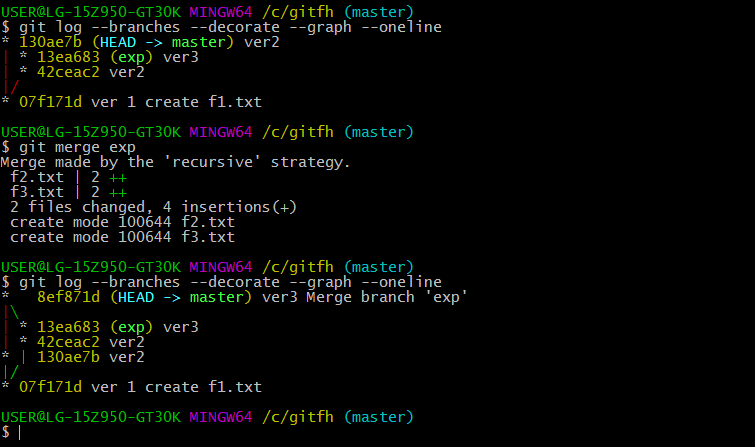
exp를 master로 병합해야 하므로 현재 사용중인 브랜치가 master인지 확인한다. 아닐 경우에는 ‘git checkout’ 명령을 통해 변경하자.



‘git merge’ 명령을 통해 병합을 하면 아래와 같이 작업이 자동적으로 이루어진다.



병합 후 확인해 보면 master브랜치의 버전이 130ae7b에서 멈춘 것이 아니라 exp와 병합되어 8ef871d 버전이 새로 생성되었음을 알 수 있다.

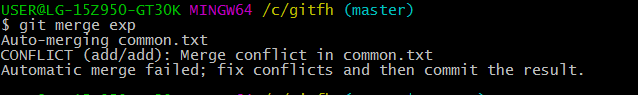


### 브랜치 병합 시 충돌이 발생할 경우

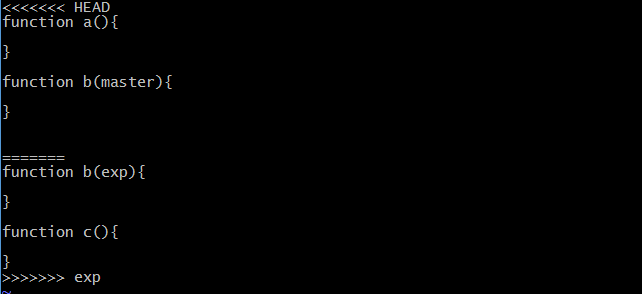
브랜치를 병합시 서로 같은 파일의 같은 부분을 수정하였을 경우 충돌이 발생한다. 이러할 경우 그 충돌난 부분을 git이 알려주는데, 그 곳(<<<HEAD === 충돌난브랜치명>>>)을 수정하여야 병합이 이루어진다. 아래와 같이 두 브랜치(master,exp)에 같은 파일이 있다고 하고 실행해보자.

|  |  |
| --- | --- |
| Common.txt | |
| master | exp |
| Function a(){  }  Function b(master){  } | Function b(exp){  }  Function c(){  } |

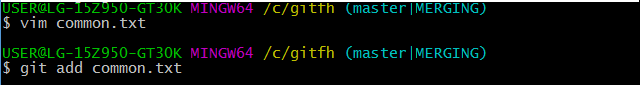
master에 exp를 병합하면 아래와 같이 충돌메세지가 나온다.



이때 해당 파일의 내용을 열어보면 충돌난 부분을 ‘=======’과 같이 표시하여 보여준다.



이 충돌난 부분을 겹치지 않게 수정하고 ‘git add’ 명령으로 Stage area에 등록하자.



‘git commit’ 명령으로 버전을 생성하면 master에 exp를 병합한 버전이 생성됨을 볼 수 있다.

