**GitHub Desktop 사용법**

**송 근 배**

TABLE OF CONTENTS

[1. 시작하기 8](#_Toc7389950)

[1.1. 버전 관리란? 8](#_Toc7389951)

[1.2. 짧게 보는 Git의 역사 12](#_Toc7389952)

[1.3. Git 기초 13](#_Toc7389953)

[1.4. CLI 17](#_Toc7389954)

[1.5. Git 설치 18](#_Toc7389955)

[1.6. Git 최초 설정 21](#_Toc7389956)

[1.7. 도움말 보기 24](#_Toc7389957)

[1.8. 요약 25](#_Toc7389958)

[2. Git의 기초 26](#_Toc7389959)

[2.1. Git 저장소 만들기 26](#_Toc7389960)

[2.2. 수정하고 저장소에 저장하기 29](#_Toc7389961)

[2.3. 커밋 히스토리 조회하기 42](#_Toc7389962)

[2.4. 되돌리기 49](#_Toc7389963)

[2.5. 리모트 저장소 52](#_Toc7389964)

[2.6. 태그 57](#_Toc7389965)

[2.7. Git Alias 63](#_Toc7389966)

[2.8. 요약 65](#_Toc7389967)

[3. Git 브랜치 66](#_Toc7389968)

[3.1. 브랜치란 무엇인가 66](#_Toc7389969)

[3.2. 브랜치와 Merge의 기초 68](#_Toc7389970)

[3.3. 브랜치 관리 68](#_Toc7389971)

[3.4. 브랜치 워크플로 68](#_Toc7389972)

[3.5. 리모트 브랜치 69](#_Toc7389973)

[3.6. Rebase 하기 69](#_Toc7389974)

[3.7. 요약 69](#_Toc7389975)

[4. GitHub Desktop 설치하기 70](#_Toc7389976)

[4.1. 다운로드 70](#_Toc7389977)

[5. Windows에 Git 설치 74](#_Toc7389978)

[5.1. Git 다운로드 74](#_Toc7389979)

[5.2. Git 설치 74](#_Toc7389980)

[5.3. 기본 설정 80](#_Toc7389981)

[6. 저장소(Repository) 생성하기 81](#_Toc7389982)

[7. 저장소(Repository) 불러오기 82](#_Toc7389983)

[8. 변경내용 커밋(Commit) 하기 86](#_Toc7389984)

[9. 커밋(Commit)한 내용을 브랜치에 Push하기 90](#_Toc7389985)

[10. 브랜치 풀(pull)하기 91](#_Toc7389986)

[11. 용어(Terms) 92](#_Toc7389987)

[12. 참고 사이트 93](#_Toc7389988)

[13. TroubleShooting 94](#_Toc7389989)

그림 목차

[[그림 1] 로컬 버전 관리 9](#_Toc7390084)

[[그림 2] 중앙 집중식 버전 관리(CVCS) 9](#_Toc7390085)

[[그림 3] 분산 버전 관리 시스템(DVCS) 10](#_Toc7390086)

[[그림 4] 각 파일에 대한 변화를 저장하는 시스템들 13](#_Toc7390087)

[[그림 5] 시간순으로 프로젝트의 스냅샷을 저장 13](#_Toc7390088)

[[그림 6] 워킹 트리, Staging Area, Git 디렉토리 15](#_Toc7390089)

[[그림 7] 파일의 라이프사이클 29](#_Toc7390090)

[[그림 8] 커밋과 트리 데이터 67](#_Toc7390091)

[[그림 9] 커밋과 이전 커밋 67](#_Toc7390092)

[[그림 10] 브랜치와 커밋 히스토리 68](#_Toc7390093)

[[그림 8] GitHub Desktop 다운로드 70](#_Toc7390094)

표 목차

APPENDIX

# 시작하기

## 버전 관리란?

이 장에서 설명하는 것은 Git을 처음 접하는 사람에게 필요한 내용이다. 먼저 버전 관리 도구에 대한 이해와 Git을 설치하는 방법을 설명하고 마지막으로 Git 서버를 설정하고 사용하는 방법을 설명한다. 이 장을 다 읽고 나면 Git 탄생 배경, Git을 사용하는 이유, Git을 설정하고 사용하는 방법을 터득하게 될 것이다.

**버전 관리란?**

“버전 관리” 는 무엇이고 우리는 왜 이것을 알아야 할까? 버전 관리 시스템은 파일 변화를 시간에 따라 기록했다가 나중에 특정 시점의 버전을 다시 꺼내올 수 있는 시스템이다. 이 책에서는 버전 관리하는 예제로 소프트웨어 소스 코드만 보여주지만, 실제로 거의 모든 컴퓨터 파일의 버전을 관리할 수 있다.

그래픽 디자이너나 웹 디자이너도 버전 관리 시스템(**VCS - Version Control System**)을 사용할 수 있다. VCS로 이미지나 레이아웃의 버전(변경 이력 혹은 수정 내용)을 관리하는 것은 매우 현명하다. VCS를 사용하면 각 파일을 이전 상태로 되돌릴 수 있고, 프로젝트를 통째로 이전 상태로 되돌릴 수 있고, 시간에 따라 수정 내용을 비교해 볼 수 있고, 누가 문제를 일으켰는지도 추적할 수 있고, 누가 언제 만들어낸 이슈인지도 알 수 있다. VCS를 사용하면 파일을 잃어버리거나 잘못 고쳤을 때도 쉽게 복구할 수 있다. 이런 모든 장점을 큰 노력 없이 이용할 수 있다.

**로컬 버전 관리**

많은 사람은 버전을 관리하기 위해 디렉토리로 파일을 복사하는 방법을 쓴다(똑똑한 사람이라면 디렉토리 이름에 시간을 넣을 거다). 이 방법은 간단하므로 자주 사용한다. 그렇지만, 정말 뭔가 잘못되기 쉽다. 작업하던 디렉토리를 지워버리거나, 실수로 파일을 잘못 고칠 수도 있고, 잘못 복사할 수도 있다.

이런 이유로 프로그래머들은 오래전에 로컬 VCS라는 걸 만들었다. 이 VCS는 아주 간단한 데이터베이스를 사용해서 파일의 변경 정보를 관리했다.



[그림 1] 로컬 버전 관리

많이 쓰는 VCS 도구 중에 RCS(Revision Control System)라고 부르는 것이 있는데 오늘날까지도 아직 많은 회사가 사용하고 있다. RCS는 기본적으로 Patch Set(파일에서 변경되는 부분)을 관리한다. 이 Patch Set은 특별한 형식의 파일로 저장한다. 그리고 일련의 Patch Set을 적용해서 모든 파일을 특정 시점으로 되돌릴 수 있다

**중앙집중식 버전 관리(CVCS)**

프로젝트를 진행하다 보면 다른 개발자와 함께 작업해야 하는 경우가 많다. 이럴 때 생기는 문제를 해결하기 위해 CVCS(중앙집중식 VCS)가 개발됐다. CVS, Subversion, Perforce 같은 시스템은 파일을 관리하는 서버가 별도로 있고 클라이언트가 중앙 서버에서 파일을 받아서 사용(Checkout)한다. 수년 동안 이러한 시스템들이 많은 사랑을 받았다.



[그림 2] 중앙 집중식 버전 관리(CVCS)

CVCS 환경은 로컬 VCS에 비해 장점이 많다. 모두 누가 무엇을 하고 있는지 알 수 있다. 관리자는 누가 무엇을 할지 꼼꼼하게 관리할 수 있다. 모든 클라이언트의 로컬 데이터베이스를 관리하는 것보다 VCS 하나를 관리하기가 훨씬 쉽다.

그러나 이 CVCS 환경은 몇 가지 치명적인 결점이 있다. 가장 대표적인 것이 중앙 서버에 발생한 문제다. 만약 서버가 한 시간 동안 다운되면 그동안 아무도 다른 사람과 협업할 수 없고 사람들이 하는 일을 백업할 방법도 없다. 그리고 중앙 데이터베이스가 있는 하드디스크에 문제가 생기면 프로젝트의 모든 히스토리를 잃는다. 물론 사람마다 하나씩 가진 스냅샷은 괜찮다. 로컬 VCS 시스템도 이와 비슷한 결점이 있고 이런 문제가 발생하면 모든 것을 잃는다.

**분산 버전 관리 시스템**

**DVCS(분산 버전 관리 시스템)**을 설명할 차례다. Git, Mecurial, Bazaar, Darcs 같은 DVCS에서의 클라이언트는 단순히 파일의 마지막 스냅샷을 Checkout 하지 않는다. 그냥 저장소를 히스토리와 더불어 전부 복제한다. 서버에 문제가 생기면 이 복제물로 다시 작업을 시작할 수 있다. 클라이언트 중에서 아무거나 골라도 서버를 복원할 수 있다. Clone은 모든 데이터를 가진 진정한 백업이다.



[그림 3] 분산 버전 관리 시스템(DVCS)

게다가 대부분의 DVCS 환경에서는 리모트 저장소가 존재한다. 리모트 저장소가 많을 수도 있다. 그래서 사람들은 동시에 다양한 그룹과 다양한 방법으로 협업할 수 있다. 계층 모델 같은 중앙집중식 시스템으로는 할 수 없는 워크플로를 다양하게 사용할 수 있다

## 짧게 보는 Git의 역사

우리네 삶의 삼라만상처럼 Git 또한 창조적 파괴와 활활 타오르는 갈등 속에서 시작됐다.

Linux 커널은 굉장히 규모가 큰 오픈소스 프로젝트다. Linux 커널의 삶 대부분은(1991–2002) Patch와 단순 압축 파일로만 관리했다. 2002년에 드디어 Linux 커널은 BitKeeper라고 불리는 상용 DVCS를 사용하기 시작했다.

2005년에 커뮤니티가 만드는 Linux 커널과 이익을 추구하는 회사가 개발한 BitKeeper의 관계는 틀어졌다. BitKeeper의 무료 사용이 제고된 것이다. 이 사건은 Linux 개발 커뮤니티(특히 Linux 창시자 Linus Torvalds)가 자체 도구를 만드는 계기가 됐다. Git은 BitKeeper를 사용하면서 배운 교훈을 기초로 아래와 같은 목표를 세웠다.

* 빠른 속도
* 단순한 구조
* 비선형적인 개발(수천 개의 동시 다발적인 브랜치)
* 완벽한 분산

Linux 커널 같은 대형 프로젝트에도 유용할 것(속도나 데이터 크기 면에서)

Git은 2005년 탄생하고 나서 아직도 초기 목표를 그대로 유지하고 있다. 그러면서도 사용하기 쉽게 진화하고 성숙했다. Git은 미친 듯이 빨라서 대형 프로젝트에 사용하기도 좋다. Git은 동시다발적인 브랜치에도 끄떡없는 슈퍼 울트라 브랜칭 시스템이다([Git 브랜치](https://git-scm.com/book/ko/v2/ch00/ch03-git-branching) 참고).

## Git 기초

Git의 핵심은 뭘까? 이 질문은 Git을 이해하는데 굉장히 중요하다. Git이 무엇이고 어떻게 동작하는지 이해한다면 쉽게 Git을 효과적으로 사용할 수 있다. Git을 배우려면 Subversion이나 Perforce 같은 다른 VCS를 사용하던 경험을 버려야 한다. Git은 미묘하게 달라서 다른 VCS에서 쓰던 개념으로는 헷갈린다. 사용자 인터페이스는 매우 비슷하지만, 정보를 취급하는 방식이 다르다. 이런 차이점을 이해하면 Git을 사용하는 것이 어렵지 않다.

**차이가 아니라 스냅샷**

Subversion과 Subversion 비슷한 놈들과 Git의 가장 큰 차이점은 데이터를 다루는 방법에 있다. 큰 틀에서 봤을 때 VCS 시스템 대부분은 관리하는 정보가 파일들의 목록이다. CVS, Subversion, Perforce, Bazaar 등의 시스템은 각 파일의 변화를 시간순으로 관리하면서 파일들의 집합을 관리한다(보통 델타 기반 버전관리 시스템이라 함).



[그림 4] 각 파일에 대한 변화를 저장하는 시스템들

Git은 이런 식으로 데이터를 저장하지도 취급하지도 않는다. 대신 Git은 데이터를 파일 시스템 스냅샷의 연속으로 취급하고 크기가 아주 작다. Git은 커밋하거나 프로젝트의 상태를 저장할 때마다 파일이 존재하는 그 순간을 중요하게 여긴다. 파일이 달라지지 않았으면 Git은 성능을 위해서 파일을 새로 저장하지 않는다. 단지 이전 상태의 파일에 대한 링크만 저장한다. Git은 데이터를 **스냅샷의 스트림**처럼 취급한다.



[그림 5] 시간순으로 프로젝트의 스냅샷을 저장

이것이 Git이 다른 VCS와 구분되는 점이다. 이점 때문에 Git은 다른 시스템들이 과거로부터 답습해왔던 버전 컨트롤의 개념과 다르다는 것이고 많은 부분을 새로운 관점에서 바라본다. Git은 강력한 도구를 지원하는 작은 파일시스템이다. Git은 단순한 VCS가 아니다. [Git 브랜치](https://git-scm.com/book/ko/v2/ch00/ch03-git-branching)에서 설명할 Git 브랜치를 사용하면 얻게 되는 이득이 무엇인지 설명한다.

**거의 모든 명령을 로컬에서 실행**

거의 모든 명령이 로컬 파일과 데이터만 사용하기 때문에 네트워크에 있는 다른 컴퓨터는 필요 없다. 대부분의 명령어가 네트워크의 속도에 영향을 받는 CVCS에 익숙하다면 Git이 매우 놀라울 것이다. Git의 이런 특징에서 나오는 미칠듯한 속도는 오직 Git느님만이 구사할 수 있는 전능이다. 프로젝트의 모든 히스토리가 로컬 디스크에 있기 때문에 모든 명령이 순식간에 실행된다.

예를 들어 Git은 프로젝트의 히스토리를 조회할 때 서버 없이 조회한다. 그냥 로컬 데이터베이스에서 히스토리를 읽어서 보여 준다. 그래서 눈 깜짝할 사이에 히스토리를 조회할 수 있다. 어떤 파일의 현재 버전과 한 달 전의 상태를 비교해보고 싶을 때도 Git은 그냥 한 달 전의 파일과 지금의 파일을 로컬에서 찾는다. 파일을 비교하기 위해 리모트에 있는 서버에 접근하고 나서 예전 버전을 가져올 필요가 없다.

즉 오프라인 상태이거나 VPN에 연결하지 못해도 막힘 없이 일 할 수 있다. 비행기나 기차 등에서 작업하고 네트워크에 접속하고 있지 않아도 커밋할 수 있다(*로컬* 저장소라는 점이 기억나는지). 다른 VCS 시스템에서는 불가능한 일이다. Perforce를 예로 들자면 서버에 연결할 수 없을 때 할 수 있는 일이 별로 없다. Subversion이나 CVS에서도 마찬가지다. 오프라인이기 때문에 데이터베이스에 접근할 수 없어서 파일을 편집할 수는 있지만, 커밋할 수 없다. 매우 사소해 보이지만 실제로 이 상황에 부닥쳐보면 느껴지는 차이가 매우 크다.

**Git의 무결성**

Git은 데이터를 저장하기 전에 항상 체크섬을 구하고 그 체크섬으로 데이터를 관리한다. 그래서 체크섬을 이해하는 Git 없이는 어떠한 파일이나 디렉토리도 변경할 수 없다. 체크섬은 Git에서 사용하는 가장 기본적인(Atomic) 데이터 단위이자 Git의 기본 철학이다. Git 없이는 체크섬을 다룰 수 없어서 파일의 상태도 알 수 없고 심지어 데이터를 잃어버릴 수도 없다.

Git은 SHA-1 해시를 사용하여 체크섬을 만든다. 만든 체크섬은 40자 길이의 16진수 문자열이다. 파일의 내용이나 디렉토리 구조를 이용하여 체크섬을 구한다. SHA-1은 아래처럼 생겼다.

24b9da6552252987aa493b52f8696cd6d3b00373

Git은 모든 것을 해시로 식별하기 때문에 이런 값은 여기저기서 보인다. 실제로 Git은 파일을 이름으로 저장하지 않고 해당 파일의 해시로 저장한다.

**Git은 데이터를 추가할 뿐**

Git으로 무얼 하든 Git 데이터베이스에 데이터가 추가 된다. 되돌리거나 데이터를 삭제할 방법이 없다. 다른 VCS처럼 Git도 커밋하지 않으면 변경사항을 잃어버릴 수 있다. 하지만, 일단 스냅샷을 커밋하고 나면 데이터를 잃어버리기 어렵다.

Git을 사용하면 프로젝트가 심각하게 망가질 걱정 없이 매우 즐겁게 여러 가지 실험을 해 볼 수 있다. [되돌리기](https://git-scm.com/book/ko/v2/ch00/_undoing)을 보면 Git에서 데이터를 어떻게 저장하고 손실을 어떻게 복구하는지 알 수 있다.

**세 가지 상태**

이 부분은 중요하기에 집중해서 읽어야 한다. Git을 공부하기 위해 반드시 짚고 넘어가야 할 부분이다. Git은 파일을 Committed, Modified, Staged 이렇게 세 가지 상태로 관리한다.

* **Committed**란 데이터가 로컬 데이터베이스에 안전하게 저장됐다는 것을 의미한다.
* **Modified**는 수정한 파일을 아직 로컬 데이터베이스에 커밋하지 않은 것을 말한다.
* **Staged**란 현재 수정한 파일을 곧 커밋할 것이라고 표시한 상태를 의미한다.

이 세 가지 상태는 Git 프로젝트의 세 가지 단계와 연결돼 있다. Git 디렉토리, 워킹 트리, Staging Area 이렇게 세 가지 단계를 이해하고 넘어가자.



[그림 6] 워킹 트리, Staging Area, Git 디렉토리

**Git 디렉토리**는 Git이 프로젝트의 메타데이터와 객체 데이터베이스를 저장하는 곳을 말한다. 이 Git 디렉토리가 Git의 핵심이다. 다른 컴퓨터에 있는 저장소를 Clone 할 때 Git 디렉토리가 만들어진다.

**워킹 트리**는 프로젝트의 특정 버전을 Checkout 한 것이다. Git 디렉토리는 지금 작업하는 디스크에 있고 그 디렉토리 안에 압축된 데이터베이스에서 파일을 가져와서 워킹 트리를 만든다.

**Staging Area**는 Git 디렉토리에 있다. 단순한 파일이고 곧 커밋할 파일에 대한 정보를 저장한다. Git에서는 기술용어로는 “Index” 라고 하지만, “Staging Area” 라는 용어를 써도 상관 없다.

Git으로 하는 일은 기본적으로 아래와 같다.

1. 워킹 트리에서 파일을 수정한다.
2. Staging Area에 파일을 Stage 해서 커밋할 스냅샷을 만든다. 모든 파일을 추가할 수도 있고 선택하여 추가할 수도 있다.
3. 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. [Git의 기초](https://git-scm.com/book/ko/v2/ch00/ch02-git-basics-chapter)에서 이 상태에 대해 좀 더 자세히 배운다. 특히 Staging Area를 이용하는 방법부터 아예 생략하는 방법까지도 설명한다.

## CLI

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

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

## Git 설치

Git을 사용하려면 우선 설치해야 한다. 이미 설치했으면 최신 버전으로 업데이트하는 게 좋다. 패키지를 설치하거나 별도의 인스톨러로 설치할 수 있다. 아니면 직접 소스코드를 내려받아서 컴파일할 수도 있다.

이 책은 Git **2.0.0** 버전을 기준으로 썼다. 대부분의 명령어는 그 이전 버전에서도 잘 동작하지만, 몇 가지 기능은 아예 없거나 미묘하게 다를 수 있다. Git의 하위 호환성은 정말 훌륭하기 때문에 2.0 이후 버전에서는 잘 동작한다.

**Linux에 설치**

Linux에서 패키지로 Git을 설치할 때는 보통 각 배포판에서 사용하는 패키지 관리도구를 사용하여 설치한다. Fedora(또는 비슷하게 RPM 기반 패키지 시스템을 사용하는 RHEL, CentOS)에서는 아래와 같이 `dnf`를 사용 한다.

$ sudo dnf install git-all

Ubuntu등의 데비안 계열 배포판에서는 `apt`를 사용한다.

$ sudo apt install git-all

다른 Unix 배포판에 설치하려면 <http://git-scm.com/download/linux> 에서 확인하라.

**Mac에 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 시작된다. 'git’이 설치돼 있지 않으면 설치하라고 안내해준다.

$ git --version

만약 Git이 시스템에 설치되어있지 않은 경우, 설치할 수 있도록 안내 메시지가 뜰 것이다.

좀 더 최신 버전이 필요하면 바이너리 인스톨러로 설치할 수 있다. macOS용 Git 인스톨러는 Git 웹사이트에서 관리하고 있으며 <http://git-scm.com/download/mac> 에서 내려받는다.



'GitHub for Mac’을 설치하는 방법도 있다. 이 도구에도 CLI 도구를 설치하는 옵션이 있다. 'GitHub for Mac’은 [http://mac.github.com](http://mac.github.com/)에서 내려받는다.

**Windows에 설치**

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

자동화된 설치 방식은 [Git Chocolatey 패키지](https://chocolatey.org/packages/git)를 통해 이용해볼 수 있다. 패키지는 커뮤니티에 의해 운영되는 프로그램인 점을 알려드린다.

Windows에서도 Git을 사용하는 또 다른 방법으로 **'GitHub Desktop’**을 설치하는 방법이 있다. 이 인스톨러는 CLI와 GUI를 모두 설치해준다. 설치하면 Git을 Powershell에서 사용할 수 있다. 인증정보(Credential) 캐싱과 CRLF 설정까지 잘 된다. 이런 것들은 차차 배우게 될 것인데, Git 사용자라면 쓰게 될 기능들이다. 'GitHub Desktop’은 [GitHub Desktop 웹사이트](http://desktop.github.com/)에서 내려받는다.

**소스코드로 설치하기**

가장 최신 버전이 필요하면 소스코드로 설치하는 것이 좋다. 바이너리 인스톨러는 약간 늦는다. 최근 Git은 매우 성숙해서 변경이 크지 않지만, 그래도 약간 차이 난다.

Git을 설치하려면 Git이 의존하고 있는 라이브러리인 autotools, curl, zlib, openssl, expat, libiconv등이 필요하다. 예를 들어 `dnf`을 사용하는 Fedora등의 시스템이나 `apt-get`이 있는 데비안 계열 시스템이면 아래 명령어 중 하나를 실행하여 필요한 패키지를 설치할 수 있다.

$ sudo dnf install dh-autoreconf curl-devel expat-devel gettext-devel \

openssl-devel perl-devel zlib-devel

$ sudo apt-get install dh-autoreconf libcurl4-gnutls-dev libexpat1-dev \

gettext libz-dev libssl-dev

문서를 다양한(doc, html, info) 형식으로 추가하려면 다음의 패키지들이 추가로 필요하다(주의: CentOS나 Scientific Linux 같은 RHEL 계열 사용자는 [EPEL 저장소를 켜고](https://fedoraproject.org/wiki/EPEL#How_can_I_use_these_extra_packages.3F) docbook2X 패키지를 다운로드해야 한다.)

$ sudo dnf install asciidoc xmlto docbook2X

$ sudo apt-get install asciidoc xmlto docbook2x

데비안 계열 시스템이면 install-info 패키지도 설치해야 한다.

$ sudo apt-get install install-info

RPM을 사용하는 Fedora나 비슷한 시스템을 사용한다면 getopt 패키지도 설치해야 한다(이는 데비안 계열 시스템에는 이미 설치되어 있다).

$ sudo dnf install getopt

$ sudo apt-get install getopt

Fedora/RHEL/RHEL 계열 사용자라면 아래 명령도 필요하다.

$ sudo ln -s /usr/bin/db2x\_docbook2texi /usr/bin/docbook2x-texi

실행파일 이름이 달라서 그렇다.

모든 준비가 완료되면, 최신 배포 버전을 가져와야 한다.

Kernel.org(<https://www.kernel.org/pub/software/scm/git>)에서 내려받을 수도 있고 GitHub에 있는 미러(<https://github.com/git/git/releases>)에서도 받을 수도 있다. 보통 GitHub 페이지에서 최신 버전을 내려받는 것이 더 간단하지만 kernel.org에는 배포 시그너처가 있어서 내려받은 것을 검증할 수 있다.

이제, 컴파일하고 설치하자.

$ tar -zxf git-2.0.0.tar.gz

$ cd git-2.0.0

$ make configure

$ ./configure --prefix=/usr

$ make all doc info

$ sudo make install install-doc install-html install-info

설치하고 나면 Git을 사용하여 Git 소스코드를 받아서 수정할 수도 있다.

$ git clone git://git.kernel.org/pub/scm/git/git.git

## Git 최초 설정

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

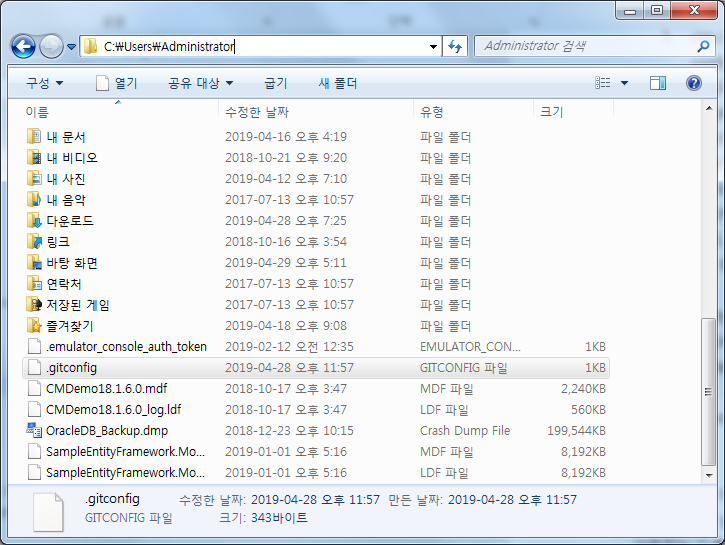
'git config’라는 도구로 설정 내용을 확인하고 변경할 수 있다. Git은 이 설정에 따라 동작한다. 이때 사용하는 설정 파일은 세 가지나 된다.

1. /etc/gitconfig 파일: 시스템의 모든 사용자와 모든 저장소에 적용되는 설정이다. git config --system 옵션으로 이 파일을 읽고 쓸 수 있다. (이 파일은 시스템 전체 설정파일이기 때문에 수정하려면 시스템의 관리자 권한이 필요하다.)
2. ~/.gitconfig, ~/.config/git/config 파일: 특정 사용자(즉 현재 사용자)에게만 적용되는 설정이다. git config --global 옵션으로 이 파일을 읽고 쓸 수 있다. 특정 사용자의 모든 저장소 설정에 적용된다.
3. .git/config : 이 파일은 Git 디렉토리에 있고 특정 저장소(혹은 현재 작업 중인 프로젝트)에만 적용된다. --local 옵션을 사용하면 이 파일을 사용하도록 지정할 수 있다. 하지만 기본적으로 이 옵션이 적용되어 있다. (당연히, 이 옵션을 적용하려면 Git 저장소인 디렉토리로 이동 한 후 적용할 수 있다.)

각 설정은 역순으로 우선시 된다. 그래서 .git/config 가 /etc/gitconfig 보다 우선한다.

Windows에서는 $HOME 디렉토리에서 .gitconfig 파일을 찾는다(아마도 C:\Users\$USER 디렉토리). Windows에서도 /etc/gitconfig 파일은 그 경로에서 찾는다.

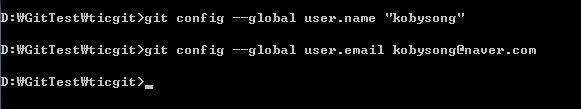
Windows에서 .gitconfig 파일 경로



이 경로는 아마도 MSys 루트의 상대경로일 텐데, MSys 루트는 인스톨러로 Git을 Windows에 설치할 때 결정된다. Git for Windows 2.x 버전에서는 조금 다르다. Windows XP 사용자는 C:\Documents and Settings\All Users\Application Data\Git\config 디렉토리에서 찾을 수 있고 Windows Vista 이후 버전 사용자는 C:\ProgramData\Git\config 에서 찾을 수 있다. 이 시스템 설정 파일의 경로는 git config -f <file> 명령으로 변경할 수 있다. 관리자 권한이 필요하다.

**사용자 정보**

Git을 설치하고 나서 가장 먼저 해야 하는 것은 사용자이름과 이메일 주소를 설정하는 것이다. Git은 커밋할 때마다 이 정보를 사용한다. 한 번 커밋한 후에는 정보를 변경할 수 없다.



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

GUI 도구들은 처음 실행할 때 이 설정을 묻는다.

**편집기**

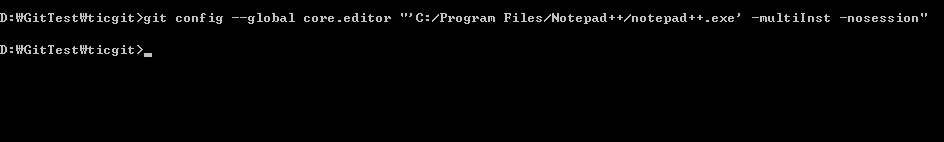
사용자 정보를 설정하고 나면 Git에서 사용할 텍스트 편집기를 고른다. 기본적으로 Git은 시스템의 기본 편집기를 사용한다.

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

$ git config --global core.editor emacs

Windows 사용자라면 다른 텍스트 편집기를 사용할 수 있다. 실행파일의 전체 경로를 설정해주면 된다. 실행파일의 전체 경로는 사용하는 편집기에 따라 다르다.

Windows 환경에서 많이 사용되는 Notepad 편집기의 경우 주로 32비트 버전을 사용하게 된다. 현재 기준으로 64비트 버전을 사용하면 동작하지 않는 플러그인이 많다. 32비트 Windows 시스템이거나, 64비트 Windows 시스템에서 64비트 Notepad을 설치했다면 다음과 같이 설정한다.



64비트 Windows 시스템에서 32비트 Notepad++을 설치했다면 `C:\Program Files (x86)`에 설치된다.

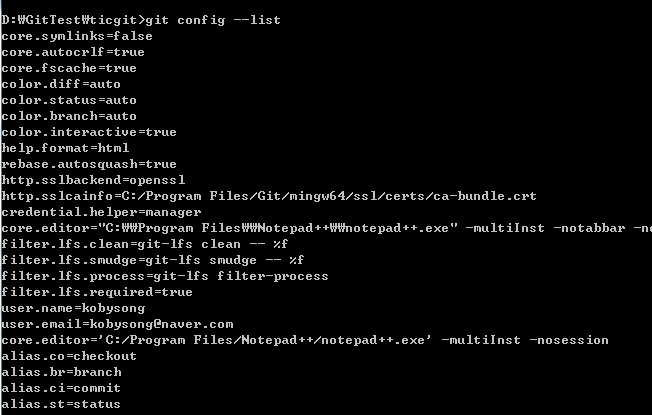
$ git config --global core.editor "'C:/Program Files (x86)/Notepad++/notepad++.exe' -multiInst -nosession"

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



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

|  |  |
| --- | --- |
| Note | Git이 설정된 값을 읽을 때 여러 파일에서 동일한 키에 대해 다른 값을 설정하고 있을 수 있다. 값이 기대한 값과 다를 수 있는데 값만 보고 쉽게 그 원인을 찾을 수 없다. 이 때 키에 설정된 값이 *어디에서* 설정되었는지 확인할 수 있는데 다음과 같은 명령으로 어떤 파일로부터 설정된 값인지를 확인할 수 있다.  $ git config --show-origin rerere.autoUpdate  file:/home/johndoe/.gitconfig false |

## 도움말 보기

명령어에 대한 도움말이 필요할 때 도움말을 보는 방법은 두 가지로 동일한 결과를 볼 수 있다.

$ git help <verb>

$ man git-<verb>

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

$ git help config

도움말은 언제 어디서나 볼 수 있다. 오프라인으로도 볼 수 있다. 도움말과 이 책으로 부족하면 다른 사람의 도움을 받는 것이 필요하다. Freenode IRC 서버(irc.freenode.net)에 있는 #git 이나 #github 채널로 찾아가라. 이 채널에는 보통 수백 명의 사람이 접속해 있다. 모두 Git에 대해 잘 알고 있다. 기꺼이 도와줄 것이다.

Git 명령을 사용하기 위해 매우 자세한 도움말 전체를 볼 필요 없이 각 명령에서 사용할 수 있는 옵션들에 대해서 간략히 살펴볼수도 있다. -h, --help 옵션을 사용하면 다음과 같이 Git 명령에서 사용할 수 있는 옵션들에 대한 간단한 도움말을 출력한다.

$ git add -h

usage: git add [<options>] [--] <pathspec>...

-n, --dry-run dry run

-v, --verbose be verbose

-i, --interactive interactive picking

-p, --patch select hunks interactively

-e, --edit edit current diff and apply

-f, --force allow adding otherwise ignored files

-u, --update update tracked files

-N, --intent-to-add record only the fact that the path will be added later

-A, --all add changes from all tracked and untracked files

--ignore-removal ignore paths removed in the working tree (same as --no-all)

--refresh don't add, only refresh the index

--ignore-errors just skip files which cannot be added because of errors

--ignore-missing check if - even missing - files are ignored in dry run

--chmod <(+/-)x> override the executable bit of the listed files

## 요약

우리는 Git이 무엇이고 지금까지 사용해 온 다른 CVCS와 어떻게 다른지 배웠다. 시스템에 Git을 설치하고 사용자 정보도 설정했다. 다음 장에서는 Git의 사용법을 배운다.

# Git의 기초

Git을 사용하는 방법을 알고 싶은데 한 챕터밖에 읽을 시간이 없다면 이번 챕터를 읽어야 한다. Git에서 자주 사용하는 명령어는 모두 2장에 등장한다. 2장을 다 읽으면 저장소를 만들고 설정하는 방법, 파일을 추적하거나(Track) 추적을 그만두는 방법, 변경 내용을 Stage 하고 커밋하는 방법을 알게 된다. 파일이나 파일 패턴을 무시하도록 Git을 설정하는 방법, 실수를 쉽고 빠르게 만회하는 방법, 프로젝트 히스토리를 조회하고 커밋을 비교하는 방법, 리모트 저장소에 Push 하고 Pull 하는 방법을 살펴본다.

## Git 저장소 만들기

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

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

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

**기존 디렉토리를 Git 저장소로 만들기**

버전관리를 하지 아니하는 기존 프로젝트를 Git으로 관리하고 싶은 경우 우선 프로젝트의 디렉토리로 이동한다. 이러한 과정을 처음 해보는 것이라면 시스템마다 조금 다른 점을 주의하자.

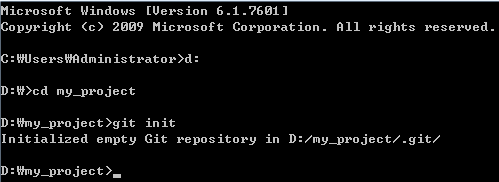
Linux:

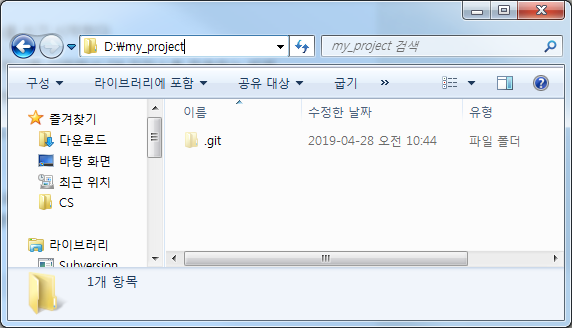
$ cd /home/user/my\_project

Mac:

$ cd /Users/user/my\_project

**Windows:**



\

이 명령은 .git 이라는 하위 디렉토리를 만든다. .git 디렉토리에는 저장소에 필요한 뼈대 파일(Skeleton)이 들어 있다. 이 명령만으로는 아직 프로젝트의 어떤 파일도 관리하지 않는다. (.git 디렉토리가 막 만들어진 직후에 정확히 어떤 파일이 있는지에 대한 내용은 [Git의 내부](https://git-scm.com/book/ko/v2/ch00/ch10-git-internals)에서 다룬다)

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

$ git add \*.c

$ git add LICENSE

$ git commit -m 'initial project version'

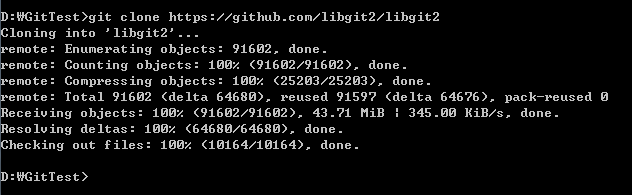
명령어 몇 개로 순식간에 Git 저장소를 만들고 파일 버전 관리를 시작했다.

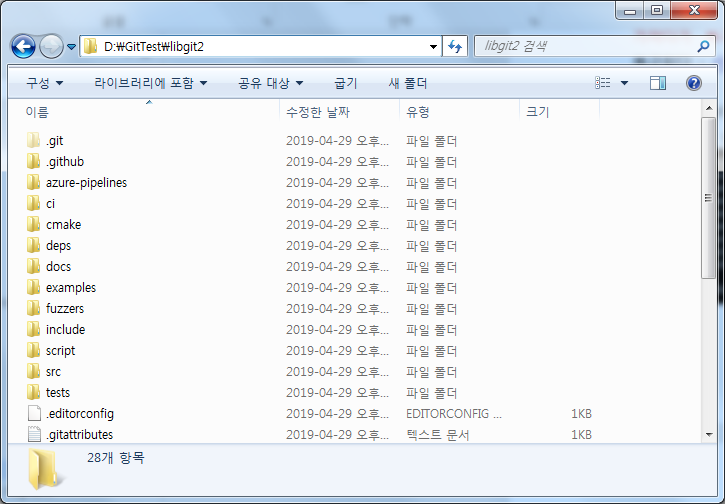
**기존 저장소를 Clone 하기**

다른 프로젝트에 참여하려거나(Contribute) Git 저장소를 복사하고 싶을 때 git clone 명령을 사용한다. 이미 Subversion 같은 VCS에 익숙한 사용자에게는 "checkout" 이 아니라 "clone" 이라는 점이 도드라져 보일 것이다. Git이 Subversion과 다른 가장 큰 차이점은 서버에 있는 거의 모든 데이터를 복사한다는 것이다. git clone 을 실행하면 프로젝트 히스토리를 전부 받아온다. 실제로 서버의 디스크가 망가져도 클라이언트 저장소 중에서 아무거나 하나 가져다가 복구하면 된다(서버에만 적용했던 설정은 복구하지 못하지만 모든 데이터는 복구된다 - [서버에 Git 설치하기](https://git-scm.com/book/ko/v2/ch00/_getting_git_on_a_server)에서 좀 더 자세히 다룬다).

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

>git clone https://github.com/libgit2/libgit2





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

아래과 같은 명령을 사용하여 저장소를 Clone 하면 `libgit2`이 아니라 다른 디렉토리 이름으로 Clone 할 수 있다.

> git clone https://github.com/libgit2/libgit2 mylibgit



디렉토리 이름이 mylibgit 이라는 것만 빼면 이 명령의 결과와 앞선 명령의 결과는 같다.

Git은 다양한 프로토콜을 지원한다. 이제까지는 https:// 프로토콜을 사용했지만 git:// 를 사용할 수도 있고 user@server:path/to/repo.git 처럼 SSH 프로토콜을 사용할 수도 있다. 자세한 내용은 [서버에 Git 설치하기](https://git-scm.com/book/ko/v2/ch00/_getting_git_on_a_server)에서 다루며 각 프로토콜의 장단점과 Git 저장소에 접근하는 방법을 설명한다.

## 수정하고 저장소에 저장하기

만질 수 있는 Git 저장소를 하나 만들었고 워킹 디렉토리에 Checkout도 했다. 이제는 파일을 수정하고 파일의 스냅샷을 커밋해 보자. 파일을 수정하다가 저장하고 싶으면 스냅샷을 커밋한다.

워킹 디렉토리의 모든 파일은 크게 Tracked(관리대상임)와 Untracked(관리대상이 아님)로 나눈다. Tracked 파일은 이미 스냅샷에 포함돼 있던 파일이다. Tracked 파일은 또 Unmodified(수정하지 않음)와 Modified(수정함) 그리고 Staged(커밋으로 저장소에 기록할) 상태 중 하나이다. 간단히 말하자면 Git이 알고 있는 파일이라는 것이다.

그리고 나머지 파일은 모두 Untracked 파일이다. Untracked 파일은 워킹 디렉토리에 있는 파일 중 스냅샷에도 Staging Area에도 포함되지 않은 파일이다. 처음 저장소를 Clone 하면 모든 파일은 Tracked이면서 Unmodified 상태이다. 파일을 Checkout 하고 나서 아무것도 수정하지 않았기 때문에 그렇다.

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

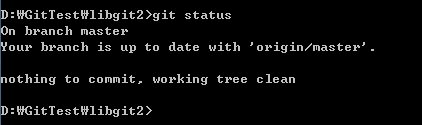


[그림 7] 파일의 라이프사이클

**파일의 상태 확인하기**

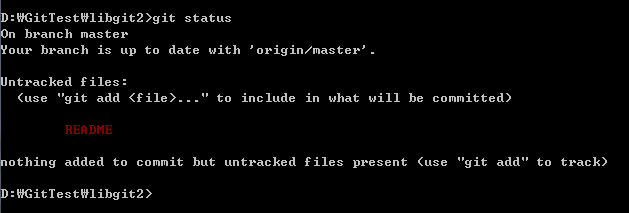
파일의 상태를 확인하려면 보통 git status 명령을 사용한다. Clone 한 후에 바로 이 명령을 실행하면 아래과 같은 메시지를 볼 수 있다.

>git status



위의 내용은 파일을 하나도 수정하지 않았다는 것을 말해준다. Tracked 파일은 하나도 수정되지 않았다는 의미다. Untracked 파일은 아직 없어서 목록에 나타나지 않는다. 그리고 현재 작업 중인 브랜치를 알려주며 서버의 같은 브랜치로부터 진행된 작업이 없는 것을 나타낸다. 기본 브랜치가 master이기 때문에 현재 브랜치 이름이 “master” 로 나온다. 브랜치 관련 내용은 차차 알아가자. [Git 브랜치](https://git-scm.com/book/ko/v2/ch00/ch03-git-branching) 에서 브랜치와 Refs에 대해 자세히 다룬다.

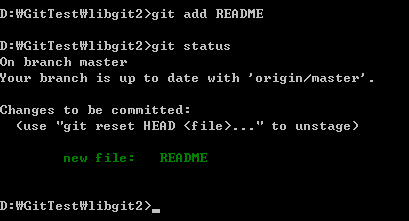
프로젝트에 README 파일을 만들어보자. README 파일은 새로 만든 파일이기 때문에 git status 를 실행하면 'Untracked files’에 들어 있다:



README 파일은 “Untracked files” 부분에 속해 있는데 이것은 README 파일이 Untracked 상태라는 것을 말한다. Git은 Untracked 파일을 아직 스냅샷(커밋)에 넣어지지 않은 파일이라고 본다. 파일이 Tracked 상태가 되기 전까지는 Git은 절대 그 파일을 커밋하지 않는다. 그래서 일하면서 생성하는 바이너리 파일 같은 것을 커밋하는 실수는 하지 않게 된다. README 파일을 추가해서 직접 Tracked 상태로 만들어 보자.

**파일을 새로 추적하기**

git add 명령으로 파일을 새로 추적할 수 있다. 아래 명령을 실행하면 Git은 README 파일을 추적한다.



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

“Changes to be committed” 에 들어 있는 파일은 Staged 상태라는 것을 의미한다. 커밋하면 git add 를 실행한 시점의 파일이 커밋되어 저장소 히스토리에 남는다. 앞에서 git init 명령을 실행한 후, git add (files) 명령을 실행했던 걸 기억할 것이다. 이 명령을 통해 디렉토리에 있는 파일을 추적하고 관리하도록 한다. git add 명령은 파일 또는 디렉토리의 경로를 아규먼트로 받는다. 디렉토리면 아래에 있는 모든 파일들까지 재귀적으로 추가한다.

**Modified 상태의 파일을 Stage 하기**

이미 Tracked 상태인 파일을 수정하는 법을 알아보자. CONTRIBUTING.md 라는 파일을 수정하고 나서 git status 명령을 다시 실행하면 결과는 아래와 같다.

$ git status

On branch master

Your branch is up-to-date with 'origin/master'.

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

new file: README

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

modified: CONTRIBUTING.md

이 CONTRIBUTING.md 파일은 “Changes not staged for commit” 에 있다. 이것은 수정한 파일이 Tracked 상태이지만 아직 Staged 상태는 아니라는 것이다. Staged 상태로 만들려면 git add 명령을 실행해야 한다. git add 명령은 파일을 새로 추적할 때도 사용하고 수정한 파일을 Staged 상태로 만들 때도 사용한다. Merge 할 때 충돌난 상태의 파일을 Resolve 상태로 만들때도 사용한다. add의 의미는 프로젝트에 파일을 추가한다기 보다는 다음 커밋에 추가한다고 받아들이는게 좋다. git add 명령을 실행하여 CONTRIBUTING.md 파일을 Staged 상태로 만들고 git status 명령으로 결과를 확인해보자.

$ git add CONTRIBUTING.md

$ git status

On branch master

Your branch is up-to-date with 'origin/master'.

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

new file: README

modified: CONTRIBUTING.md

두 파일 모두 Staged 상태이므로 다음 커밋에 포함된다. 하지만 아직 더 수정해야 한다는 것을 알게 되어 바로 커밋하지 못하는 상황이 되었다고 생각해보자. 이 상황에서 CONTRIBUTING.md 파일을 열고 수정한다. 이제 커밋할 준비가 다 됐다고 생각할 테지만, Git은 그렇지 않다. git status 명령으로 파일의 상태를 다시 확인해보자.

$ vim CONTRIBUTING.md

$ git status

On branch master

Your branch is up-to-date with 'origin/master'.

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

new file: README

modified: CONTRIBUTING.md

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

헉! CONTRIBUTING.md 가 Staged 상태이면서 동시에 Unstaged 상태로 나온다. 어떻게 이런 일이 가능할까? git add 명령을 실행하면 Git은 파일을 바로 Staged 상태로 만든다. 지금 이 시점에서 커밋을 하면 git commit 명령을 실행하는 시점의 버전이 커밋되는 것이 아니라 마지막으로 git add 명령을 실행했을 때의 버전이 커밋된다. 그러니까 git add 명령을 실행한 후에 또 파일을 수정하면 git add 명령을 다시 실행해서 최신 버전을 Staged 상태로 만들어야 한다.

$ git add CONTRIBUTING.md

$ git status

On branch master

Your branch is up-to-date with 'origin/master'.

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

new file: README

modified: CONTRIBUTING.md

**파일 상태를 짤막하게 확인하기**

git status 명령으로 확인할 수 있는 내용이 좀 많아 보일 수 있다. 사실 그렇다. 좀 더 간단하게 변경 내용을 보여주는 옵션이 있다. git status -s 또는 git status --short 처럼 옵션을 주면 현재 변경한 상태를 짤막하게 보여준다.

$ git status -s

M README

MM Rakefile

A lib/git.rb

M lib/simplegit.rb

?? LICENSE.txt

아직 추적하지 않는 새 파일 앞에는 ?? 표시가 붙는다. Staged 상태로 추가한 파일 중 새로 생성한 파일 앞에는 A 표시가, 수정한 파일 앞에는 M 표시가 붙는다. 위 명령의 결과에서 상태정보 컬럼에는 두 가지 정보를 보여준다. 왼쪽에는 Staging Area에서의 상태를, 오른쪽에는 Working Tree에서의 상태를 표시한다. README 파일 같은 경우 내용을 변경했지만 아직 Staged 상태로 추가하지는 않았다. lib/simplegit.rb 파일은 내용을 변경하고 Staged 상태로 추가까지 한 상태이다. 위 결과에서 차이점을 비교해보자. Rakefile 은 변경하고 Staged 상태로 추가한 후 또 내용을 변경해서 Staged 이면서 Unstaged 상태인 파일이다.

**파일 무시하기**

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

$ cat .gitignore

\*.[oa]

\*~

첫번째 라인은 확장자가 “.o” 나 “.a” 인 파일을 Git이 무시하라는 것이고 둘째 라인은 ~ 로 끝나는 모든 파일을 무시하라는 것이다. 보통 대부분의 텍스트 편집기에서 임시파일로 사용하는 파일 이름이기 때문이다. “.o” 와 “.a” 는 각각 빌드 시스템이 만들어내는 오브젝트와 아카이브 파일이고 ~ 로 끝나는 파일은 Emacs나 VI 같은 텍스트 편집기가 임시로 만들어내는 파일이다. 또 log, tmp, pid 같은 디렉토리나, 자동으로 생성하는 문서 같은 것들도 추가할 수 있다. .gitignore 파일은 보통 처음에 만들어 두는 것이 편리하다. 그래서 Git 저장소에 커밋하고 싶지 않은 파일을 실수로 커밋하는 일을 방지할 수 있다.

.gitignore 파일에 입력하는 패턴은 아래 규칙을 따른다.

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

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

아래는 .gitignore 파일의 예이다.

# 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

|  |  |
| --- | --- |
| Tip | GitHub은 다양한 프로젝트에서 자주 사용하는 .gitignore 예제를 관리하고 있다. 어떤 내용을 넣을지 막막하다면 <https://github.com/github/gitignore> 사이트에서 적당한 예제를 찾을 수 있다. |
| Note | .gitignore`를 사용하는 간단한 방식은 하나의 `.gitignore 파일을 최상위 디렉토리에 하나 두고 모든 하위 디렉토리에까지 적용시키는 방식이다. 물론 .gitignore 파일을 하나만 두는 것이 아니라 하위 디렉토리에도 추가로 둘 수도 있다. .gitignore 정책은 현재 .gitignore 파일이 위치한 디렉토리와 그 하위 디렉토리에 적용된다. (리눅스 커널 소스 저장소에는 .gitignore 파일이 206개나 있음)  다수의 .gitignore 파일을 두고 정책을 적용하는 부분은 이 책에서 다루는 범위를 벗어난다. 자세한 내용은 `man gitignore`에서 확인할 수 있다. |

**Staged와 Unstaged 상태의 변경 내용을 보기**

단순히 파일이 변경됐다는 사실이 아니라 어떤 내용이 변경됐는지 살펴보려면 git status 명령이 아니라 git diff 명령을 사용해야 한다. 보통 우리는 '수정했지만, 아직 Staged 파일이 아닌 것?'과 '어떤 파일이 Staged 상태인지?'가 궁금하기 때문에 git status 명령으로도 충분하다. 더 자세하게 볼 때는 git diff 명령을 사용하는데 Patch처럼 어떤 라인을 추가했고 삭제했는지가 궁금할 때 사용한다. git diff 는 나중에 더 자세히 다룬다.

README 파일을 수정해서 Staged 상태로 만들고 CONTRIBUTING.md 파일은 그냥 수정만 해둔다. 이 상태에서 git status 명령을 실행하면 아래와 같은 메시지를 볼 수 있다.

$ git status

On branch master

Your branch is up-to-date with 'origin/master'.

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

modified: README

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

modified: CONTRIBUTING.md

git diff 명령을 실행하면 수정했지만 아직 staged 상태가 아닌 파일을 비교해 볼 수 있다.

$ git diff

diff --git a/CONTRIBUTING.md b/CONTRIBUTING.md

index 8ebb991..643e24f 100644

--- a/CONTRIBUTING.md

+++ b/CONTRIBUTING.md

@@ -65,7 +65,8 @@ branch directly, things can get messy.

Please include a nice description of your changes when you submit your PR;

if we have to read the whole diff to figure out why you're contributing

in the first place, you're less likely to get feedback and have your change

-merged in.

+merged in. Also, split your changes into comprehensive chunks if your patch is

+longer than a dozen lines.

If you are starting to work on a particular area, feel free to submit a PR

that highlights your work in progress (and note in the PR title that it's

이 명령은 워킹 디렉토리에 있는 것과 Staging Area에 있는 것을 비교한다. 그래서 수정하고 아직 Stage 하지 않은 것을 보여준다.

만약 커밋하려고 Staging Area에 넣은 파일의 변경 부분을 보고 싶으면 git diff --staged 옵션을 사용한다. 이 명령은 저장소에 커밋한 것과 Staging Area에 있는 것을 비교한다.

$ git diff --staged

diff --git a/README b/README

new file mode 100644

index 0000000..03902a1

--- /dev/null

+++ b/README

@@ -0,0 +1 @@

+My Project

꼭 잊지 말아야 할 것이 있는데 git diff 명령은 마지막으로 커밋한 후에 수정한 것들 전부를 보여주지 않는다. git diff 는 Unstaged 상태인 것들만 보여준다. 수정한 파일을 모두 Staging Area에 넣었다면 git diff 명령은 아무것도 출력하지 않는다.

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

$ git add CONTRIBUTING.md

$ echo '# test line' >> CONTRIBUTING.md

$ git status

On branch master

Your branch is up-to-date with 'origin/master'.

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

modified: CONTRIBUTING.md

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

modified: CONTRIBUTING.md

git diff 명령으로 Unstaged 상태인 변경 부분을 확인할 수 있다.

$ git diff

diff --git a/CONTRIBUTING.md b/CONTRIBUTING.md

index 643e24f..87f08c8 100644

--- a/CONTRIBUTING.md

+++ b/CONTRIBUTING.md

@@ -119,3 +119,4 @@ at the

## Starter Projects

See our [projects list](https://github.com/libgit2/libgit2/blob/development/PROJECTS.md).

+# test line

Staged 상태인 파일은 git diff --cached 옵션으로 확인한다. --staged 와 --cached 는 같은 옵션이다.

$ git diff --cached

diff --git a/CONTRIBUTING.md b/CONTRIBUTING.md

index 8ebb991..643e24f 100644

--- a/CONTRIBUTING.md

+++ b/CONTRIBUTING.md

@@ -65,7 +65,8 @@ branch directly, things can get messy.

Please include a nice description of your changes when you submit your PR;

if we have to read the whole diff to figure out why you're contributing

in the first place, you're less likely to get feedback and have your change

-merged in.

+merged in. Also, split your changes into comprehensive chunks if your patch is

+longer than a dozen lines.

If you are starting to work on a particular area, feel free to submit a PR

that highlights your work in progress (and note in the PR title that it's

|  |  |
| --- | --- |
| Note | 외부 도구로 비교하기  이 책에서는 계속 git diff 명령으로 여기저기서 써 먹는다. 즐겨 쓰거나 결과를 아름답게 보여주는 Diff 도구가 있으면 사용할 수 있다. git diff 대신 git difftool 명령을 사용해서 emerge, vimdiff 같은 도구로 비교할 수 있다. 상용 제품도 사용할 수 있다. git difftool --tool-help 라는 명령은 사용가능한 도구를 보여준다. |

**변경사항 커밋하기**

수정한 것을 커밋하기 위해 Staging Area에 파일을 정리했다. Unstaged 상태의 파일은 커밋되지 않는다는 것을 기억해야 한다. Git은 생성하거나 수정하고 나서 git add 명령으로 추가하지 않은 파일은 커밋하지 않는다. 그 파일은 여전히 Modified 상태로 남아 있다. 커밋하기 전에 git status 명령으로 모든 것이 Staged 상태인지 확인할 수 있다. 그 후에 git commit 을 실행하여 커밋한다.

$ git commit

Git 설정에 지정된 편집기가 실행되고, 아래와 같은 텍스트가 자동으로 포함된다 (아래 예제는 Vim 편집기의 화면이다. 이 편집기는 쉘의 EDITOR 환경 변수에 등록된 편집기이고 보통은 Vim이나 Emacs을 사용한다. 또 [시작하기](https://git-scm.com/book/ko/v2/ch00/ch01-getting-started) 에서 설명했듯이 git config --global core.editor 명령으로 어떤 편집기를 사용할지 설정할 수 있다).

편집기는 아래와 같은 내용을 표시한다(아래 예제는 Vim 편집기).

# Please enter the commit message for your changes. Lines starting

# with '#' will be ignored, and an empty message aborts the commit.

# On branch master

# Your branch is up-to-date with 'origin/master'.

#

# Changes to be committed:

# new file: README

# modified: CONTRIBUTING.md

#

~

~

~

".git/COMMIT\_EDITMSG" 9L, 283C

자동으로 생성되는 커밋 메시지의 첫 라인은 비어 있고 둘째 라인부터 git status 명령의 결과가 채워진다. 커밋한 내용을 쉽게 기억할 수 있도록 이 메시지를 포함할 수도 있고 메시지를 전부 지우고 새로 작성할 수 있다 (정확히 뭘 수정했는지도 보여줄 수 있는데, git commit 에 -v 옵션을 추가하면 편집기에 diff 메시지도 추가된다). 내용을 저장하고 편집기를 종료하면 Git은 입력된 내용(#로 시작하는 내용을 제외한)으로 새 커밋을 하나 완성한다.

메시지를 인라인으로 첨부할 수도 있다. commit 명령을 실행할 때 아래와 같이 -m 옵션을 사용한다.

$ git commit -m "Story 182: Fix benchmarks for speed"

[master 463dc4f] Story 182: Fix benchmarks for speed

2 files changed, 2 insertions(+)

create mode 100644 README

이렇게 첫번째 커밋을 작성해보았다. commit 명령은 몇 가지 정보를 출력하는데 위 예제는 (master) 브랜치에 커밋했고 체크섬은 (463dc4f)이라고 알려준다. 그리고 수정한 파일이 몇 개이고 삭제됐거나 추가된 라인이 몇 라인인지 알려준다.

Git은 Staging Area에 속한 스냅샷을 커밋한다는 것을 기억해야 한다. 수정은 했지만, 아직 Staging Area에 넣지 않은 것은 다음에 커밋할 수 있다. 커밋할 때마다 프로젝트의 스냅샷을 기록하기 때문에 나중에 스냅샷끼리 비교하거나 예전 스냅샷으로 되돌릴 수 있다.

**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.

$ git status

On branch master

Your branch is up-to-date with 'origin/master'.

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

modified: CONTRIBUTING.md

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

$ git commit -a -m 'added new benchmarks'

[master 83e38c7] added new benchmarks

1 file changed, 5 insertions(+), 0 deletions(-)

이 예제에서는 커밋하기 전에 git add 명령으로 CONTRIBUTING.md 파일을 추가하지 않았다는 점을 눈여겨보자. -a 옵션을 사용하면 모든 파일이 자동으로 추가된다. 편리한 옵션이긴 하지만 주의 깊게 사용해야 한다. 생각 없이 이 옵션을 사용하다 보면 추가하지 말아야 할 변경사항도 추가될 수 있기 때문이다.

**파일 삭제하기**

Git에서 파일을 제거하려면 git rm 명령으로 Tracked 상태의 파일을 삭제한 후에(정확하게는 Staging Area에서 삭제하는 것) 커밋해야 한다. 이 명령은 워킹 디렉토리에 있는 파일도 삭제하기 때문에 실제로 파일도 지워진다.

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

$ rm PROJECTS.md

$ git status

On branch master

Your branch is up-to-date with 'origin/master'.

Changes not staged for commit:

(use "git add/rm <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

deleted: PROJECTS.md

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

그리고 git rm 명령을 실행하면 삭제한 파일은 Staged 상태가 된다.

$ git rm PROJECTS.md

rm 'PROJECTS.md'

$ git status

On branch master

Your branch is up-to-date with 'origin/master'.

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

deleted: PROJECTS.md

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

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

$ git rm --cached README

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

$ git rm log/\\*.log

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

$ git rm \\*~

이 명령은 ~ 로 끝나는 파일을 모두 삭제한다.

**파일 이름 변경하기**

Git은 다른 VCS 시스템과는 달리 파일 이름의 변경이나 파일의 이동을 명시적으로 관리하지 않는다. 다시 말해서 파일 이름이 변경됐다는 별도의 정보를 저장하지 않는다. Git은 똑똑해서 굳이 파일 이름이 변경되었다는 것을 추적하지 않아도 아는 방법이 있다. 파일의 이름이 변경된 것을 Git이 어떻게 알아내는지 살펴보자.

이렇게 말하고 Git에 mv 명령이 있는 게 좀 이상하겠지만, 아래와 같이 파일 이름을 변경할 수 있다.

$ git mv file\_from file\_to

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

$ git mv README.md README

$ git status

On branch master

Your branch is up-to-date with 'origin/master'.

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

renamed: README.md -> README

사실 git mv 명령은 아래 명령어를 수행한 것과 완전 똑같다.

$ mv README.md README

$ git rm README.md

$ git add README

git mv 명령은 일종의 단축 명령어이다. 이 명령으로 파일 이름을 바꿔도 되고 mv 명령으로 파일 이름을 직접 바꿔도 된다. 단지 git mv 명령은 편리하게 명령을 세 번 실행해주는 것 뿐이다. 어떤 도구로 이름을 바꿔도 상관없다. 중요한 것은 이름을 변경하고 나서 꼭 rm/add 명령을 실행해야 한다는 것 뿐이다.

## 커밋 히스토리 조회하기

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

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

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

이 프로젝트 디렉토리에서 git log 명령을 실행하면 아래와 같이 출력된다.

$ git log

commit ca82a6dff817ec66f44342007202690a93763949

Author: Scott Chacon <schacon@gee-mail.com>

Date: Mon Mar 17 21:52:11 2008 -0700

changed the version number

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7

Author: Scott Chacon <schacon@gee-mail.com>

Date: Sat Mar 15 16:40:33 2008 -0700

removed unnecessary test

commit a11bef06a3f659402fe7563abf99ad00de2209e6

Author: Scott Chacon <schacon@gee-mail.com>

Date: Sat Mar 15 10:31:28 2008 -0700

first commit

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

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

여러 옵션 중 -p, --patch 는 굉장히 유용한 옵션이다. -p 는 각 커밋의 diff 결과를 보여준다. 다른 유용한 옵션으로

가 있는데 최근 두 개의 결과만 보여주는 옵션이다:

$ git log -p -2

commit ca82a6dff817ec66f44342007202690a93763949

Author: Scott Chacon <schacon@gee-mail.com>

Date: Mon Mar 17 21:52:11 2008 -0700

changed the version number

diff --git a/Rakefile b/Rakefile

index a874b73..8f94139 100644

--- a/Rakefile

+++ b/Rakefile

@@ -5,7 +5,7 @@ require 'rake/gempackagetask'

spec = Gem::Specification.new do |s|

s.platform = Gem::Platform::RUBY

s.name = "simplegit"

- s.version = "0.1.0"

+ s.version = "0.1.1"

s.author = "Scott Chacon"

s.email = "schacon@gee-mail.com"

s.summary = "A simple gem for using Git in Ruby code."

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7

Author: Scott Chacon <schacon@gee-mail.com>

Date: Sat Mar 15 16:40:33 2008 -0700

removed unnecessary test

diff --git a/lib/simplegit.rb b/lib/simplegit.rb

index a0a60ae..47c6340 100644

--- a/lib/simplegit.rb

+++ b/lib/simplegit.rb

@@ -18,8 +18,3 @@ class SimpleGit

end

end

-

-if $0 == \_\_FILE\_\_

- git = SimpleGit.new

- puts git.show

-end

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

$ git log --stat

commit ca82a6dff817ec66f44342007202690a93763949

Author: Scott Chacon <schacon@gee-mail.com>

Date: Mon Mar 17 21:52:11 2008 -0700

changed the version number

Rakefile | 2 +-

1 file changed, 1 insertion(+), 1 deletion(-)

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7

Author: Scott Chacon <schacon@gee-mail.com>

Date: Sat Mar 15 16:40:33 2008 -0700

removed unnecessary test

lib/simplegit.rb | 5 -----

1 file changed, 5 deletions(-)

commit a11bef06a3f659402fe7563abf99ad00de2209e6

Author: Scott Chacon <schacon@gee-mail.com>

Date: Sat Mar 15 10:31:28 2008 -0700

first commit

README | 6 ++++++

Rakefile | 23 +++++++++++++++++++++++

lib/simplegit.rb | 25 +++++++++++++++++++++++++

3 files changed, 54 insertions(+)

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

다른 또 유용한 옵션은 --pretty 옵션이다. 이 옵션을 통해 히스토리 내용을 보여줄 때 기본 형식 이외에 여러 가지 중에 하나를 선택할 수 있다. 몇개 선택할 수 있는 옵션의 값이 있다. oneline 옵션은 각 커밋을 한 라인으로 보여준다. 이 옵션은 많은 커밋을 한 번에 조회할 때 유용하다. 추가로 short, full, fuller 옵션도 있는데 이것은 정보를 조금씩 가감해서 보여준다.

$ git log --pretty=oneline

ca82a6dff817ec66f44342007202690a93763949 changed the version number

085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7 removed unnecessary test

a11bef06a3f659402fe7563abf99ad00de2209e6 first commit

가장 재밌는 옵션은 format 옵션이다. 나만의 포맷으로 결과를 출력하고 싶을 때 사용한다. 특히 결과를 다른 프로그램으로 파싱하고자 할 때 유용하다. 이 옵션을 사용하면 포맷을 정확하게 일치시킬 수 있기 때문에 Git을 새 버전으로 바꿔도 결과 포맷이 바뀌지 않는다.

$ git log --pretty=format:"%h - %an, %ar : %s"

ca82a6d - Scott Chacon, 6 years ago : changed the version number

085bb3b - Scott Chacon, 6 years ago : removed unnecessary test

a11bef0 - Scott Chacon, 6 years ago : first commit

[git log --pretty=format 에 쓸 몇가지 유용한 옵션`](https://git-scm.com/book/ko/v2/ch00/pretty_format) 포맷에서 사용하는 유용한 옵션.

| Table 1. git log --pretty=format 에 쓸 몇가지 유용한 옵션` | |
| --- | --- |
| **옵션** | **설명** |
| %H | 커밋 해시 |
| %h | 짧은 길이 커밋 해시 |
| %T | 트리 해시 |
| %t | 짧은 길이 트리 해시 |
| %P | 부모 해시 |
| %p | 짧은 길이 부모 해시 |
| %an | 저자 이름 |
| %ae | 저자 메일 |
| %ad | 저자 시각 (형식은 –-date=옵션 참고) |
| %ar | 저자 상대적 시각 |
| %cn | 커미터 이름 |
| %ce | 커미터 메일 |
| %cd | 커미터 시각 |
| %cr | 커미터 상대적 시각 |
| %s | 요약 |

저자(Author) 와 커미터(Committer) 를 구분하는 것이 조금 이상해 보일 수 있다. 저자는 원래 작업을 수행한 원작자이고 커밋터는 마지막으로 이 작업을 적용한(저장소에 포함시킨) 사람이다. 만약 당신이 어떤 프로젝트에 패치를 보냈고 그 프로젝트의 담당자가 패치를 적용했다면 두 명의 정보를 모두 알 필요가 있다. 그래서 이 경우 당신이 저자고 그 담당자가 커미터다. [분산 환경에서의 Git](https://git-scm.com/book/ko/v2/ch00/ch05-distributed-git) 에서 이 주제에 대해 자세히 다룰 것이다.

oneline 옵션과 format 옵션은 --graph 옵션과 함께 사용할 때 더 빛난다. 이 명령은 브랜치와 머지 히스토리를 보여주는 아스키 그래프를 출력한다.

$ 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

다음 장에서 살펴볼 브랜치나 Merge 결과의 히스토리를 이런 식으로 살펴보면 훨씬 흥미롭다.

git log 명령의 기본적인 옵션과 출력물의 형식에 관련된 옵션을 살펴보았다. git log 명령은 앞서 살펴본 것보다 더 많은 옵션을 지원한다. [git log 주요 옵션](https://git-scm.com/book/ko/v2/ch00/log_options) 는 지금 설명한 것과 함께 유용하게 사용할 수 있는 옵션이다. 각 옵션으로 어떻게 log 명령을 제어할 수 있는지 보여준다.

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

**조회 제한 조건**

출력 형식과 관련된 옵션을 살펴봤지만 git log 명령은 조회 범위를 제한하는 옵션들도 있다. 히스토리 전부가 아니라 부분만 조회한다. 이미 최근 두 개만 조회하는 -2 옵션은 살펴봤다. 실제 사용법은 `-<n>`이고 n은 최근 n개의 커밋을 의미한다. 사실 이 옵션을 자주 쓰진 않는다. Git은 기본적으로 출력을 pager류의 프로그램을 거쳐서 내보내므로 한 번에 한 페이지씩 보여준다.

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

$ git log --since=2.weeks

이 옵션은 다양한 형식을 지원한다. "2008-01-15" 같이 정확한 날짜도 사용할 수 있고 "2 years 1 day 3 minutes ago" 같이 상대적인 기간을 사용할 수도 있다.

또 다른 기준도 있다. --author 옵션으로 저자를 지정하여 검색할 수도 있고 --grep 옵션으로 커밋 메시지에서 키워드를 검색할 수도 있다

|  |  |
| --- | --- |
| Note | --author`와 `--grep 옵션을 함께 사용하여 모두 만족하는 커밋을 찾으려면 --all-match 옵션도 반드시 함께 사용해야 한다. |

진짜 유용한 옵션으로 -S 가 있는데 이 옵션은 코드에서 추가되거나 제거된 내용 중에 특정 텍스트가 포함되어 있는지를 검색한다. 예를 들어 어떤 함수가 추가되거나 제거된 커밋만을 찾아보려면 아래와 같은 명령을 사용한다.

$ git log -S function\_name

마지막으로 파일 경로로 검색하는 옵션이 있는데 이것도 정말 유용하다. 디렉토리나 파일 이름을 사용하여 그 파일이 변경된 log의 결과를 검색할 수 있다. 이 옵션은 -- 와 함께 경로 이름을 사용하는데 명령어 끝 부분에 쓴다(역주 - git log -- path1 path2).

[git log 조회 범위를 제한하는 옵션](https://git-scm.com/book/ko/v2/ch00/limit_options) 은 조회 범위를 제한하는 옵션들이다.

| Table 3. git log 조회 범위를 제한하는 옵션 | |
| --- | --- |
| **옵션** | **설명** |
| -(n) | 최근 n 개의 커밋만 조회한다. |
| --since, --after | 명시한 날짜 이후의 커밋만 검색한다. |
| --until, --before | 명시한 날짜 이전의 커밋만 조회한다. |
| --author | 입력한 저자의 커밋만 보여준다. |
| --committer | 입력한 커미터의 커밋만 보여준다. |
| --grep | 커밋 메시지 안의 텍스트를 검색한다. |
| -S | 커밋 변경(추가/삭제) 내용 안의 텍스트를 검색한다. |

이제 살펴볼 예제는 Merge 커밋을 제외한 순수한 커밋을 확인해보는 명령이다. Junio Hamano가 2008년 10월에 Git 소스코드 저장소에서 테스트 파일을 수정한 커밋들이다.

$ git log --pretty="%h - %s" --author=gitster --since="2008-10-01" \

--before="2008-11-01" --no-merges -- t/

5610e3b - Fix testcase failure when extended attributes are in use

acd3b9e - Enhance hold\_lock\_file\_for\_{update,append}() API

f563754 - demonstrate breakage of detached checkout with symbolic link HEAD

d1a43f2 - reset --hard/read-tree --reset -u: remove unmerged new paths

51a94af - Fix "checkout --track -b newbranch" on detached HEAD

b0ad11e - pull: allow "git pull origin $something:$current\_branch" into an unborn branch

총 4만여 개의 커밋 히스토리에서 이 명령의 검색 조건에 만족하는 것은 단 6개였다.

|  |  |
| --- | --- |
| Tip | 머지 커밋 표시하지 않기  저장소를 사용하는 워크플로우에 따라 머지 커밋이 차지하는 비중이 클 수도 있다. --no-merges 옵션을 사용하면 검색 결과에서 머지 커밋을 표시하지 않도록 할 수 있다. |

## 되돌리기

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

종종 완료한 커밋을 수정해야 할 때가 있다. 너무 일찍 커밋했거나 어떤 파일을 빼먹었을 때 그리고 커밋 메시지를 잘못 적었을 때 한다. 다시 커밋하고 싶으면 파일 수정 작업을 하고 Staging Area에 추가한 다음 --amend 옵션을 사용하여 커밋을 재작성 할 수 있다.

$ git commit --amend

이 명령은 Staging Area를 사용하여 커밋한다. 만약 마지막으로 커밋하고 나서 수정한 것이 없다면(커밋하자마자 바로 이 명령을 실행하는 경우) 조금 전에 한 커밋과 모든 것이 같다. 이때는 커밋 메시지만 수정한다.

편집기가 실행되면 이전 커밋 메시지가 자동으로 포함된다. 메시지를 수정하지 않고 그대로 커밋해도 기존의 커밋을 덮어쓴다.

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

$ git commit -m 'initial commit'

$ git add forgotten\_file

$ git commit --amend

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

|  |  |
| --- | --- |
| Note | 이렇게 --amend 옵션으로 커밋을 고치는 작업은, 추가로 작업한 일이 작다고 하더라도 이전의 커밋을 완전히 새로 고쳐서 새 커밋으로 변경하는 것을 의미한다. 이전의 커밋은 일어나지 않은 일이 되는 것이고 당연히 히스토리에도 남지 않는다.  --amend 옵션으로 커밋을 고치는 작업이 주는 장점은 마지막 커밋 작업에서 아주 살짝 뭔가 빠뜨린 것을 넣거나 변경하는 것을 새 커밋으로 분리하지 않고 하나의 커밋에서 처리하는 것이다. “앗차, 빠진 파일 넣었음”, “이전 커밋에서 오타 살짝 고침” 등의 커밋을 만들지 않겠다는 말이다. |

**파일 상태를 Unstage로 변경하기**

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

$ git add \*

$ git status

On branch master

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

renamed: README.md -> README

modified: CONTRIBUTING.md

Changes to be commited 밑에 git reset HEAD <file>... 메시지가 보인다. 이 명령으로 Unstaged 상태로 변경할 수 있다. CONTRIBUTING.md 파일을 Unstaged 상태로 변경해보자.

$ 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

명령어가 낮설게 느껴질 수도 있지만 잘 동작한다. CONTRIBUTING.md 파일은 Unstaged 상태가 됐다.

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

지금까지 살펴본 내용이 git reset 명령에 대해 알아야 할 대부분의 내용이다. reset 명령이 정확히는 어떻게 동작하는지, 어떻게 전문적으로 활용하는지는 [Reset 명확히 알고 가기](https://git-scm.com/book/ko/v2/ch00/_git_reset) 부분에서 자세히 살펴보기로 한다.

**Modified 파일 되돌리기**

어떻게 해야 CONTRIBUTING.md 파일을 수정하고 나서 다시 되돌릴 수 있을까? 그러니까 최근 커밋된 버전으로(아니면 처음 Clone 했을 때처럼 워킹 디렉토리에 처음 Checkout 한 그 내용으로) 되돌리는 방법이 무얼까? git status 명령이 친절하게 알려준다. 바로 위에 있는 예제에서 Unstaged 부분을 보자.

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

modified: CONTRIBUTING.md

위의 메시지는 수정한 파일을 되돌리는 방법을 꽤 정확하게 알려준다. 알려주는 대로 한 번 해보자.

$ git checkout -- CONTRIBUTING.md

$ git status

On branch master

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

renamed: README.md -> README

정상적으로 복원된 것을 알 수 있다.

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

변경한 내용을 쉽게 버릴수는 없고 하지만 당장은 되돌려야만 하는 상황이라면 Stash와 Branch를 사용하자. [Git 브랜치](https://git-scm.com/book/ko/v2/ch00/ch03-git-branching) 에서 다루는 이 방법들이 훨씬 낫다.

Git으로 *커밋* 한 모든 것은 언제나 복구할 수 있다. 삭제한 브랜치에 있었던 것도, --amend 옵션으로 다시 커밋한 것도 복구할 수 있다(자세한 것은 [데이터 복구](https://git-scm.com/book/ko/v2/ch00/_data_recovery) 에서 다룬다). 하지만 커밋하지 않고 잃어버린 것은 절대로 되돌릴 수 없다.

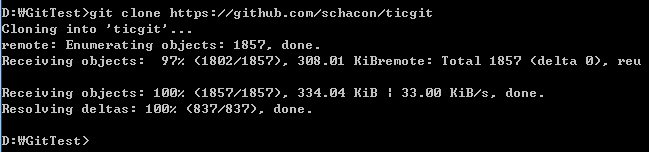
## 리모트 저장소

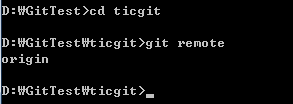
리모트 저장소를 관리할 줄 알아야 다른 사람과 함께 일할 수 있다. 리모트 저장소는 인터넷이나 네트워크 어딘가에 있는 저장소를 말한다. 저장소는 여러 개가 있을 수 있는데 어떤 저장소는 읽고 쓰기 모두 할 수 있고 어떤 저장소는 읽기만 가능할 수 있다. 간단히 말해서 다른 사람들과 함께 일한다는 것은 리모트 저장소를 관리하면서 데이터를 거기에 Push 하고 Pull 하는 것이다. 리모트 저장소를 관리한다는 것은 저장소를 추가, 삭제하는 것뿐만 아니라 브랜치를 관리하고 추적할지 말지 등을 관리하는 것을 말한다. 이번에는 리모트 저장소를 관리하는 방법에 대해 설명한다.

|  |  |
| --- | --- |
| Note | 원격 저장소라 하더라도 로컬 시스템에 위치할 수도 있다.  “remote” 저장소라고 이름이 붙어있어도 이 원격 저장소가 사실 같은 로컬 시스템에 존재할 수도 있다. 여기서 “remote” 라는 이름은 반드시 저장소가 네트워크나 인터넷을 통해 어딘가 멀리 떨어져 있어야만 한다는 것을 의미하지 않는다. 물론 일반적인 원격 저장소와 마찬가지로 Push, Pull 등의 기능은 동일하게 사용한다. |

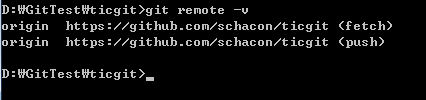
**리모트 저장소 확인하기**

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





-v 옵션을 주어 단축이름과 URL을 함께 볼 수 있다.



리모트 저장소가 여러 개 있다면 이 명령은 등록된 전부를 보여준다. 여러 사람과 함께 작업하는 리모트 저장소가 여러개라면 아래와 같은 결과를 얻을 수도 있다.

$ cd grit

$ git remote -v

bakkdoor https://github.com/bakkdoor/grit (fetch)

bakkdoor https://github.com/bakkdoor/grit (push)

cho45 https://github.com/cho45/grit (fetch)

cho45 https://github.com/cho45/grit (push)

defunkt https://github.com/defunkt/grit (fetch)

defunkt https://github.com/defunkt/grit (push)

koke git://github.com/koke/grit.git (fetch)

koke git://github.com/koke/grit.git (push)

origin git@github.com:mojombo/grit.git (fetch)

origin git@github.com:mojombo/grit.git (push)

이렇게 리모트 저장소가 여러 개 등록되어 있으면 다른 사람이 기여한 내용(Contributions)을 쉽게 가져올 수 있다. 어떤 저장소에는 Push 권한까지 제공하기도 하지만 일단 이 화면에서 Push 가능 권한까지는 확인할 수 없다.

리모트 저장소와 데이터를 주고받는데 사용하는 다양한 프로토콜에 대해서는 [서버에 Git 설치하기](https://git-scm.com/book/ko/v2/ch00/_getting_git_on_a_server) 에서 자세히 살펴보기로 한다.

**리모트 저장소 추가하기**

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

$ git remote

origin

$ git remote add pb https://github.com/paulboone/ticgit

$ git remote -v

origin https://github.com/schacon/ticgit (fetch)

origin https://github.com/schacon/ticgit (push)

pb https://github.com/paulboone/ticgit (fetch)

pb https://github.com/paulboone/ticgit (push)

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

$ git fetch pb

remote: Counting objects: 43, done.

remote: Compressing objects: 100% (36/36), done.

remote: Total 43 (delta 10), reused 31 (delta 5)

Unpacking objects: 100% (43/43), done.

From https://github.com/paulboone/ticgit

\* [new branch] master -> pb/master

\* [new branch] ticgit -> pb/ticgit

로컬에서 pb/master 가 Paul의 master 브랜치이다. 이 브랜치를 로컬 브랜치중 하나에 Merge 하거나 Checkout 해서 브랜치 내용을 자세히 확인할 수 있다. (브랜치를 어떻게 사용하는지는 [Git 브랜치](https://git-scm.com/book/ko/v2/ch00/ch03-git-branching) 에서 자세히 살펴본다)

**리모트 저장소를 Pull 하거나 Fetch 하기**

앞서 설명했듯이 리모트 저장소에서 데이터를 가져오려면 간단히 아래와 같이 실행한다.

$ git fetch <remote>

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

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

그냥 쉽게 git pull 명령으로 리모트 저장소 브랜치에서 데이터를 가져올 뿐만 아니라 자동으로 로컬 브랜치와 Merge 시킬 수 있다(다음 섹션과 [Git 브랜치](https://git-scm.com/book/ko/v2/ch00/ch03-git-branching) 에서 좀더 자세히 살펴본다). 먼저 git clone 명령은 자동으로 로컬의 master 브랜치가 리모트 저장소의 master 브랜치를 추적하도록 한다(물론 리모트 저장소에 master 브랜치가 있다는 가정에서). 그리고 git pull 명령은 Clone 한 서버에서 데이터를 가져오고 그 데이터를 자동으로 현재 작업하는 코드와 Merge 시킨다.

**리모트 저장소에 Push 하기**

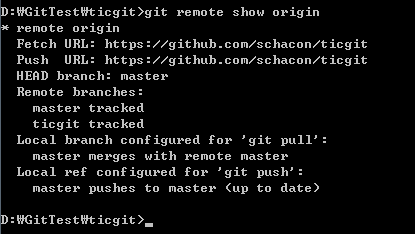
프로젝트를 공유하고 싶을 때 Upstream 저장소에 Push 할 수 있다. 이 명령은 git push <리모트 저장소 이름> <브랜치 이름>`으로 단순하다.(((git commands, push))) master 브랜치를 `origin 서버에 Push 하려면(다시 말하지만 Clone 하면 보통 자동으로 origin 이름이 생성된다) 아래와 같이 서버에 Push 한다.

$ git push origin master

이 명령은 Clone 한 리모트 저장소에 쓰기 권한이 있고, Clone 하고 난 이후 아무도 Upstream 저장소에 Push 하지 않았을 때만 사용할 수 있다. 다시 말해서 Clone 한 사람이 여러 명 있을 때, 다른 사람이 Push 한 후에 Push 하려고 하면 Push 할 수 없다. 먼저 다른 사람이 작업한 것을 가져와서 Merge 한 후에 Push 할 수 있다. [Git 브랜치](https://git-scm.com/book/ko/v2/ch00/ch03-git-branching) 에서 서버에 Push 하는 방법에 대해 자세히 설명할 것이다.

**리모트 저장소 살펴보기**

git remote show <리모트 저장소 이름> 명령으로 리모트 저장소의 구체적인 정보를 확인할 수 있다. origin 같은 단축이름으로 이 명령을 실행하면 아래와 같은 정보를 볼 수 있다.



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

좀 더 Git을 열심히 사용하다 보면 git remote show 명령으로 더 많은 정보를 보는 날이 온다. 여러분도 언젠가는 아래와 같은 메시지(역주 - 다수의 브랜치를 사용하는 메시지)를 볼 날이 올 것이다.

$ 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)

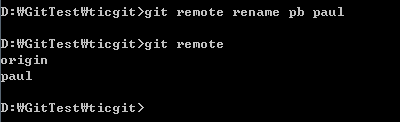
markdown-strip pushes to markdown-strip (up to date)

master pushes to master (up to date)

브랜치명을 생략하고 git push 명령을 실행할 때 어떤 브랜치가 어떤 브랜치로 Push 되는지 보여준다. 또 아직 로컬로 가져오지 않은 리모트 저장소의 브랜치는 어떤 것들이 있는지, 서버에서는 삭제됐지만 아직 가지고 있는 브랜치는 어떤 것인지, git pull 명령을 실행했을 때 자동으로 Merge 할 브랜치는 어떤 것이 있는지 보여준다.

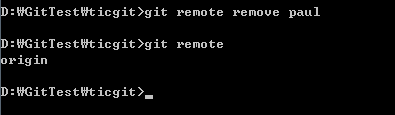
**리모트 저장소 이름을 바꾸거나 리모트 저장소를 삭제하기**

git remote rename 명령으로 리모트 저장소의 이름을 변경할 수 있다. 예를 들어 pb 를 paul 로 변경하려면 git remote rename 명령을 사용한다.



로컬에서 관리하던 리모트 저장소의 브랜치 이름도 바뀐다는 점을 생각해두자. 여태까지 pb/master 로 리모트 저장소 브랜치를 사용했으면 이제는 paul/master 라고 사용해야 한다.

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



위와 같은 방법으로 리모트 저장소를 삭제하면 해당 리모트 저장소에 관련된 추적 브랜치 정보나 모든 설정 내용도 함께 삭제된다.

## 태그

다른 VCS처럼 Git도 태그를 지원한다. 사람들은 보통 릴리즈할 때 사용한다(v1.0, 등등). 이번에는 태그를 조회하고 생성하는 법과 태그의 종류를 설명한다.

**태그 조회하기**

우선 git tag 명령으로 (-l, `--list`는 옵션) 이미 만들어진 태그가 있는지 확인할 수 있다.

$ git tag

v0.1

v1.3

이 명령은 알파벳 순서로 태그를 보여준다. 사실 순서는 별로 중요한 게 아니다.

검색 패턴을 사용하여 태그를 검색할 수 있다. Git 소스 저장소는 500여 개의 태그가 있다. 만약 1.8.5 버전의 태그들만 검색하고 싶으면 아래와 같이 실행한다.

$ git tag -l "v1.8.5\*"

v1.8.5

v1.8.5-rc0

v1.8.5-rc1

v1.8.5-rc2

v1.8.5-rc3

v1.8.5.1

v1.8.5.2

v1.8.5.3

v1.8.5.4

v1.8.5.5

|  |  |
| --- | --- |
| Note | 와일드카드를 사용하여 Tag 리스트를 확인하려면 -l, --list 옵션을 지정  단순히 모든 Tag 목록을 확인하기 위해 git tag 명령을 실행했을 때 -l 또는 --list 옵션이 적용된 것과 동일한 결과가 출력된다.  하지만 와일드카드를 사용하여 태그 목록을 검색하는 경우에는 반드시 -l 또는 --list 옵션을 같이 써 줘야 원하는 결과를 얻을 수 있다. |

**태그 붙이기**

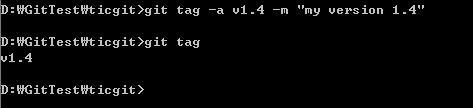
Git의 태그는 Lightweight 태그와 Annotated 태그로 두 종류가 있다.

Lightweight 태그는 브랜치와 비슷한데 브랜치처럼 가리키는 지점을 최신 커밋으로 이동시키지 않는다. 단순히 특정 커밋에 대한 포인터일 뿐이다.

한편 Annotated 태그는 Git 데이터베이스에 태그를 만든 사람의 이름, 이메일과 태그를 만든 날짜, 그리고 태그 메시지도 저장한다. GPG(GNU Privacy Guard)로 서명할 수도 있다. 일반적으로 Annotated 태그를 만들어 이 모든 정보를 사용할 수 있도록 하는 것이 좋다. 하지만 임시로 생성하는 태그거나 이러한 정보를 유지할 필요가 없는 경우에는 Lightweight 태그를 사용할 수도 있다.

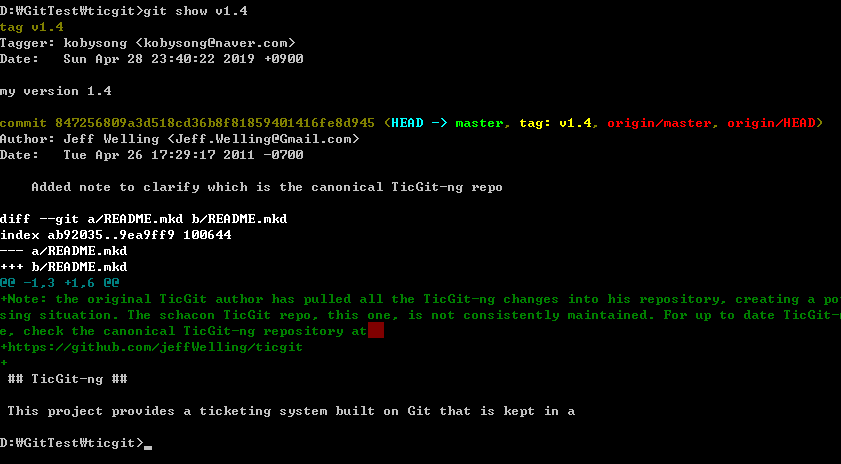
**Annotated 태그**

Annotated 태그를 만드는 방법은 간단하다. tag 명령을 실행할 때 -a 옵션을 추가한다.



-m 옵션으로 태그를 저장할 때 메시지를 함께 저장할 수 있다. 명령을 실행할 때 메시지를 입력하지 않으면 Git은 편집기를 실행시킨다.

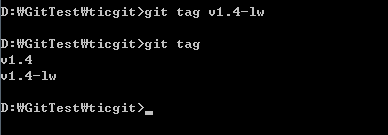
git show 명령으로 태그 정보와 커밋 정보를 모두 확인할 수 있다.



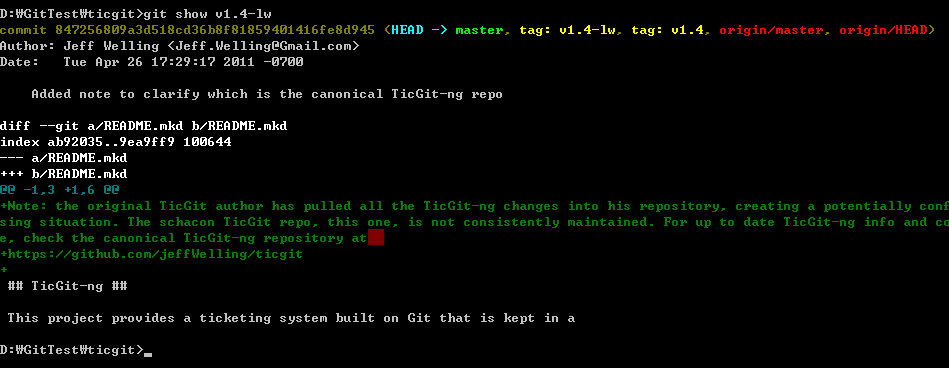
커밋 정보를 보여주기 전에 먼저 태그를 만든 사람이 누구인지, 언제 태그를 만들었는지, 그리고 태그 메시지가 무엇인지 보여준다.

**Lightweight 태그**

Lightweight 태그는 기본적으로 파일에 커밋 체크섬을 저장하는 것뿐이다. 다른 정보는 저장하지 않는다. Lightweight 태그를 만들 때는 -a, -s, -m 옵션을 사용하지 않는다. 이름만 달아줄 뿐이다.



이 태그에 git show 를 실행하면 별도의 태그 정보를 확인할 수 없다. 이 명령은 단순히 커밋 정보만을 보여준다.



**나중에 태그하기**

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

$ git log --pretty=oneline

15027957951b64cf874c3557a0f3547bd83b3ff6 Merge branch 'experiment'

a6b4c97498bd301d84096da251c98a07c7723e65 beginning write support

0d52aaab4479697da7686c15f77a3d64d9165190 one more thing

6d52a271eda8725415634dd79daabbc4d9b6008e Merge branch 'experiment'

0b7434d86859cc7b8c3d5e1dddfed66ff742fcbc added a commit function

4682c3261057305bdd616e23b64b0857d832627b added a todo file

166ae0c4d3f420721acbb115cc33848dfcc2121a started write support

9fceb02d0ae598e95dc970b74767f19372d61af8 updated rakefile

964f16d36dfccde844893cac5b347e7b3d44abbc commit the todo

8a5cbc430f1a9c3d00faaeffd07798508422908a updated readme

“updated rakefile” 커밋을 v1.2로 태그하지 못했다고 해도 나중에 태그를 붙일 수 있다. 특정 커밋에 태그하기 위해서 명령의 끝에 커밋 체크섬을 명시한다(긴 체크섬을 전부 사용할 필요는 없다).

$ git tag -a v1.2 9fceb02

이제 아래와 같이 만든 태그를 확인한다.

$ git tag

v0.1

v1.2

v1.3

v1.4

v1.4-lw

v1.5

$ git show v1.2

tag v1.2

Tagger: Scott Chacon <schacon@gee-mail.com>

Date: Mon Feb 9 15:32:16 2009 -0800

version 1.2

commit 9fceb02d0ae598e95dc970b74767f19372d61af8

Author: Magnus Chacon <mchacon@gee-mail.com>

Date: Sun Apr 27 20:43:35 2008 -0700

updated rakefile

...

**태그 공유하기**

git push 명령은 자동으로 리모트 서버에 태그를 전송하지 않는다. 태그를 만들었으면 서버에 별도로 Push 해야 한다. 브랜치를 공유하는 것과 같은 방법으로 할 수 있다. `git push origin <태그 이름>`을 실행한다.

$ git push origin v1.5

Counting objects: 14, done.

Delta compression using up to 8 threads.

Compressing objects: 100% (12/12), done.

Writing objects: 100% (14/14), 2.05 KiB | 0 bytes/s, done.

Total 14 (delta 3), reused 0 (delta 0)

To git@github.com:schacon/simplegit.git

\* [new tag] v1.5 -> v1.5

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

$ git push origin --tags

Counting objects: 1, done.

Writing objects: 100% (1/1), 160 bytes | 0 bytes/s, done.

Total 1 (delta 0), reused 0 (delta 0)

To git@github.com:schacon/simplegit.git

\* [new tag] v1.4 -> v1.4

\* [new tag] v1.4-lw -> v1.4-lw

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

**태그를 Checkout 하기**

예를 들어 태그가 특정 버전을 가리키고 있고, 특정 버전의 파일을 체크아웃 해서 확인하고 싶다면 다음과 같이 실행한다. 단 태그를 체크아웃하면(브랜치를 체크아웃 하는 것이 아니라면) “detached HEAD”(떨어져나온 HEAD) 상태가 되며 일부 Git 관련 작업이 브랜치에서 작업하는 것과 다르게 동작할 수 있다.

$ git checkout 2.0.0

Note: checking out '2.0.0'.

You are in 'detached HEAD' state. You can look around, make experimental

changes and commit them, and you can discard any commits you make in this

state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may

do so (now or later) by using -b with the checkout command again. Example:

git checkout -b <new-branch>

HEAD is now at 99ada87... Merge pull request #89 from schacon/appendix-final

$ git checkout 2.0-beta-0.1

Previous HEAD position was 99ada87... Merge pull request #89 from schacon/appendix-final

HEAD is now at df3f601... add atlas.json and cover image

“detached HEAD”(떨어져나온 HEAD) 상태에서는 작업을 하고 커밋을 만들면, 태그는 그대로 있으나 새로운 커밋이 하나 쌓이 상태가 되고 새 커밋에 도달할 수 있는 방법이 따로 없게 된다. 물론 커밋의 해시 값을 정확히 기억하고 있으면 가능하긴 하다. 특정 태그의 상태에서 새로 작성한 커밋이 버그 픽스와 같이 의미있도록 하려면 반드시 브랜치를 만들어서 작업하는 것이 좋다.

$ git checkout -b version2 v2.0.0

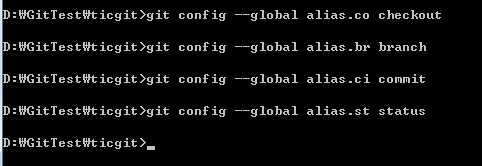
Switched to a new branch 'version2'

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

## Git Alias

Git의 기초를 마치기 전에 Git을 좀 더 쉽고 편안하게 쓸 수 있게 만들어 줄 Alias 라는 팁 알려주려 한다. 우리는 이 책에서 이 팁을 다시 거론하지 않고 이런 팁을 알고 있다고 가정한다. 그래서 알고 있는 것이 좋다.

명령을 완벽하게 입력하지 않으면 Git은 알아듣지 못한다. Git의 명령을 전부 입력하는 것이 귀찮다면 git config 를 사용하여 각 명령의 Alias을 쉽게 만들 수 있다. 아래는 Alias을 만드는 예이다.



이제 git commit 대신 git ci 만으로도 커밋할 수 있다. Git을 계속 사용한다면 다른 명령어도 자주 사용하게 될 것이다. 주저말고 자주 사용하는 명령은 Alias을 만들어 편하게 사용하시길 바란다.

이미 있는 명령을 편리하고 새로운 명령으로 만들어 사용할 수 있다. 예를 들어 파일을 Unstaged 상태로 변경하는 명령을 만들어서 불편함을 덜 수 있다. 아래와 같이 unstage 라는 Alias을 만든다.

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

아래 두 명령은 동일한 명령이다.

$ git unstage fileA

$ git reset HEAD -- fileA

한결 간결해졌다. 추가로 last 명령을 만들어 보자:

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

이제 최근 커밋을 좀 더 쉽게 확인할 수 있다.

$ git last

commit 66938dae3329c7aebe598c2246a8e6af90d04646

Author: Josh Goebel <dreamer3@example.com>

Date: Tue Aug 26 19:48:51 2008 +0800

test for current head

Signed-off-by: Scott Chacon <schacon@example.com>

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

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

## 요약

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

# Git 브랜치

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

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

## 브랜치란 무엇인가

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

Git은 데이터를 Change Set이나 변경사항(Diff)으로 기록하지 않고 일련의 스냅샷으로 기록한다는 것을 [시작하기](https://git-scm.com/book/ko/v2/ch00/ch01-getting-started) 에서 보여줬다.

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

파일이 3개 있는 디렉토리가 하나 있고 이 파일을 Staging Area에 저장하고 커밋하는 예제를 살펴 보자. 파일을 Stage 하면 Git 저장소에 파일을 저장하고(Git은 이것을 Blob이라고 부른다) Staging Area에 해당 파일의 체크섬을 저장한다([시작하기](https://git-scm.com/book/ko/v2/ch00/ch01-getting-started) 에서 살펴본 SHA-1을 사용한다).

$ git add README test.rb LICENSE

$ git commit -m 'The initial commit of my project'

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

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



[그림 8] 커밋과 트리 데이터

다시 파일을 수정하고 커밋하면 이전 커밋이 무엇인지도 저장한다.



[그림 9] 커밋과 이전 커밋

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

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



[그림 10] 브랜치와 커밋 히스토리

**새 브랜치 생성하기**

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

$ git branch testing

새로 만든 브랜치도 지금 작업하고 있던 마지막 커밋을 가리킨다.



[그림 11] 한 커밋 히스토리를 가리키는 두 브랜치

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



[그림 12] 현재 작업 중인 브랜치를 가리키는 HEAD

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

$ git log --oneline --decorate

f30ab (HEAD -> master, testing) add feature #32 - ability to add new formats to the central interface

34ac2 Fixed bug #1328 - stack overflow under certain conditions

98ca9 The initial commit of my project

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

**브랜치 이동하기**

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

$ git checkout testing

이렇게 하면 HEAD는 testing 브랜치를 가리킨다.



[그림 13] HEAD는 testing 브랜치를 가리킴

자, 이제 핵심이 보일 거다! 커밋을 새로 한 번 해보자.

$ vim test.rb

$ git commit -a -m 'made a change'



[그림 14] HEAD가 가리키는 testing 브랜치가 새 커밋을 가리킴

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

$ git checkout master



[그림 15] HEAD가 Checkout 한 브랜치로 이동함

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

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

파일을 수정하고 다시 커밋을 해보자.

$ vim test.rb

$ git commit -a -m 'made other changes'

프로젝트 히스토리는 분리돼 진행한다([갈라지는 브랜치](https://git-scm.com/book/ko/v2/ch00/divergent_history)). 우리는 브랜치를 하나 만들어 그 브랜치에서 일을 좀 하고, 다시 원래 브랜치로 되돌아와서 다른 일을 했다. 두 작업 내용은 서로 독립적으로 각 브랜치에 존재한다. 커밋 사이를 자유롭게 이동하다가 때가 되면 두 브랜치를 Merge 한다. 간단히 branch, checkout, commit 명령을 써서 말이다.



[그림 16] 갈라지는 브랜치

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

$ git log --oneline --decorate --graph --all

\* c2b9e (HEAD, master) made other changes

| \* 87ab2 (testing) made a change

|/

\* f30ab add feature #32 - ability to add new formats to the

\* 34ac2 fixed bug #1328 - stack overflow under certain conditions

\* 98ca9 initial commit of my project

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

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

이제 왜 그렇게 브랜치를 수시로 만들고 사용해야 하는지 알아보자.

## 브랜치와 Merge의 기초

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

1. 웹사이트가 있고 뭔가 작업을 진행하고 있다.
2. 새로운 이슈를 처리할 새 Branch를 하나 생성한다.
3. 새로 만든 Branch에서 작업을 진행한다.

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

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

**브랜치의 기초**

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



[그림 17] 현재 커밋 히스토리

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

$ git checkout -b iss53

Switched to a new branch "iss53"

위 명령은 아래 명령을 줄여놓은 것이다.

$ git branch iss53

$ git checkout iss53



[그림 18] 브랜치 포인터를 새로 만듦

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

$ vim index.html

$ git commit -a -m 'added a new footer [issue 53]'



[그림 19] 진행 중인 iss53 브랜치

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

그렇지만, 브랜치를 이동하려면 해야 할 일이 있다. 아직 커밋하지 않은 파일이 Checkout 할 브랜치와 충돌 나면 브랜치를 변경할 수 없다. 브랜치를 변경할 때는 워킹 디렉토리를 정리하는 것이 좋다. 이런 문제를 다루는 방법은(주로, Stash이나 커밋 Amend에 대해) 나중에 [Stashing과 Cleaning](https://git-scm.com/book/ko/v2/ch00/_git_stashing) 에서 다룰 것이다. 지금은 작업하던 것을 모두 커밋하고 master 브랜치로 옮긴다:

$ git checkout master

Switched to branch 'master'

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

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

$ git checkout -b hotfix

Switched to a new branch 'hotfix'

$ vim index.html

$ git commit -a -m 'fixed the broken email address'

[hotfix 1fb7853] fixed the broken email address

1 file changed, 2 insertions(+)



[그림 20] master 브랜치에서 갈라져 나온 hotfix 브랜치

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

$ git checkout master

$ git merge hotfix

Updating f42c576..3a0874c

Fast-forward

index.html | 2 ++

1 file changed, 2 insertions(+)

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

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



[그림 21] Merge 후 hotfix 같은 것을 가리키는 master 브랜치

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

$ git branch -d hotfix

Deleted branch hotfix (3a0874c).

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

$ git checkout iss53

Switched to branch "iss53"

$ vim index.html

$ git commit -a -m 'finished the new footer [issue 53]'

[iss53 ad82d7a] finished the new footer [issue 53]

1 file changed, 1 insertion(+)



[그림 21] master와 별개로 진행하는 iss53 브랜치

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

**Merge의 기초**

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

$ git checkout master

Switched to branch 'master'

$ git merge iss53

Merge made by the 'recursive' strategy.

index.html | 1 +

1 file changed, 1 insertion(+)

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



[그림 23] 커밋 3개를 Merge

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



[그림 24] Merge 커밋

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

$ git branch -d iss53

**충돌의 기초**

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

$ git merge iss53

Auto-merging index.html

CONFLICT (content): Merge conflict in index.html

Automatic merge failed; fix conflicts and then commit the result.

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

$ git status

On branch master

You have unmerged paths.

(fix conflicts and run "git commit")

Unmerged paths:

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

both modified: index.html

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

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

<<<<<<< HEAD:index.html

<div id="footer">contact : email.support@github.com</div>

=======

<div id="footer">

please contact us at support@github.com

</div>

>>>>>>> iss53:index.html

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

<div id="footer">

please contact us at email.support@github.com

</div>

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

다른 Merge 도구도 충돌을 해결할 수 있다. git mergetool 명령으로 실행한다.

$ 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):

기본 도구 말고 사용할 수 있는 다른 Merge 도구도 있는데(Mac에서는 opendiff 가 실행된다), “one of the following tools.” 부분에 보여준다. 여기에 표시된 도구 중 하나를 고를 수 있다.

|  |  |
| --- | --- |
| Note | Merge 시에 발생한 충돌을 다루는 더 어렵고 요상한 내용은 뒤에 [고급 Merge](https://git-scm.com/book/ko/v2/ch00/_advanced_merging) 에서 다루기로 한다. |

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

$ git status

On branch master

All conflicts fixed but you are still merging.

(use "git commit" to conclude merge)

Changes to be committed:

modified: index.html

충돌을 해결하고 나서 해당 파일이 Staging Area에 저장됐는지 확인했으면 git commit 명령으로 Merge 한 것을 커밋한다. 충돌을 해결하고 Merge 할 때는 커밋 메시지가 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.

# Please enter the commit message for your changes. Lines starting

# with '#' will be ignored, and an empty message aborts the commit.

# On branch master

# All conflicts fixed but you are still merging.

#

# Changes to be committed:

# modified: index.html

#

어떻게 충돌을 해결했고 좀 더 확인해야 하는 부분은 무엇이고 왜 그렇게 해결했는지에 대해서 자세하게 기록한다. 자세한 기록은 나중에 이 Merge 커밋을 이해하는데 도움을 준다.

## 브랜치 관리

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

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

$ git branch

iss53

\* master

testing

\* 기호가 붙어 있는 master 브랜치는 현재 Checkout 해서 작업하는 브랜치를 나타낸다. 즉, 지금 수정한 내용을 커밋하면 master 브랜치에 커밋되고 포인터가 앞으로 한 단계 나아간다. git branch -v 명령을 실행하면 브랜치마다 마지막 커밋 메시지도 함께 보여준다.

$ git branch -v

iss53 93b412c fix javascript issue

\* master 7a98805 Merge branch 'iss53'

testing 782fd34 add scott to the author list in the readmes

각 브랜치가 지금 어떤 상태인지 확인하기에 좋은 옵션도 있다. 현재 Checkout 한 브랜치를 기준으로 --merged 와 --no-merged 옵션을 사용하여 Merge 된 브랜치인지 그렇지 않은지 필터링해 볼 수 있다. git branch --merged 명령으로 이미 Merge 한 브랜치 목록을 확인한다.

$ git branch --merged

iss53

\* master

iss53 브랜치는 앞에서 이미 Merge 했기 때문에 목록에 나타난다. \* 기호가 붙어 있지 않은 브랜치는 git branch -d 명령으로 삭제해도 되는 브랜치다. 이미 다른 브랜치와 Merge 했기 때문에 삭제해도 정보를 잃지 않는다.

반대로 현재 Checkout 한 브랜치에 Merge 하지 않은 브랜치를 살펴보려면 git branch --no-merged 명령을 사용한다.

$ git branch --no-merged

testing

위에는 없었던 다른 브랜치가 보인다. 아직 Merge 하지 않은 커밋을 담고 있기 때문에 git branch -d 명령으로 삭제되지 않는다.

$ git branch -d testing

error: The branch 'testing' is not fully merged.

If you are sure you want to delete it, run 'git branch -D testing'.

Merge 하지 않은 브랜치를 강제로 삭제하려면 -D 옵션으로 삭제한다.

|  |  |
| --- | --- |
| Tip | 위에서 설명한 --merged, --no-merged 옵션을 사용할 때 커밋이나 브랜치 이름을 지정해주지 않으면 현재 브랜치를 기준으로 Merge 되거나 Merge 되지 않은 내용을 출력한다.  위 명령을 사용할 때 특정 브랜치를 기준으로 Merge 되거나 혹은 Merge 되지 않은 브랜치 정보를 살펴보려면 명령에 브랜치 이름을 지정해주면 된다. 예를 들어 master 브랜치에 아직 Merge되지 않은 브랜치를 살펴보려면 다음과 같은 명령을 실행한다.  $ git checkout testing  $ git branch --no-merged master  topicA  featureB |

## 브랜치 워크플로

브랜치를 만들고 Merge 하는 것을 어디에 써먹어야 할까. 이 절에서는 Git 브랜치가 유용한 몇 가지 워크플로를 살펴본다. 여기서 설명하는 워크플로를 개발에 적용하면 도움이 될 것이다.

**Long-Running 브랜치**

Git은 꼼꼼하게 3-way Merge를 사용하기 때문에 장기간에 걸쳐서 한 브랜치를 다른 브랜치와 여러 번 Merge 하는 것이 쉬운 편이다. 그래서 개발 과정에서 필요한 용도에 따라 브랜치를 만들어 두고 계속 사용할 수 있다. 그리고 정기적으로 브랜치를 다른 브랜치로 Merge 한다.

이런 접근법에 따라서 Git 개발자가 많이 선호하는 워크플로가 하나 있다. 배포했거나 배포할 코드만 master 브랜치에 Merge 해서 안정 버전의 코드만 master 브랜치에 둔다. 개발을 진행하고 안정화하는 브랜치는 develop 이나 next 라는 이름으로 추가로 만들어 사용한다. 이 브랜치는 언젠가 안정 상태가 되겠지만, 항상 안정 상태를 유지해야 하는 것이 아니다. 테스트를 거쳐서 안정적이라고 판단되면 master 브랜치에 Merge 한다. 토픽 브랜치(앞서 살펴본 iss53 브랜치 같은 짧은 호흡 브랜치)에도 적용할 수 있는데, 해당 토픽을 처리하고 테스트해서 버그도 없고 안정적이면 그때 Merge 한다.

사실 우리가 얘기하는 것은 커밋을 가리키는 포인터에 대한 얘기다. 커밋 포인터를 만들고 수정하고 분리하고 합치는지에 대한 것이다. 개발 브랜치는 공격적으로 히스토리를 만들어 나아가고 안정 브랜치는 이미 만든 히스토리를 뒤따르며 나아간다.



[그림 25] 안정적인 브랜치일수록 커밋 히스토리가 뒤쳐짐

실험실에서 충분히 테스트하고 실전에 배치하는 과정으로 보면 이해하기 쉽다



[그림 26] 각 브랜치를 하나의 “실험실” 로 생각

코드를 여러 단계로 나누어 안정성을 높여가며 운영할 수 있다. 프로젝트 규모가 크면 proposed 혹은 pu (proposed updates)라는 이름의 브랜치를 만들고 next 나 master 브랜치에 아직 Merge 할 준비가 되지 않은 것을 일단 Merge 시킨다. 중요한 개념은 브랜치를 이용해 여러 단계에 걸쳐서 안정화해 나아가면서 충분히 안정화가 됐을 때 안정 브랜치로 Merge 한다는 점이다. 다시 말해서 Long-Running의 브랜치가 여러 개일 필요는 없지만 정말 유용하다는 점이다. 특히 규모가 크고 복잡한 프로젝트일수록 그 유용성이 반짝반짝 빛난다.

**토픽 브랜치**

토픽 브랜치는 프로젝트 크기에 상관없이 유용하다. 토픽 브랜치는 어떤 한 가지 주제나 작업을 위해 만든 짧은 호흡의 브랜치다. 다른 버전 관리 시스템에서는 이런 브랜치를 본 적이 없을 것이다. Git이 아닌 다른 버전 관리 도구에서는 브랜치를 하나 만드는 데 큰 비용이 든다. Git에서는 매우 일상적으로 브랜치를 만들고 Merge 하고 삭제한다.

앞서 사용한 iss53 이나 hotfix 브랜치가 토픽 브랜치다. 우리는 브랜치를 새로 만들고 어느 정도 커밋하고 나서 다시 master 브랜치에 Merge 하고 브랜치 삭제도 해 보았다. 보통 주제별로 브랜치를 만들고 각각은 독립돼 있기 때문에 매우 쉽게 컨텍스트 사이를 옮겨 다닐 수 있다. 묶음별로 나눠서 일하면 내용별로 검토하기에도, 테스트하기에도 더 편하다. 각 작업을 하루든 한 달이든 유지하다가 master 브랜치에 Merge 할 시점이 되면 순서에 관계없이 그때 Merge 하면 된다.

master 브랜치를 checkout 한 상태에서 어떤 작업을 한다고 해보자. 한 이슈를 처리하기 위해서 iss91 브랜치를 만들고 해당 작업을 한다. 같은 이슈를 다른 방법으로 해결해보고 싶을 때도 있다. iss91v2 브랜치를 만들고 다른 방법을 시도해 본다. 확신할 수 없는 아이디어를 적용해보기 위해 다시 master 브랜치로 되돌아가서 dumbidea 브랜치를 하나 더 만든다. 지금까지 말했던 커밋 히스토리는 아래 그림 같다.



[그림 27] 토픽 브랜치가 많음

이슈를 처리했던 방법 중 두 번째 방법인 iss91v2 브랜치가 괜찮아서 적용하기로 결정했다. 그리고 아이디어를 확신할 수 없었던 dumbidea 브랜치를 같이 일하는 다른 개발자에게 보여줬더니 썩 괜찮다는 반응을 얻었다. iss91 브랜치는 (C5, C6 커밋도 함께) 버리고 다른 두 브랜치를 Merge 하면 아래 그림과 같이 된다.



[그림 28] dumbidea 와 iss91v2 브랜치를 Merge 하고 난 후의 모습

[분산 환경에서의 Git](https://git-scm.com/book/ko/v2/ch00/ch05-distributed-git)에서 프로젝트를 Git으로 관리할 때 브랜치를 이용하여 만들 수 있는 여러 워크플로에 대해 살펴본다. 관련 부분을 살펴보면 프로젝트에 어떤 형태로 응용할수 있을 지 감이 올 것이다.

지금까지 한 작업은 전부 로컬에서만 처리한다는 것을 꼭 기억하자. 로컬 저장소에서만 브랜치를 만들고 Merge 했으며 서버와 통신을 주고받는 일은 없었다.

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

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

|  |  |
| --- | --- |
| Note | “origin” 의 의미  브랜치 이름으로 많이 사용하는 “master” 라는 이름이 괜히 특별한 의미를 가지는 게 아닌 것처럼 “origin” 도 특별한 의미가 있는 것은 아니다. git init 명령이 자동으로 만들기 때문에 사용하는 이름인 “master” 와 마찬가지로 “origin” 도 git clone 명령이 자동으로 만들어주는 리모트 이름이다. git clone -o booyah 라고 옵션을 주고 명령을 실행하면 booyah/master 라고 사용자가 정한 대로 리모트 이름을 생성해준다. |



[그림 29] Clone 이후 서버와 로컬의 master 브랜치

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



[그림 30] 로컬과 서버의 커밋 히스토리는 독립적임

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



[그림 31] git fetch 명령은 리모트 브랜치 정보를 업데이트

리모트 저장소를 여러 개 운영하는 상황을 이해할 수 있도록 개발용으로 사용할 Git 저장소를 팀 내부에 하나 추가해 보자. 이 저장소의 주소가 git.team1.ourcompany.com 이며 [Git의 기초](https://git-scm.com/book/ko/v2/ch00/ch02-git-basics-chapter)에서 살펴본 git remote add 명령으로 현재 작업 중인 프로젝트에 팀의 저장소를 추가한다. 이름을 teamone 으로 짓고 긴 서버 주소 대신 사용한다.



[그림 32] 서버를 리모트 저장소로 추가

서버를 추가하고 나면 git fetch teamone 명령으로 teamone 서버의 데이터를 내려받는다. 명령을 실행해도 teamone 서버의 데이터는 모두 origin 서버에도 있는 것들이라서 아무것도 내려받지 않는다. 하지만, 이 명령은 리모트 트래킹 브랜치 teamone/master 가 teamone 서버의 master 브랜치가 가리키는 커밋을 가리키게 한다.



[그림 32] teamone/master 의 리모트 트래킹 브랜치

**Push 하기**

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

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

$ git push origin serverfix

Counting objects: 24, done.

Delta compression using up to 8 threads.

Compressing objects: 100% (15/15), done.

Writing objects: 100% (24/24), 1.91 KiB | 0 bytes/s, done.

Total 24 (delta 2), reused 0 (delta 0)

To https://github.com/schacon/simplegit

\* [new branch] serverfix -> serverfix

Git은 serverfix라는 브랜치 이름을 refs/heads/serverfix:refs/heads/serverfix 로 확장한다. 이것은 serverfix 라는 로컬 브랜치를 서버로 Push 하는데 리모트의 serverfix 브랜치로 업데이트한다는 것을 의미한다. 나중에 [Git의 내부](https://git-scm.com/book/ko/v2/ch00/ch10-git-internals)에서 refs/heads/ 의 뜻을 자세히 알아볼 것이기 때문에 일단 넘어가도록 한다. git push origin serverfix:serverfix 라고 Push 하는 것도 같은 의미인데 이것은 “로컬의 serverfix 브랜치를 리모트 저장소의 serverfix 브랜치로 Push 하라” 라는 뜻이다. 로컬 브랜치의 이름과 리모트 서버의 브랜치 이름이 다를 때 필요하다. 리모트 저장소에 serverfix 라는 이름 대신 다른 이름을 사용하려면 git push origin serverfix:awesomebranch 처럼 사용한다.

|  |  |
| --- | --- |
| Note | 암호를 매번 입력하지 않아도 된다  HTTPS URL로 시작하는 리모트 저장소를 사용한다면 아마도 Push 나 Pull을 할 때 인증을 위한 사용자이름이나 암호를 묻는 것을 볼 수 있다. 보통 터미널에서 작업하는 경우 Git이 이 정보를 사용자로부터 받기 위해 사용자이름이나 암호를 입력받아 서버로 전달해서 권한을 확인한다.  이 리모트에 접근할 때마다 매번 사용자이름나 암호를 입력하지 않도록 “credential cache” 기능을 이용할 수 있다. 이 기능을 활성화하면 Git은 몇 분 동안 입력한 사용자이름이나 암호를 저장해둔다. 이 기능을 활성화하려면 git config --global credential.helper cache 명령을 실행하여 환경설정을 추가한다.  이 기능이 제공하는 다른 옵션에 대한 자세한 설명은 [Credential 저장소](https://git-scm.com/book/ko/v2/ch00/_credential_caching)를 참고한다. |

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

$ git fetch origin

remote: Counting objects: 7, done.

remote: Compressing objects: 100% (2/2), done.

remote: Total 3 (delta 0), reused 3 (delta 0)

Unpacking objects: 100% (3/3), done.

From https://github.com/schacon/simplegit

\* [new branch] serverfix -> origin/serverfix

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

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

$ git checkout -b serverfix origin/serverfix

Branch serverfix set up to track remote branch serverfix from origin.

Switched to a new branch 'serverfix'

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

**브랜치 추적**

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

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

$ git checkout --track origin/serverfix

Branch serverfix set up to track remote branch serverfix from origin.

Switched to a new branch 'serverfix'

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

$ git checkout serverfix

Branch serverfix set up to track remote branch serverfix from origin.

Switched to a new branch 'serverfix'

리모트 브랜치와 다른 이름으로 브랜치를 만들려면 로컬 브랜치의 이름을 아래와 같이 다르게 지정한다.

$ git checkout -b sf origin/serverfix

Branch sf set up to track remote branch serverfix from origin.

Switched to a new branch 'sf'

이제 sf 브랜치에서 Push 나 Pull 하면 자동으로 origin/serverfix 로 데이터를 보내거나 가져온다.

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

$ git branch -u origin/serverfix

Branch serverfix set up to track remote branch serverfix from origin.

|  |  |
| --- | --- |
| Note | Upstream 별명  추적 브랜치를 설정했다면 추적 브랜치 이름을 @{upstream} 이나 @{u} 로 짧게 대체하여 사용할 수 있다. master 브랜치가 origin/master 브랜치를 추적하는 경우라면 git merge origin/master 명령과 git merge @{u} 명령을 똑같이 사용할 수 있다. |

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

$ git branch -vv

iss53 7e424c3 [origin/iss53: ahead 2] forgot the brackets

master 1ae2a45 [origin/master] deploying index fix

\* serverfix f8674d9 [teamone/server-fix-good: ahead 3, behind 1] this should do it

testing 5ea463a trying something new

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

여기서 중요한 점은 명령을 실행했을 때 나타나는 결과는 모두 마지막으로 서버에서 데이터를 가져온(fetch) 시점을 바탕으로 계산한다는 점이다. 단순히 이 명령만으로는 서버의 최신 데이터를 반영하지는 않으며 로컬에 저장된 서버의 캐시 데이터를 사용한다. 현재 시점에서 진짜 최신 데이터로 추적 상황을 알아보려면 먼저 서버로부터 최신 데이터를 받아온 후에 추적 상황을 확인해야 한다. 아래처럼 두 명령을 이어서 사용하는 것이 적당하다 하겠다.

$ git fetch --all; git branch -vv

**Pull 하기**

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

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

**리모트 브랜치 삭제**

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

$ git push origin --delete serverfix

To https://github.com/schacon/simplegit

- [deleted] serverfix

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

## Rebase 하기

Git에서 한 브랜치에서 다른 브랜치로 합치는 방법으로는 두 가지가 있다. 하나는 Merge 이고 다른 하나는 Rebase 다. 이 절에서는 Rebase가 무엇인지, 어떻게 사용하는지, 좋은 점은 뭐고, 어떤 상황에서 사용하고 어떤 상황에서 사용하지 말아야 하는지 알아 본다.

**Rebase의 기초**

앞의 [Merge 의 기초](https://git-scm.com/book/ko/v2/ch00/_basic_merging) 절에서 살펴본 예제로 다시 돌아가 보자. 두 개의 나누어진 브랜치의 모습을 볼 수 있다.



[그림 34] 두 개의 브랜치로 나누어진 커밋 히스토리

이 두 브랜치를 합치는 가장 쉬운 방법은 앞에서 살펴본 대로 merge 명령을 사용하는 것이다. 두 브랜치의 마지막 커밋 두 개(C3, C4)와 공통 조상(C2)을 사용하는 3-way Merge로 새로운 커밋을 만들어 낸다.



[그림 35] 나뉜 브랜치를 Merge 하기

비슷한 결과를 만드는 다른 방식으로, C3 에서 변경된 사항을 Patch로 만들고 이를 다시 C4 에 적용시키는 방법이 있다. Git에서는 이런 방식을 Rebase 라고 한다. rebase 명령으로 한 브랜치에서 변경된 사항을 다른 브랜치에 적용할 수 있다.

위의 예제는 아래와 같은 명령으로 Rebase 한다.

$ git checkout experiment

$ git rebase master

First, rewinding head to replay your work on top of it...

Applying: added staged command

실제로 일어나는 일을 설명하자면 일단 두 브랜치가 나뉘기 전인 공통 커밋으로 이동하고 나서 그 커밋부터 지금 Checkout 한 브랜치가 가리키는 커밋까지 diff를 차례로 만들어 어딘가에 임시로 저장해 놓는다. Rebase 할 브랜치(역주 - experiment)가 합칠 브랜치(역주 - master)가 가리키는 커밋을 가리키게 하고 아까 저장해 놓았던 변경사항을 차례대로 적용한다.



[그림 36] `C4`의 변경사항을 `C3`에 적용하는 Rebase 과정

그리고 나서 master 브랜치를 Fast-forward 시킨다.

$ git checkout master

$ git merge experiment



[그림 37] master 브랜치를 Fast-forward시키기

C4' 로 표시된 커밋에서의 내용은 Merge 예제에서 살펴본 C5 커밋에서의 내용과 같을 것이다. Merge 이든 Rebase 든 둘 다 합치는 관점에서는 서로 다를 게 없다. 하지만, Rebase가 좀 더 깨끗한 히스토리를 만든다. Rebase 한 브랜치의 Log를 살펴보면 히스토리가 선형이다. 일을 병렬로 동시에 진행해도 Rebase 하고 나면 모든 작업이 차례대로 수행된 것처럼 보인다.

Rebase는 보통 리모트 브랜치에 커밋을 깔끔하게 적용하고 싶을 때 사용한다. 아마 이렇게 Rebase 하는 리모트 브랜치는 직접 관리하는 것이 아니라 그냥 참여하는 브랜치일 것이다. 메인 프로젝트에 Patch를 보낼 준비가 되면 하는 것이 Rebase 니까 브랜치에서 하던 일을 완전히 마치고 origin/master 로 Rebase 한다. 이렇게 Rebase 하고 나면 프로젝트 관리자는 어떠한 통합작업도 필요 없다. 그냥 master 브랜치를 Fast-forward 시키면 된다.

Rebase를 하든지, Merge를 하든지 최종 결과물은 같고 커밋 히스토리만 다르다는 것이 중요하다. Rebase 의 경우는 브랜치의 변경사항을 순서대로 다른 브랜치에 적용하면서 합치고 Merge 의 경우는 두 브랜치의 최종결과만을 가지고 합친다.

**Rebase 활용**

Rebase는 단순히 브랜치를 합치는 것만 아니라 다른 용도로도 사용할 수 있다. [다른 토픽 브랜치에서 갈라져 나온 토픽 브랜치](https://git-scm.com/book/ko/v2/ch00/rbdiag_e) 같은 히스토리가 있다고 하자. server 브랜치를 만들어서 서버 기능을 추가하고 그 브랜치에서 다시 client 브랜치를 만들어 클라이언트 기능을 추가한다. 마지막으로 server 브랜치로 돌아가서 몇 가지 기능을 더 추가한다.



[그림 38] 다른 토픽 브랜치에서 갈라져 나온 토픽 브랜치

이때 테스트가 덜 된 server 브랜치는 그대로 두고 client 브랜치만 master 로 합치려는 상황을 생각해보자. server 와는 아무 관련이 없는 client 커밋은 C8, C9 이다. 이 두 커밋을 master 브랜치에 적용하기 위해서 --onto 옵션을 사용하여 아래와 같은 명령을 실행한다:

$ git rebase --onto master server client

이 명령은 master 브랜치부터 server 브랜치와 client 브랜치의 공통 조상까지의 커밋을 client 브랜치에서 없애고 싶을 때 사용한다. client 브랜치에서만 변경된 패치를 만들어 master 브랜치에서 client 브랜치를 기반으로 새로 만들어 적용한다. 조금 복잡하긴 해도 꽤 쓸모 있다.



[그림 39] 다른 토픽 브랜치에서 갈라져 나온 토픽 브랜치를 Rebase 하기

이제 master 브랜치로 돌아가서 Fast-forward 시킬 수 있다([master 브랜치를 client 브랜치 위치로 진행 시키기](https://git-scm.com/book/ko/v2/ch00/rbdiag_g) 참고).

$ git checkout master

$ git merge client



[그림 40] . master 브랜치를 client 브랜치 위치로 진행 시키기

server 브랜치의 일이 다 끝나면 git rebase <basebranch> <topicbranch> 라는 명령으로 Checkout 하지 않고 바로 server 브랜치를 master 브랜치로 Rebase 할 수 있다. 이 명령은 토픽(server) 브랜치를 Checkout 하고 베이스(master) 브랜치에 Rebase 한다.

$ git rebase master server

server 브랜치의 수정사항을 master 브랜치에 적용했다. 그 결과는 [master 브랜치에 server 브랜치의 수정 사항을 적용](https://git-scm.com/book/ko/v2/ch00/rbdiag_h) 같다.



[그림 41] master 브랜치에 server 브랜치의 수정 사항을 적용

그리고 나서 master 브랜치를 Fast-forward 시킨다.

$ git checkout master

$ git merge server

모든 것이 master 브랜치에 통합됐기 때문에 더 필요하지 않다면 client 나 server 브랜치는 삭제해도 된다. 브랜치를 삭제해도 커밋 히스토리는 [최종 커밋 히스토리](https://git-scm.com/book/ko/v2/ch00/rbdiag_i) 같이 여전히 남아 있다.

$ git branch -d client

$ git branch -d server



[그림 42] master 브랜치에 server 브랜치의 수정 사항을 적용

**Rebase 의 위험성**

Rebase가 장점이 많은 기능이지만 단점이 없는 것은 아니니 조심해야 한다. 그 주의사항은 아래 한 문장으로 표현할 수 있다.

**이미 공개 저장소에 Push 한 커밋을 Rebase 하지 마라**

이 지침만 지키면 Rebase를 하는 데 문제 될 게 없다. 하지만, 이 주의사항을 지키지 않으면 사람들에게 욕을 먹을 것이다.

Rebase는 기존의 커밋을 그대로 사용하는 것이 아니라 내용은 같지만 다른 커밋을 새로 만든다. 새 커밋을 서버에 Push 하고 동료 중 누군가가 그 커밋을 Pull 해서 작업을 한다고 하자. 그런데 그 커밋을 git rebase 로 바꿔서 Push 해버리면 동료가 다시 Push 했을 때 동료는 다시 Merge 해야 한다. 그리고 동료가 다시 Merge 한 내용을 Pull 하면 내 코드는 정말 엉망이 된다.

이미 공개 저장소에 Push 한 커밋을 Rebase 하면 어떤 결과가 초래되는지 예제를 통해 알아보자. 중앙 저장소에서 Clone 하고 일부 수정을 하면 커밋 히스토리는 아래와 같아 진다.



[그림 43] 저장소를 Clone 하고 일부 수정함

이제 팀원 중 누군가 커밋, Merge 하고 나서 서버에 Push 한다. 이 리모트 브랜치를 Fetch, Merge 하면 히스토리는 아래와 같이 된다.



[그림 44] . Fetch 한 후 Merge 함

그런데 Push 했던 팀원은 Merge 한 일을 되돌리고 다시 Rebase 한다. 서버의 히스토리를 새로 덮어씌우려면 git push --force 명령을 사용해야 한다. 이후에 저장소에서 Fetch 하고 나면 아래 그림과 같은 상태가 된다.



[그림 45] 한 팀원이 다른 팀원이 의존하는 커밋을 없애고 Rebase 한 커밋을 다시 Push 함

자 이렇게 되면 짬뽕이 된다. git pull 로 서버의 내용을 가져와서 Merge 하면 같은 내용의 수정사항을 포함한 Merge 커밋이 아래와 같이 만들어진다.



[그림 46] 같은 Merge를 다시 한다

git log 로 히스토리를 확인해보면 저자, 커밋 날짜, 메시지가 같은 커밋이 두 개 있다(C4, C4'). 이렇게 되면 혼란스럽다. 게다가 이 히스토리를 서버에 Push 하면 같은 커밋이 두 개 있기 때문에 다른 사람들도 혼란스러워한다. `C4`와 `C6`는 포함되지 말았어야 할 커밋이다. 애초에 서버로 데이터를 보내기 전에 Rebase로 커밋을 정리했어야 했다.

**Rebase 한 것을 다시 Rebase 하기**

만약 이런 상황에 빠질 때 유용한 Git 기능이 하나 있다. 어떤 팀원이 강제로 내가 한일을 덮어썼다고 하자. 그러면 내가 했던 일이 무엇이고 덮어쓴 내용이 무엇인지 알아내야 한다.

커밋 SHA 체크섬 외에도 Git은 커밋에 Patch 할 내용으로 SHA-1 체크섬을 한번 더 구한다. 이 값은 “patch-id” 라고 한다.

덮어쓴 커밋을 받아서 그 커밋을 기준으로 Rebase 할 때 Git은 원래 누가 작성한 코드인지 잘 찾아 낸다. 그래서 Patch가 원래대로 잘 적용된다.

예를 들어 앞서 살펴본 예제를 보면 [한 팀원이 다른 팀원이 의존하는 커밋을 없애고 Rebase 한 커밋을 다시 Push 함](https://git-scm.com/book/ko/v2/ch00/_pre_merge_rebase_work) 상황에서 Merge 하는 대신 git rebase teamone/master 명령을 실행하면 Git은 아래와 같은 작업을 한다.

* 현재 브랜치에만 포함된 커밋을 찾는다. (C2, C3, C4, C6, C7)
* Merge 커밋을 가려낸다. (C2, C3, C4)
* 이 중 덮어쓰지 않은 커밋들만 골라낸다. (C2, C3. C4는 C4’와 동일한 Patch다)
* 남은 커밋들만 다시 teamone/master 바탕으로 커밋을 쌓는다.

결과를 확인해보면 [같은 Merge를 다시 한다](https://git-scm.com/book/ko/v2/ch00/_merge_rebase_work) 같은 결과 대신 제대로 정리된 [강제로 덮어쓴 브랜치에 Rebase 하기](https://git-scm.com/book/ko/v2/ch00/_rebase_rebase_work) 같은 결과를 얻을 수 있다.



[그림 47] 강제로 덮어쓴 브랜치에 Rebase 하기

동료가 생성했던 C4와 C4' 커밋 내용이 완전히 같을 때만 이렇게 동작된다. 커밋 내용이 아예 다르거나 비슷하다면 커밋이 두 개 생긴다(같은 내용이 두 번 커밋될 수 있기 때문에 깔끔하지 않다).

git pull 명령을 실행할 때 옵션을 붙여서 git pull --rebase 로 Rebase 할 수도 있다. 물론 git fetch 와 git rebase teamone/master 이 두 명령을 직접 순서대로 실행해도 된다.

git pull 명령을 실행할 때 기본적으로 --rebase 옵션이 적용되도록 pull.rebase 설정을 추가할 수 있다. git config --global pull.rebase true 명령으로 추가한다.

Push 하기 전에 정리하려고 Rebase 하는 것은 괜찮다. 또 절대 공개하지 않고 혼자 Rebase 하는 경우도 괜찮다. 하지만, 이미 공개하여 사람들이 사용하는 커밋을 Rebase 하면 틀림없이 문제가 생긴다.

나중에 후회하지 말고 git pull --rebase 로 문제를 미리 방지할 수 있다는 것을 같이 작업하는 동료와 모두 함께 공유하기 바란다.

**Rebase vs. Merge**

Merge가 뭔지, Rebase가 뭔지 여러 예제를 통해 간단히 살펴보았다. 지금쯤 이런 의문이 들 거로 생각한다. 둘 중 무엇을 쓰는 게 좋지? 이 질문에 대한 답을 찾기 전에 히스토리의 의미에 대해서 잠깐 다시 생각해보자.

히스토리를 보는 관점 중에 하나는 **작업한 내용의 기록**으로 보는 것이 있다. 작업 내용을 기록한 문서이고, 각 기록은 각각 의미를 가지며, 변경할 수 없다. 이런 관점에서 커밋 히스토리를 변경한다는 것은 역사를 부정하는 꼴이 된다. 언제 무슨 일이 있었는지 기록에 대해 *거짓말* 을 하게 되는 것이다. 이렇게 했을 때 지저분하게 수많은 Merge 커밋이 히스토리에 남게 되면 문제가 없을까? **역사**는 후세를 위해 기록하고 보존해야 한다.

히스토리를 **프로젝트가 어떻게 진행되었나에 대한 이야기**로도 볼 수 있다. 소프트웨어를 주의 깊게 편집하는 방법에 메뉴얼이나 세세한 작업내용을 초벌부터 공개하고 싶지 않을 수 있다. 나중에 다른 사람에게 들려주기 좋도록 Rebase 나 filter-branch 같은 도구로 프로젝트의 진행 이야기를 다듬으면 좋다.

Merge 나 Rebase 중 무엇이 나으냐는 질문은 다시 생각해봐도 답이 그리 간단치 않다. Git은 매우 강력한 도구고 기능이 많아서 히스토리를 잘 쌓을 수 있지만, 모든 팀과 모든 이가 처한 상황은 모두 다르다. 예제를 통해 Merge 나 Rebase가 무엇이고 어떤 의미인지 배웠다. 이 둘을 어떻게 쓸지는 각자의 상황과 각자의 판단에 달렸다.

일반적인 해답을 굳이 드리자면 로컬 브랜치에서 작업할 때는 히스토리를 정리하기 위해서 Rebase 할 수도 있지만, 리모트 등 어딘가에 Push로 내보낸 커밋에 대해서는 절대 Rebase 하지 말아야 한다.

## 요약

우리는 이 장에서 Git으로 브랜치를 만들고 Merge 의 기본적인 사용법을 다루었다. 이제 브랜치를 만들고 옮겨다니고 Merge 하는 것에 익숙해졌을 것으로 생각한다. 브랜치를 Rebase 해서 Push 해서 공유하는 것 정도는 어렵지 않게 할 수 있을 것이다. 다음 장에서는 Git 저장소 서버를 직접 운영하는 방법을 설명한다.

# Git 명령어 정리

## git checkout

git checkout <commit\_id> //특정 커밋시점으로 되돌린다.

git checkout <branch명>  //지정된 브랜치가 가리키는 커밋을 HEAD가 가리키게 한다. (작업 디렉토리의 파일은 그대로 유지됨)

git checkout -b <branch명> //새로운 브랜치를 만들고  해당 브랜치를 checkout 한다.

git checkout -b <new branch name> <server branch name> // 서버에서 생성되어 있는 브랜치 가져오기

git checkout  <commit\_id> -- <file> // 특정 커밋시점의 특정 파일만 되돌린다.

git checkout -- <file>  // work directory의 변경된 unstaged 파일을 최신 커밋된 내용으로 덮어쓴다. (작업중이던 내용이 있으면 날라가며, 복구 불가!!!)

git checkout -f // 현재 HEAD가 가리키는 커밋으로 작업 디렉토리의 파일을 되돌려 놓는다. (작업중이던 내용이 있으면 날라가며, 복구 불가!!!)

git checkout -f <branch>//  branch가 가리키는 커밋으로 작업 디렉토리의 파일을 되돌려 놓는다. (작업중이던 내용이 있으면 날라가며, 복구 불가!!!)

# GitHub

## 메인 저장소와 동기화(SYNCHRONIZE) 유지

자, 이제 오픈소스 프로젝트의 소스코드를 나의 작업공간까지 정상적으로 가져왔고, 수정사항이 있는 경우 이를 수정하여 이를 반영하면 됩니다. 그러나, 이에 대해서 알아보기 이전에 한 가지 중요한 점이 있습니다. 바로 메인 저장소와의 동기화(synchronize)를 유지하는 방법입니다. 여러 명의 개발자들과 하나의 프로젝트를 함께 수행해본 사람들은 잘 알겠지만, 소스코드를 수정하는 중간중간 메인 저장소와 동기화를 유지하지 않는 경우, 여러명이 동시에 수정한 사항들간의 충돌이 발생하여 소스코드의 변경사항이 원하는 대로 동작하지 않는 경우가 발생합니다. 그러므로, 처음 오픈소스 프로젝트를 fork 하였던 메인 저장소와 동기화를 유지하는 방식에 대해서 알아보도록 하겠습니다.

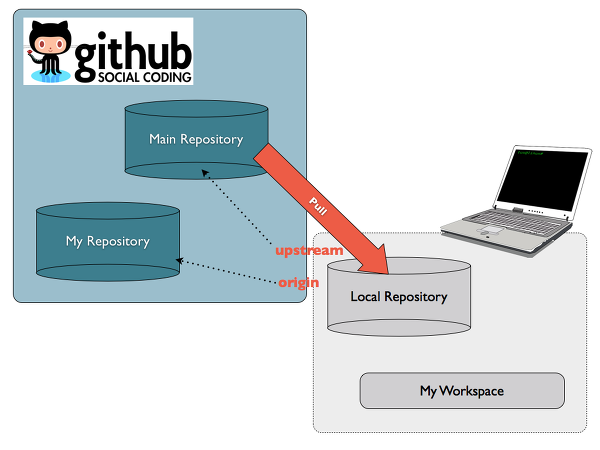
먼저, git에서 사용하는 remote 의 개념에 대해서 이해하고 넘어가도록 하겠습니다. Git에서 말하는 remote라는 것은 원격저장소(github)상의 특정 위치를 바라보는 참조(reference)라고 설명할 수 있습니다. github로 부터 프로젝트를 로컬 저장소로 clone한 경우, 자동적으로 origin 이라는 이름의 remote가 생성됩니다. 우리의 예시에서 설명하자면, origin은 나의 저장소를 바라보고 있는 remote라고 할 수 있습니다.

이제 나의 저장소가 아니라, 메인 저장소를 바라보는 upstream 이라는 이름의 remote를 생성해 보도록 하겠습니다. 우리의 경우처럼 메인프로젝트를 fork한 경우 메인프로젝트 저장소를 바라보는 remote를 주로 “upstream” 이라는 용어로 많이 사용합니다. upstream remote를 추가한다고 하여도 우리는 권한이 없기 때문에, upstream으로 변경사항을 push할 수는 없습니다.

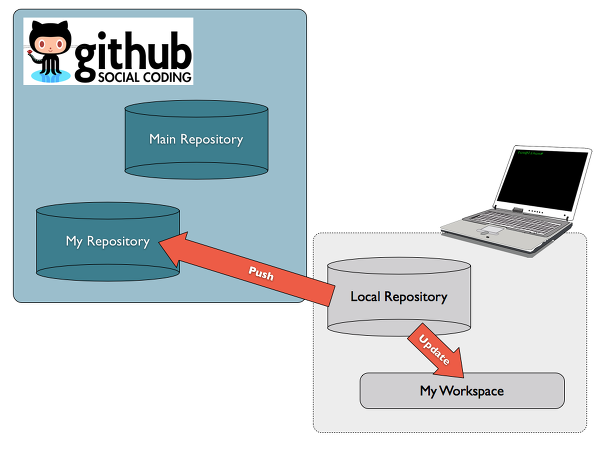
|  |
| --- |
| git remote  # origin    git remote add upstream https://github.com/elasticsearch/elasticsearch-cloud-aws.git    git remote  # origin  # upstream |

우리는 조금 전에 메인저장소로 부터 프로젝트를 fork하였기 때문에, 짧은 시간사이에 메인저장소에 변경사항이 있지는 않을 테지만, 한번 메인저장소로 부터 변경사항이 있는지 pull 명령어를 실행하여 보도록 하겠습니다. 변경된 사항이 없다면 아래와 같은 결과가 출력될 것입니다.

|  |  |
| --- | --- |
| 1  2  3  4 | git pull upstream master  # From https://github.com/elasticsearch/elasticsearch-cloud-aws  # \* branch master -> FETCH\_HEAD  # Already up-to-date. |

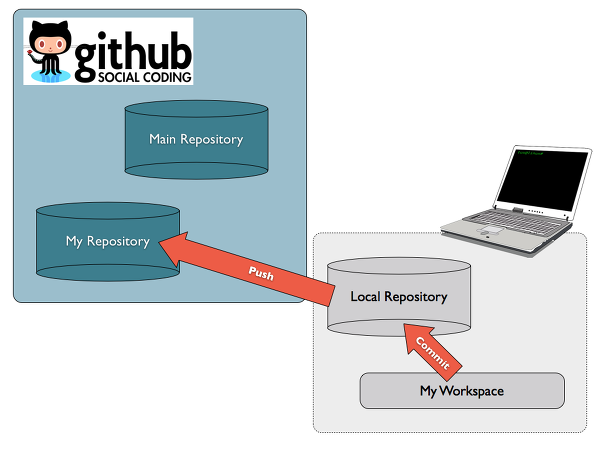
[](http://guruble.com/wp-content/uploads/2014/02/opensource_contributor-004.png)

만약, 메인저장소에 변경사항이 있었다면 해당사항이 로컬 저장소에 반영되고, 결과적으로 로컬 저장소의 소스코드는 github 상의 나의 저장소의 소스, 그리고 로컬 작업공간의 소스와 서로 다르게 됩니다. 이렇게 서로 다른 부분들을 동기화하기 위하여 나의 저장소에 변경사항을 push하고, 나의 작업공간에 변경사항을 update하도록 하겠습니다.

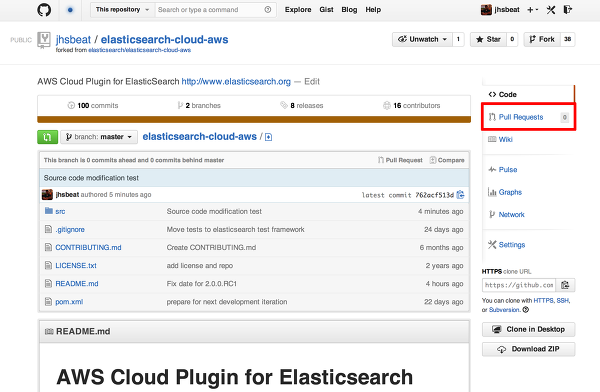
[](http://guruble.com/wp-content/uploads/2014/02/opensource_contributor-005.png)

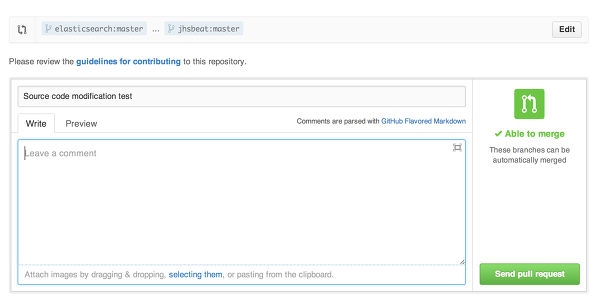
## 컨트리뷰트 하기

자, 이제 모든 준비는 완료되었습니다. 실제 소스코드의 버그를 수정하거나 기능을 개선하여 오픈소스에 기여하는 일만 남았습니다. 실제 소스코드의 내용을 수정하는 것은 각자의 몫이므로 이 부분은 생략하고 여기에서는 특정 소스가 수정되었다고 가정하고 설명하도록 하겠습니다. 작업공간에서 수정된 소스코드를 로컬 저장소에 commit 한 뒤, 해당사항을 나의 저장소에 push 합니다.



이제부터가 진짜 오픈소스 프로젝트 컨트리뷰터가 되기 위한 최종관문입니다. 아래의 그림과 같이 github의 나의 저장소에서 우측에 보이는 **Pull Request** 라는 버튼이 보일 것입니다. 이것은 메인 저장소의 주인(owner)에게 ‘나의 변경사항을 가져가 주세요’라고 요청하는 것 입니다. 이 마지막 단계가 사실상 가장 어려운 부분입니다. 만약, 버그를 수정했다면 관련된 이슈 트래킹 시스템의 번호(아이디)와 함께 수정된 사항을 간략하면서도 정확하게 작성해야 하며, 새로운 기능적 개선이나 추가가 있었다면 해당내용에 대한 설명을 명확하게 작성해야 합니다. 오픈소스 프로젝트의 리더는 수 많은 pull request를 받기 때문에 해당 내용이 불분명하거나 우선순위에서 떨어지면 쉽게 요청을 허락하지 않습니다.

[](http://guruble.com/wp-content/uploads/2014/02/ec8aa4ed81aceba6b0ec83b7_2014-02-06_ec98a4ed9b84_8-44-38.png)

[](http://guruble.com/wp-content/uploads/2014/02/ec8aa4ed81aceba6b0ec83b7_2014-02-06_ec98a4ed9b84_8-58-31.png)

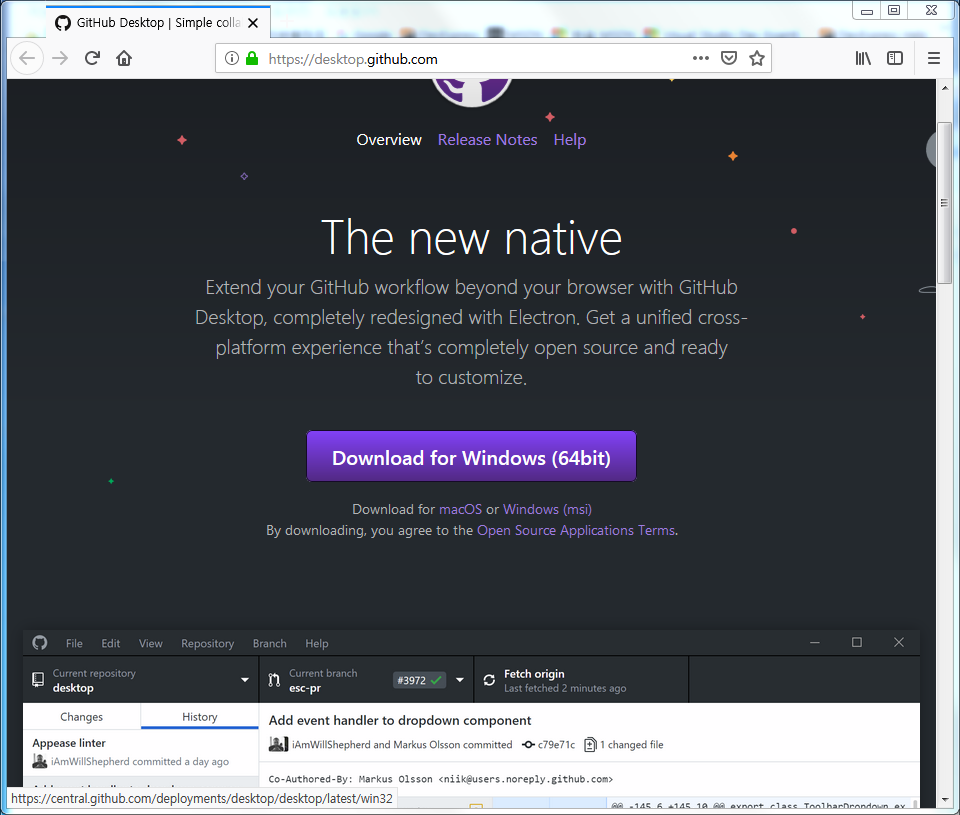
만약, 메인 저장소의 주인이 여러분의 pull request를 수용하게 되면 해당 프로젝트의 컨트리뷰터로 여러분의 이름이 오르게 되며, 이때부터 ‘나는 …. 프로젝트의 컨트리뷰터이다’라고 자랑을 하여도 크게 거짓말이 아니게 되는 것입니다.

아마도 진정한 오픈소스 프로젝트의 컨트리뷰터가 되는 길은 위의 설명처럼 순서대로 따라하면 자연스럽게 얻어지는 쉬운 일이 아니지만, 단계별로 차근차근 오랜기간 노력하게 되면 언젠가는 컨트리뷰터가 반드시 될 수 있으리라 믿습니다. 만약, 컨트리뷰터가 될 수 없다고 하여도 많은 사람들의 경험과 지혜가 담긴 오픈소스를 함께 들여다보고 참여하는 것 만으로도 개발자로서 쉽게 얻을 수 없는 많은 경험을 하게 될 것이라고 생각합니다.

# GitHub Desktop 설치하기

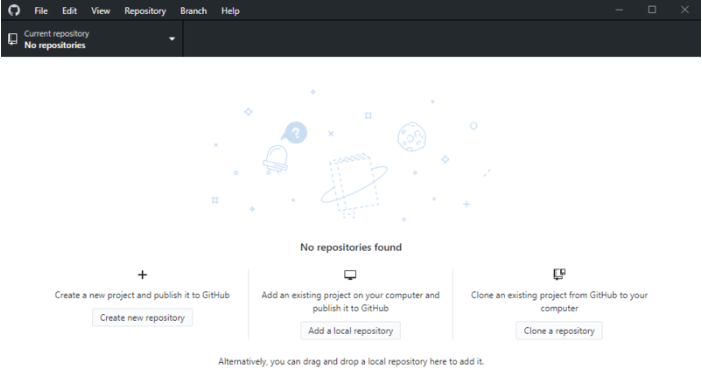
## 다운로드

 ▶ 다운로드 사이트 : <https://desktop.github.com/>

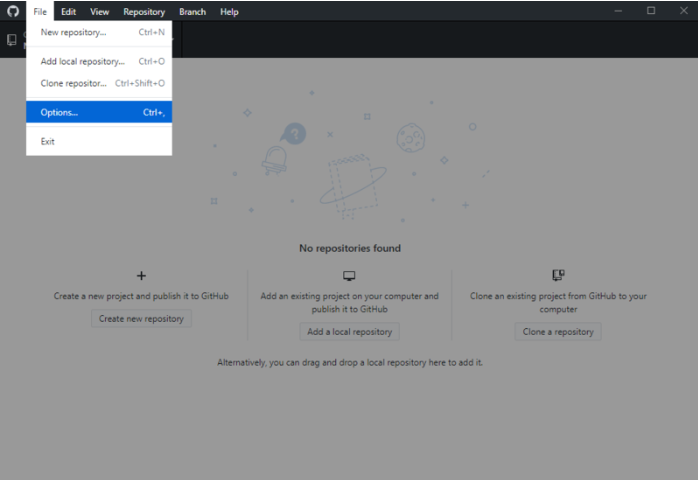


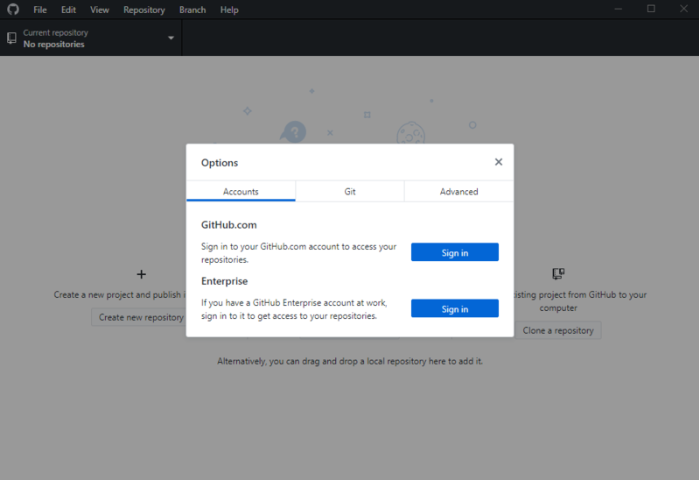
[그림 11] GitHub Desktop 다운로드

깃허브 데스크탑을 설치하셨으면 실행하여 진행합니다.

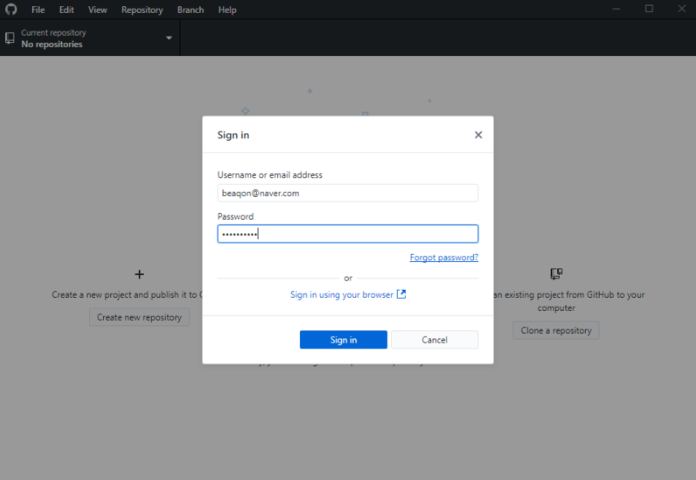


최초로 시작했을 때 로그인은 되어 있지 않습니다.





File-Option으로 이동한 후 Account 탭에서 Github.com 항목의 Sign in 버튼을 눌러 진행합니다.



아이디가 없는 경우 만드셔야 이용할 수 있습니다.

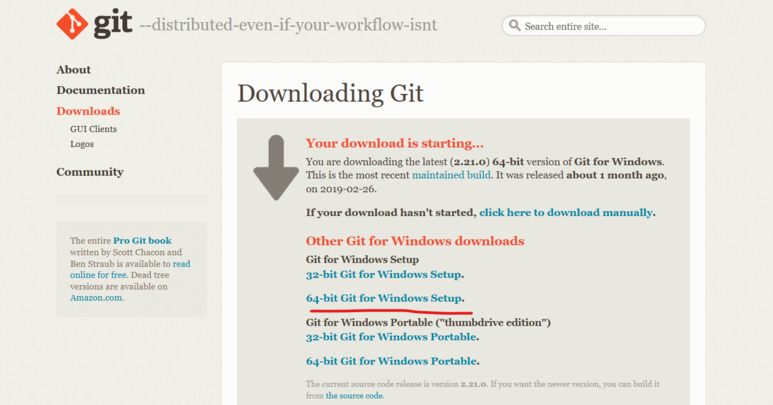
# Windows에 Git 설치

## Git 다운로드

우선 아래 링크를 통해 Git을 배포하는 사이트로 이동합니다.

<https://git-scm.com/download/win>

해당 사이트가 열리면 자동으로 다운로드를 시작하지만 다운로드가 되지 않는다면 아래 그림에 있는 빨간색 줄이 있는 링크를 선택해주세요.

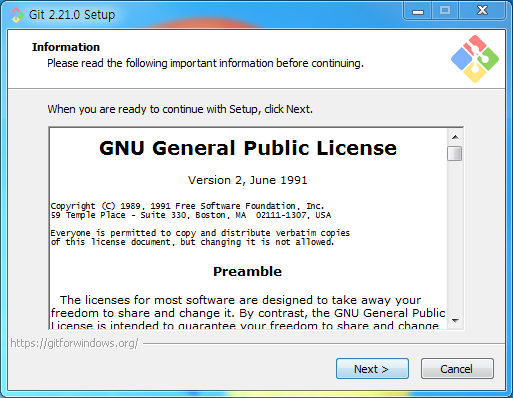
[](https://blog.naver.com/PostView.nhn?blogId=goyadb&logNo=221506934875&from=search&redirect=Log&widgetTypeCall=true&topReferer=http://search.naver.com/search.naver?sm%3Dtop_hty%26fbm%3D1%26ie%3Dutf8%26query%3Dgit%2B%EC%84%A4%EC%B9%98%26url%3Dhttps://goyadb.blog.me/221506934875%26ucs%3D4j/T1rAivl9O&directAccess=false)

​

## Git 설치

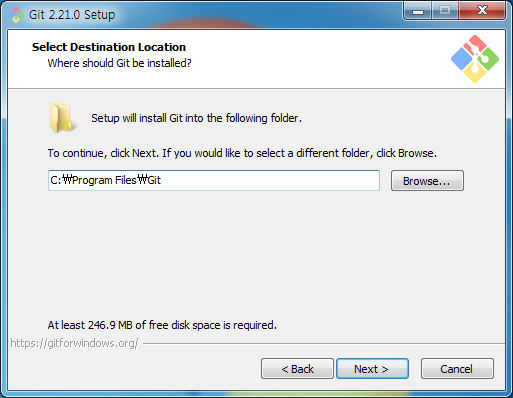
다운로드 받은 설치파일을 더블 클릭해서 실행하세요.

실행하시면 아래와 같은 화면을 보실 수 있는데, 여기에서는 "Next" 버튼을 선택하세요.



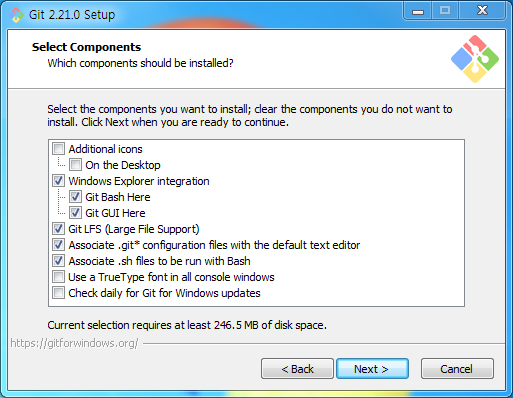
Git이 설치될 경로를 선택하는 화면입니다.

저는 기본 설정된 경로로 설치하겠습니다. "Next"를 선택하세요.



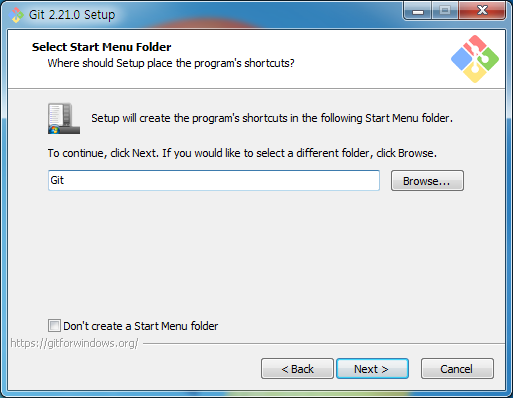
Git 설치에 필요한 구성요소를 선택하는 화면입니다.

기본 구성요소만 선택된 상태로 "Next"를 선택하세요.



다음으로는 시작 화면에서 사용할 폴더를 선택하는 화면입니다.

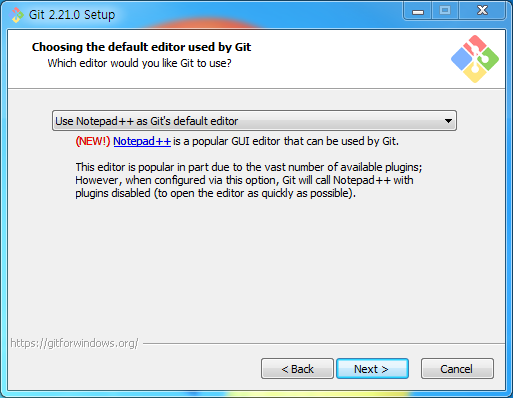
기본값 변경없이 "Next"를 선택하세요.



다음은 Git에서 사용할 에디터를 선택하는 화면입니다.

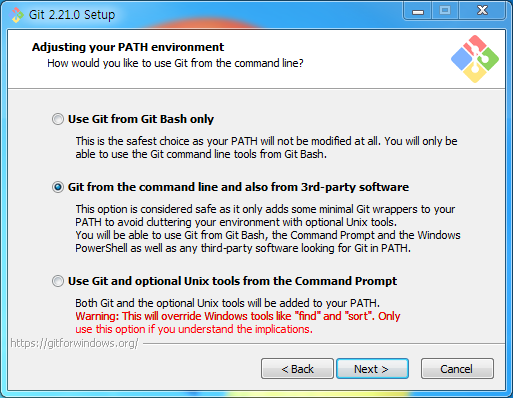
에디터란 Git은 소스 상태를 저장할 때(커밋할때) 메시지를 작성해야 합니다. 이 때 메시지 작성을 위한 에디터를 말합니다.

TextEdit, VSCode ,Sublime Text 등 다양한 에디터 중 하나를 선택할 수 있는데, 기본값은 콘솔 화면에서 Vim을 통해 편집을 하도록 설정되어 있습니다. 저는 Notepad++을 선호하기 때문에 변경 후 "Next"를 선택합니다.



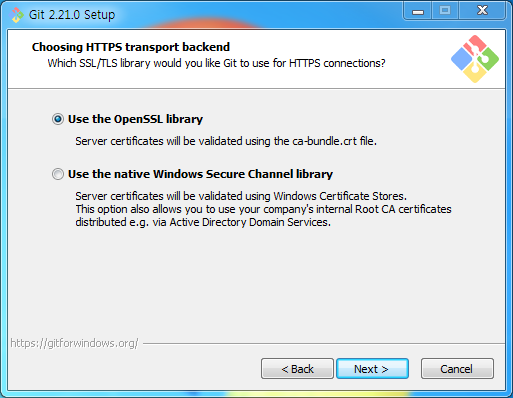
다음 화면은 Git을 콘솔 화면에서만 사용할지 여부를 선택하는 화면입니다.

저는 지금 설치하는 Git을 콘솔 화면 외에도 다른 개발툴이 함께 사용할 수 있도록 기본값을 유지한채 "Next"를 선택합니다.



다음 화면은 Git의 보안과 관련된 설정입니다.

Git으로 소스를 주고 받을때 사용자를 확인하는 과정에서 보안과 관련된 설정이 가능한데, 저는 기본값으로 OpenSSL을 선택한 상태로 "Next"를 선택합니다.

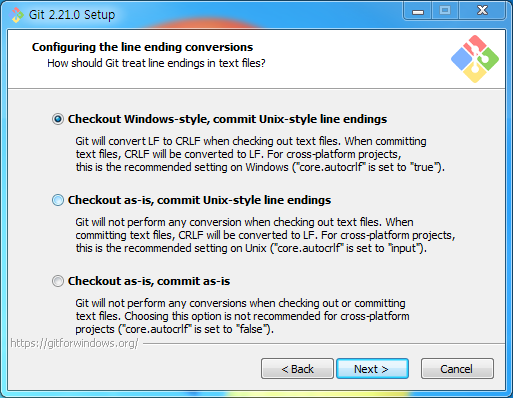


다음 화면은 소스코드의 Line ending 즉, 코드 한 줄의 끝을 어떤 방식으로 처리할지 여부를 설정하는 화면입니다.

코드 한 줄의 끝을 처리하는 방식은 운영체제에 따라 다른데, 기본값은 Checkout은 Windows 스타일, Commit은 Unix 스타일로 하도록 하였습니다.

제가 기본값을 선택한 이유는 Windows에 Git을 설치하기 때문이고, 올라간 소스를 내려받을 다른 개발자는 Windows 사용자일수도 있지만, Unix 계열의 운영체제(Linux, Unix, macOS)를 사용할수도 있기 때문입니다.

그래서 기본값을 선택한 상태로 "Next"를 선택합니다.

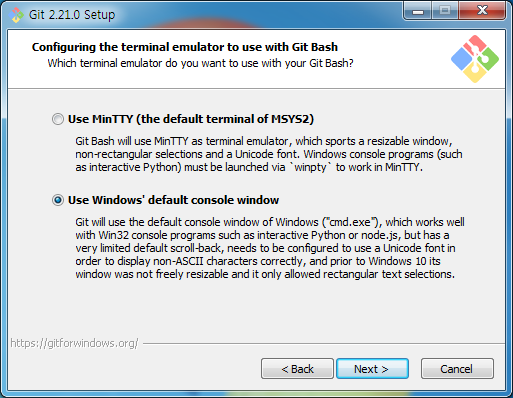


아... 무쟈게 기네요 ㅠㅠ 사실 계속 "Next"만 누르면 됩니다.

평소에는 별 생각없이 "Next"만 누르는데, 여러분들께 짧게라도 설명을 드리려고 했더니... 이렇게 길 줄은 몰랐네요.

아무튼 이번에는 콘솔 화면을 어떤 방식으로 사용하는지 결정하는 화면입니다.

Windows 기본 콘솔 윈도우를 선택합니다.

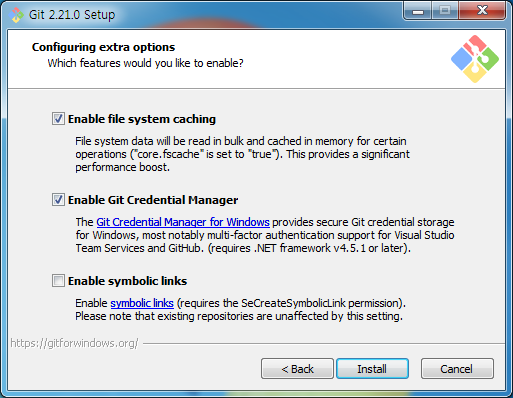


아~ 아직도 안 끝났어 ㅠㅠ

이건 추가 옵션인데, 파일 시스템 캐싱과...

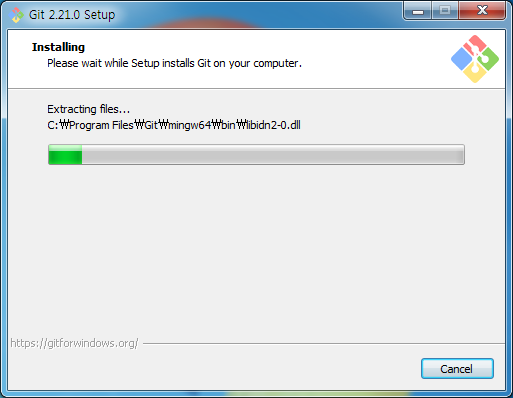
사실 이건 뭔지 잘 모르겠어요.

아무튼 여기가 끝인가 봅니다. 기본값으로 선택된 상태에서 "Install"을 선택합니다.

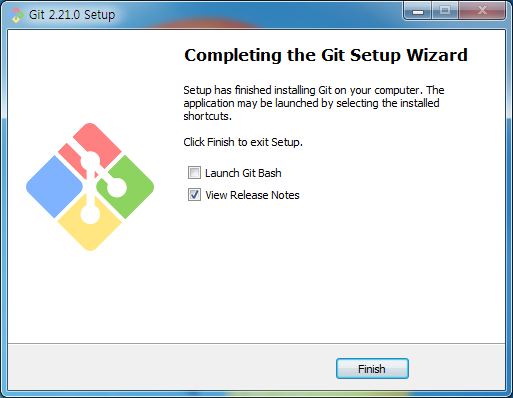


휴~ 드디어 인스톨이 시작되었네요.

이렇게 선택할 항목이 많은지 몰랐네요.



드디어 설치가 완료되었습니다.



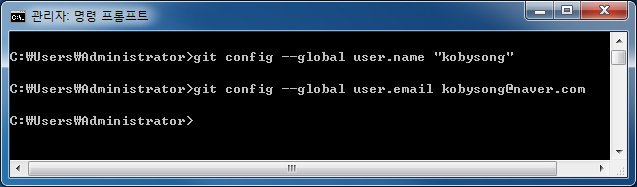
## 기본 설정

Git은 설치 후 기본 설정을 해주셔야 하는데, 당장 하지 않으셔도 상관없습니다만 나중에 Git을 사용할 때 진행이 되지 않기 때문에 지금 해주겠습니다.

​

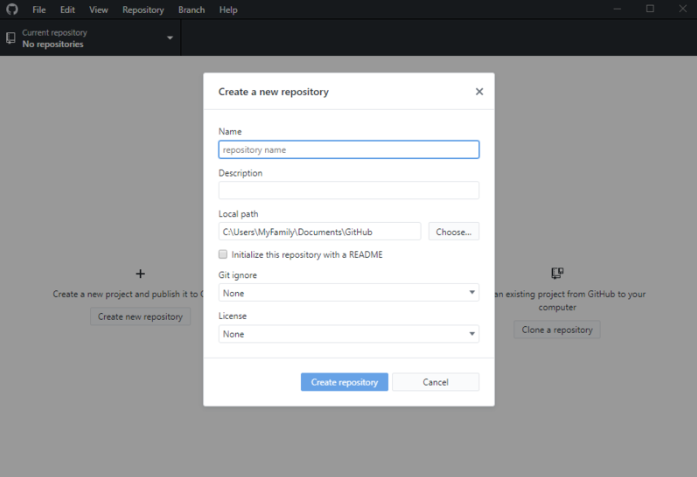
명령 프롬프트(cmd.exe)를 실행합니다.

다음과 같이 이름과 이메일 주소를 설정합니다.



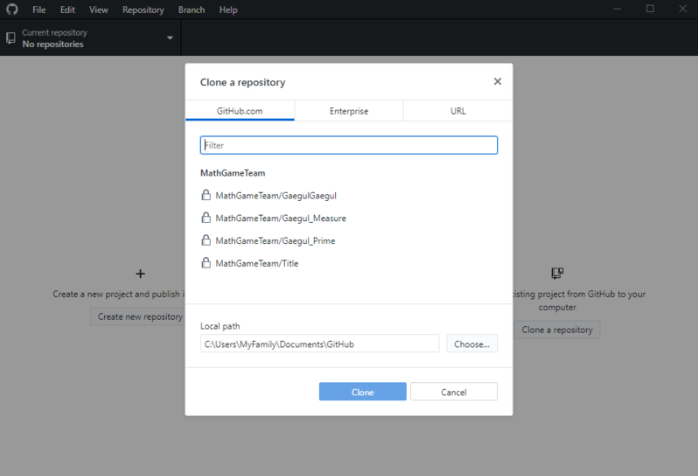
이것으로 Git 설치 과정을 마치겠습니다.

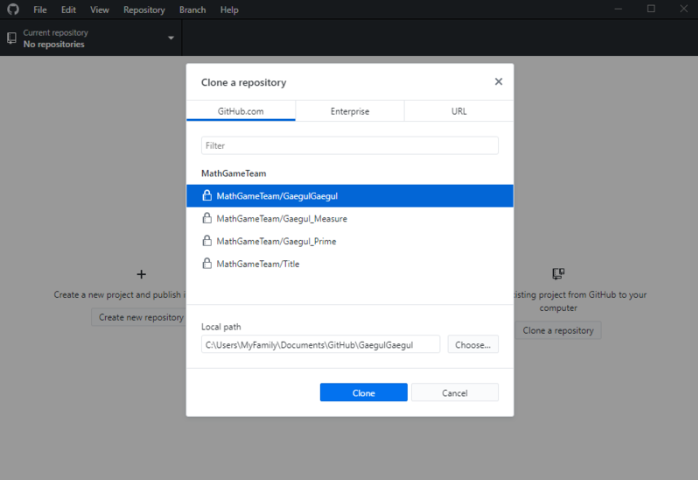
# 저장소(Repository) 생성하기



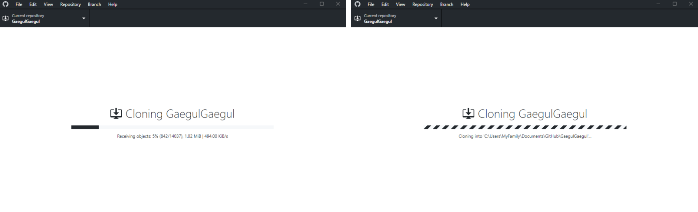
가장 첫번째 위치한 Create new repository 버튼을 누르면 저장소를 만들 수 있는 창이 생깁니다. 이 창에서 저장소의 이름, 설명, 로컬 경로와 Git ignore, 라이선스 등을 설정할 수 있는데 Git ignore은 나중에 알아보도록 하겠습니다. 저장소는 이름 그대로 코드 소스를 저장할 수 있는 원격 공간을 의미하며 유료 이용자의 경우 private, public을 선택할 수 있습니다. Create new repository 항목 옆에 위치한 항목은 add a local repository인데 이 항목은 이미 저장소가 있을 때 데스크탑 앱에 불러오는 기능입니다.

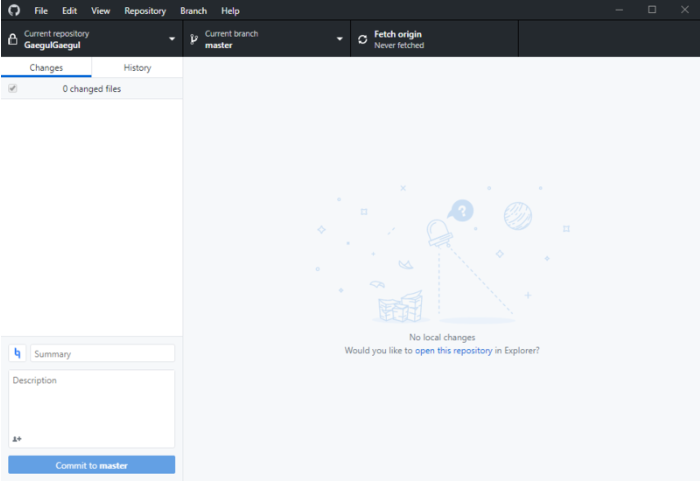
# 저장소(Repository) 불러오기



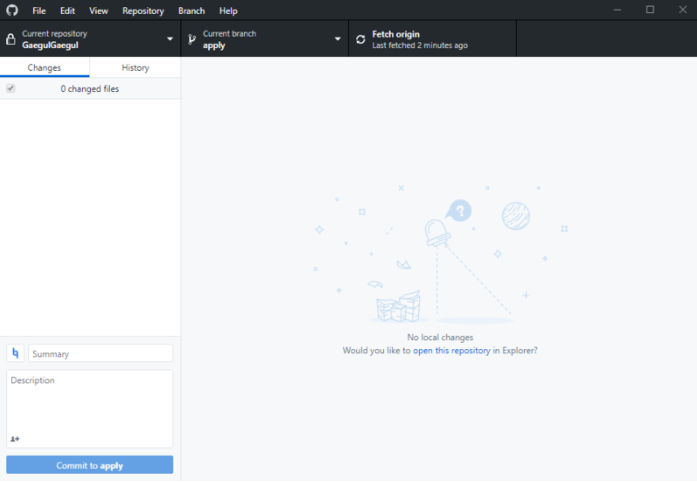


깃허브가 강점을 가지는 데에는 바로 협업을 하고 있을 때에 있습니다. 제가 개발하고 있는 외주작업을 하나 불러오겠습니다. Clone은 단어의 뜻 자체로써 복제한다는 의미를 가지고 있는데, 깃허브 서버에 업로드되어 있는 소스 파일을 로컬에 복제한다는 의미를 가지고 있습니다.

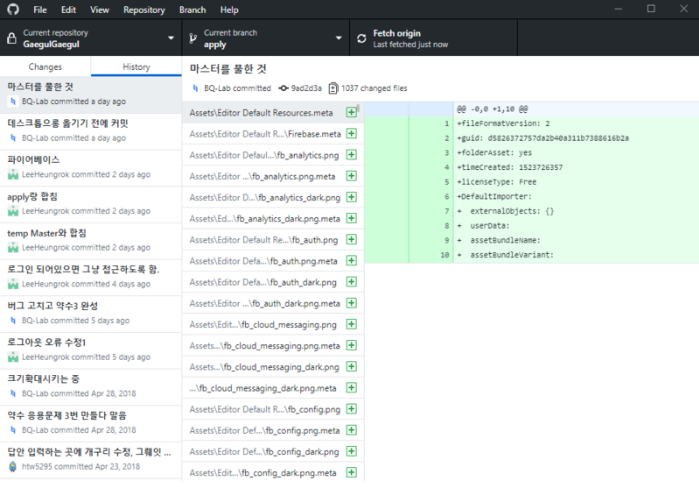




소스 파일의 규모에 따라 불러오는 시간은 달라집니다.



제 브랜치는 apply라서 apply 브랜치를 살펴보도록 하겠습니다. 아직 제가 변경한 내용이 없기 때문에 Changes 에는 아무것도 나오지 않는 겁니다. 만약에 어떤 내용을 수정했다면 변경 내용에 자동으로 보여질 것이며, Summary 부분에 커밋 이름과 디스크립션을 채워 넣어서 해당 브랜치에 커밋하면 됩니다.

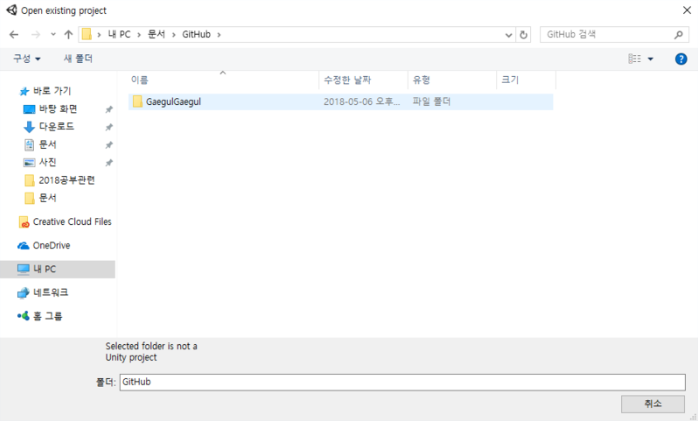


히스토리를 보았더니 커밋한 내역이 보여지네요.

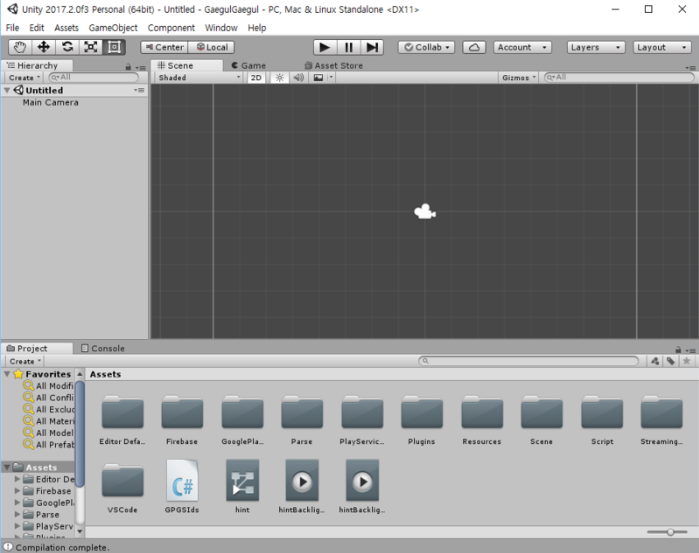
# 변경내용 커밋(Commit) 하기

테스트를 위해 아무거나 변경해보고 커밋하겠습니다.

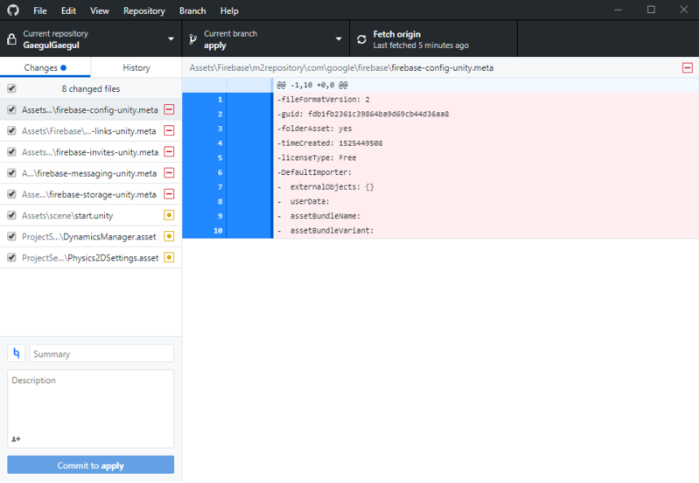
개발 환경에서 파일을 불러봅시다.



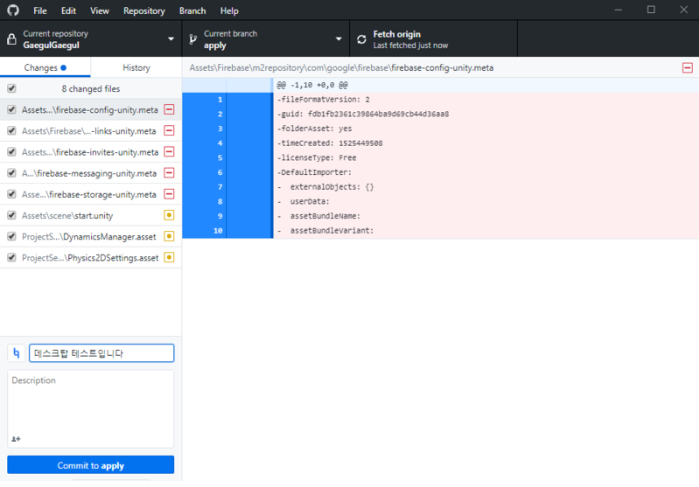
서버에서 불러온 프로젝트가 제대로 자리잡고 있음을 볼 수 있습니다. 이 폴더를 선택합니다.



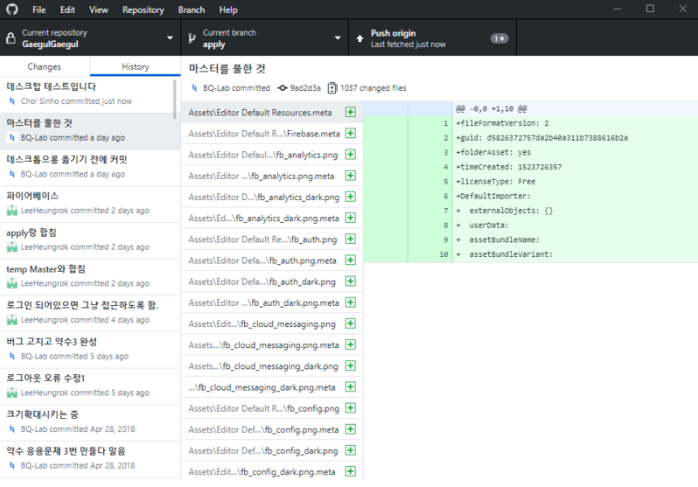
불러오는 성공했으니 메인 화면의 버튼 하나를 지워보겠습니다.



메인 화면의 버튼 하나를 지웠더니 깃허브가 바로 알아채고 변경 내역을 나타내줍니다. 이제 커밋을 해봅시다.

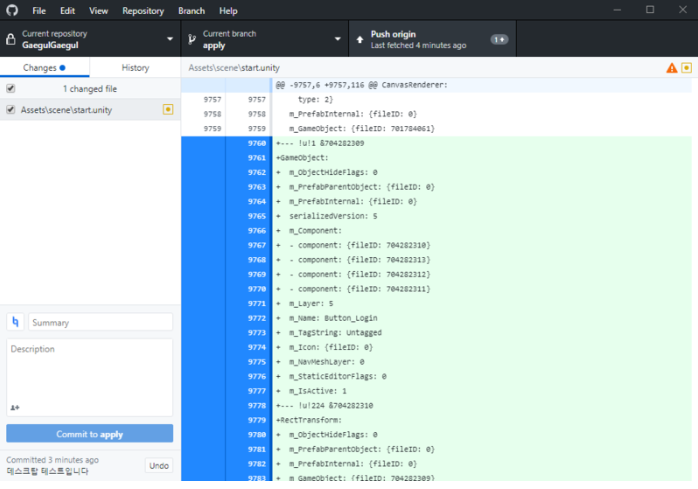


커밋 서머리를 입력하고, apply에 커밋해 볼게요.



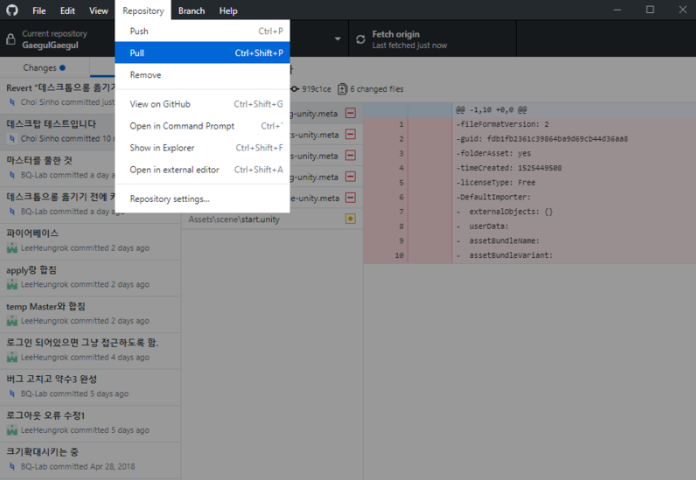
잘 됩니다.

# 커밋(Commit)한 내용을 브랜치에 Push하기



Push origin 이라고 적혀 있는 버튼을 누르기만 하면 알아서 오리진으로 보냅니다. 여기서 말하는 오리진은 서버의 주소를 가리키는 단어이며, push와 commit의 차이점이라면 커밋은 로컬에 데이터를 패키징해서 저장하는 것이고, 푸쉬는 오리진으로 전송하는 작업을 말하는 겁니다. 아마 푸쉬 안하면 불러올 수도 없을 겁니다.

# 브랜치 풀(pull)하기



Menu-Repository-Pull을 클릭합니다.

# 용어(Terms)

* **Git**

코드의 변경 사항을 추적하는 시스템으로 코드가 언제, 어떻게, 누구에 의해 변경되었는 지를 기록하는 것.

* **Github**

이런 변경 사항들을 클라우드에 올리는 웹사이트.

* **Repository**

소스코드를 저장하는 폴더. repository를 컴퓨터에 저장 하면 깃은 파일을 체크하고 변경 사항을 추적한다. 요약하면, repository는 폴더이고, 깃은 그걸 추적하는 시스템이라고 할 수 있다.

* **Commit**

예를 들면, 헤더를 만들었다고 하면 '헤더를 만들었음' 혹은 '푸터 생성' 같은 변경사항이 있을 때 이 변경사항의 내용을 기록하는 것이 commit이다. commit은 깃이 기록한 변경사항 기록함이라고 할 수 있으며, 프로젝트의 상황창과 같은 것이다.

* **Branch**

처음 디폴트로 생성되는 것은 master이다. master branch에는 사용자의 모든 commit이 반영되어 있다. 마스터 브랜치는 마지막 작업 결과물이라고 할 수 있다. 예를 들면 새로운 기능을 추가하고 싶거나 뭔가 실험적인 코드를 해보고 싶지만 마스터 브랜치는 건드리고 싶지 않을 때, 새로운 브랜치를 생성하면 마스터 브랜치와 똑같은 소스코드를 가진 브랜치를 만들어서 사용할 수 있다. 즉, 브랜치는 마스터는 건드리지 않고 새로운 기능이나 실험을 하고 싶을 때 생성한다. 브랜치에서 새로운 작업을 완료하면 마스터에 결합하여 업데이트 할 수 있다.

# 참고 사이트

* <https://m.blog.naver.com/PostView.nhn?blogId=beaqon&logNo=221269438710&proxyReferer=http%3A%2F%2Fwww.google.co.kr%2Furl%3Fsa%3Dt%26rct%3Dj%26q%3D%26esrc%3Ds%26source%3Dweb%26cd%3D1%26ved%3D2ahUKEwisgs25itThAhWTyYsBHZsZD6oQFjAAegQIBBAB%26url%3Dhttp%253A%252F%252Fm.blog.naver.com%252Fbeaqon%252F221269438710%26usg%3DAOvVaw3F3udYtBteuoK0r79lwHdn>
* <https://3jun.tistory.com/41>
* <https://git-scm.com/book/ko/v2>
* <http://guruble.com/%EC%98%A4%ED%94%88%EC%86%8C%EC%8A%A4-%ED%94%84%EB%A1%9C%EC%A0%9D%ED%8A%B8%EC%9D%98-%EC%BB%A8%ED%8A%B8%EB%A6%AC%EB%B7%B0%ED%84%B0%EB%8A%94-%EC%96%B4%EB%96%BB%EA%B2%8C-%EB%90%98%EB%8A%94-%EA%B2%83/?ckattempt=1>

# TroubleShooting

1. **에러**



**조치**

|  |
| --- |
| Had similar error: error:1407742E:SSL routines:SSL23\_GET\_SERVER\_HELLO:tlsv1 alert protocol version Just reinstalled Git and that fixed the issue.  📌 And after installation Path to Git was changed from C:\Program Files (x86)\Git\cmd\git.exe  to  C:\Program Files\Git\bin\git.exe |



|  |
| --- |
| I also received an issue similar to what is here. As wowkin2 had: error:1407742E:SSL routines:SSL23\_GET\_SERVER\_HELLO:tlsv1 alert protocol version. Happened whenever I tried to pull or push, though local commands (add, commit) functioned just fine.  I had GIT 1.9 . . . something  The solution was simply to uninstall GIT and reinstall the new version. I did not do anything else. |