2019-09-02

**Pro Git //** [**https://git-scm.com/book/ko/v2**](https://git-scm.com/book/ko/v2)

1. **시작하기**

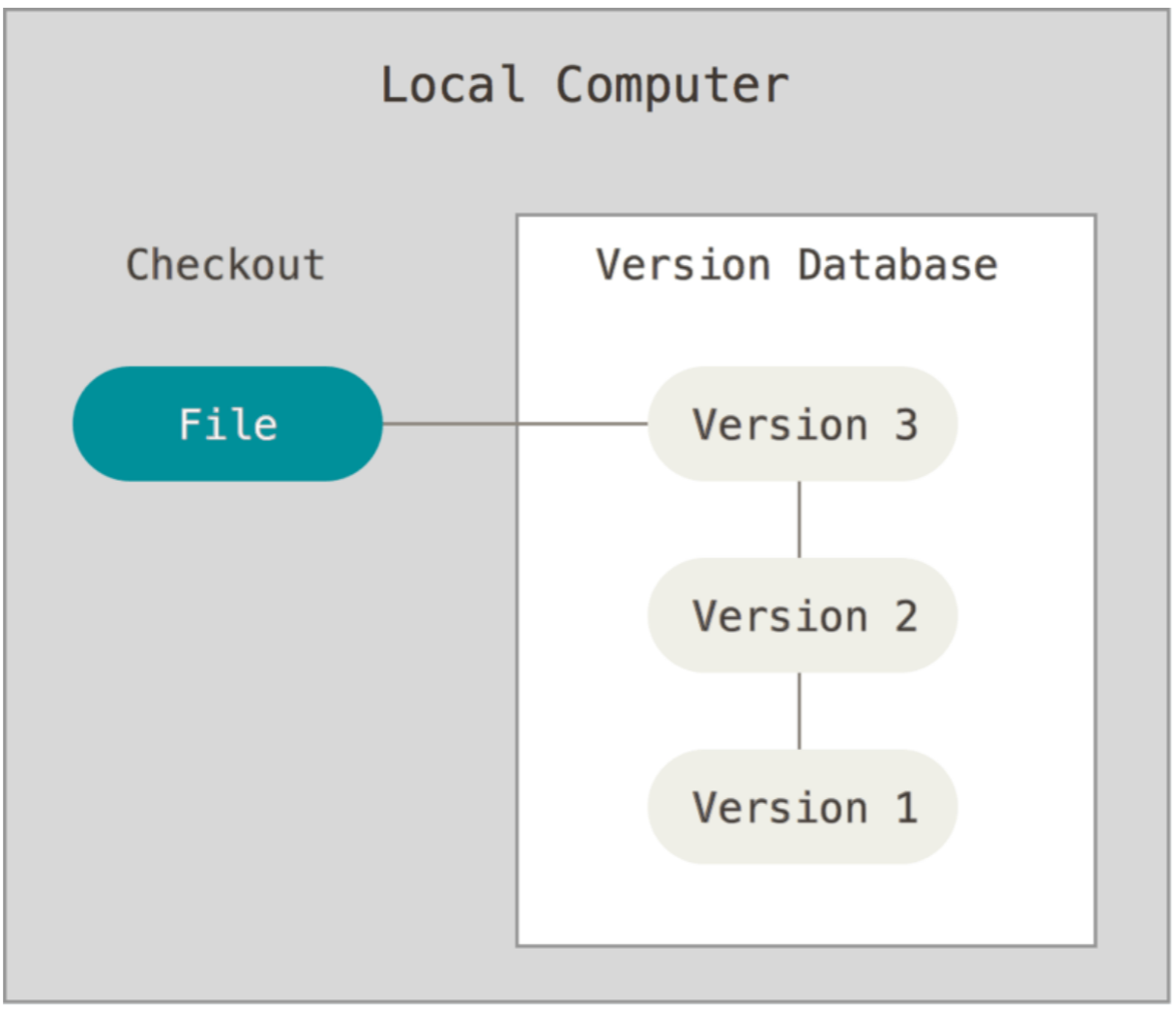
**1. 관리 시스템 (Version Control System)**

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

**버전 관리**(version control, revision control), 소스 관리(source control), 소스 코드 관리(source code management, SCM)란 동일한 정보에 대한 여러 버전을 관리하는 것을 말한다. [공학](https://ko.wikipedia.org/wiki/%EA%B3%B5%ED%95%99)과 [소프트웨어 개발](https://ko.wikipedia.org/wiki/%EC%86%8C%ED%94%84%ED%8A%B8%EC%9B%A8%EC%96%B4_%EA%B0%9C%EB%B0%9C)에서 팀 단위로 개발 중인 [소스 코드](https://ko.wikipedia.org/wiki/%EC%86%8C%EC%8A%A4_%EC%BD%94%EB%93%9C)나, [청사진](https://ko.wikipedia.org/wiki/%EC%B2%AD%EC%82%AC%EC%A7%84) 같은 설계도 등의 디지털 문서를 관리하는데 사용된다. "버전"을 통해서 시간적으로 변경 사항과 그 변경 사항을 작성한 작업자를 추적할 수 있다.

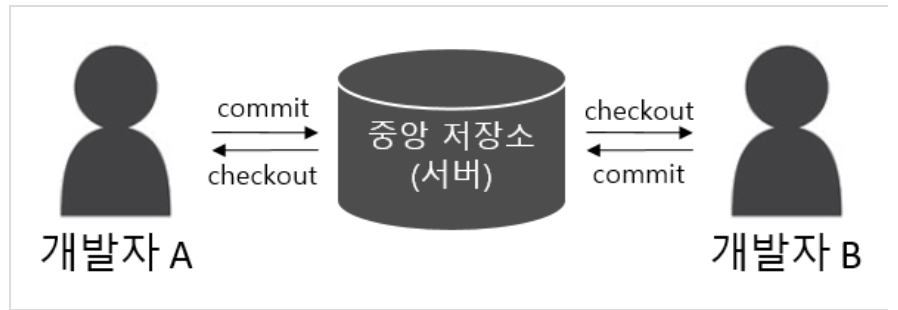
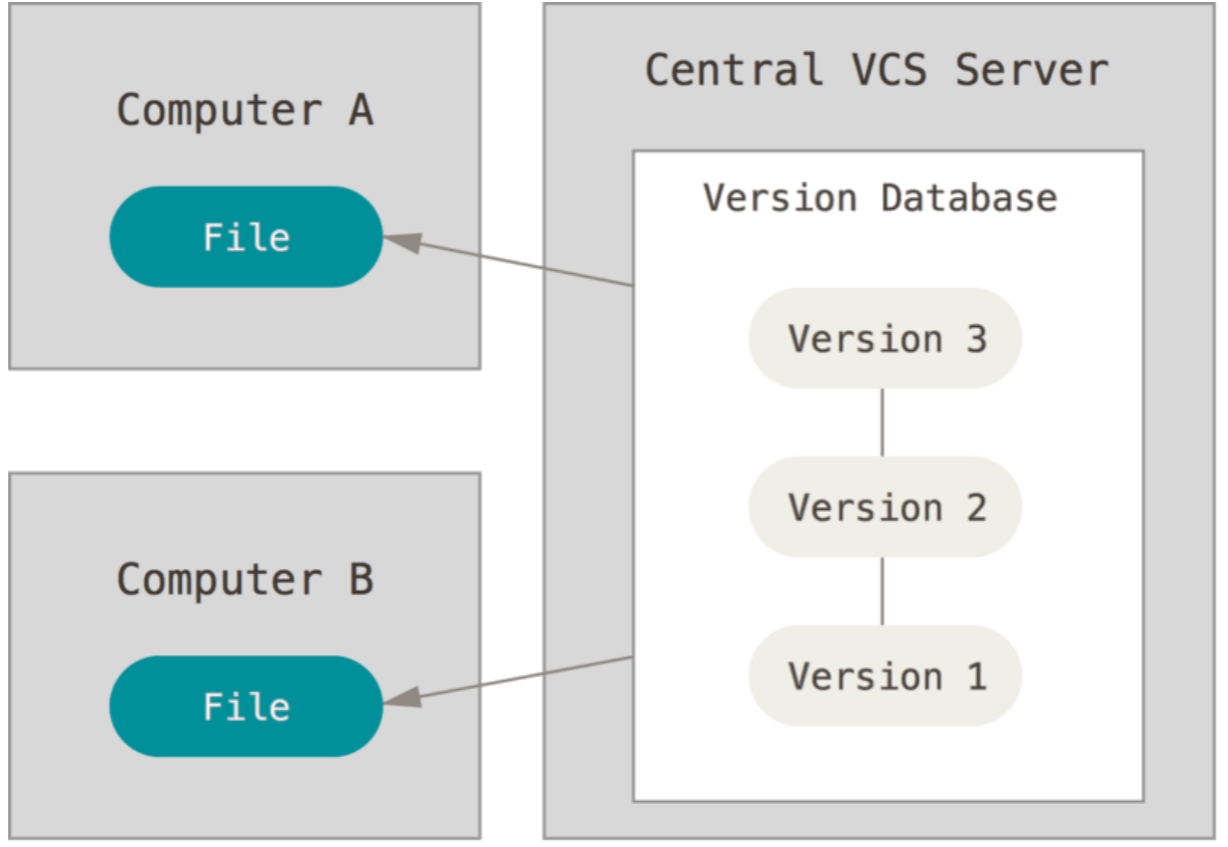
**- 로컬 버전 관리 시스템**

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



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

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



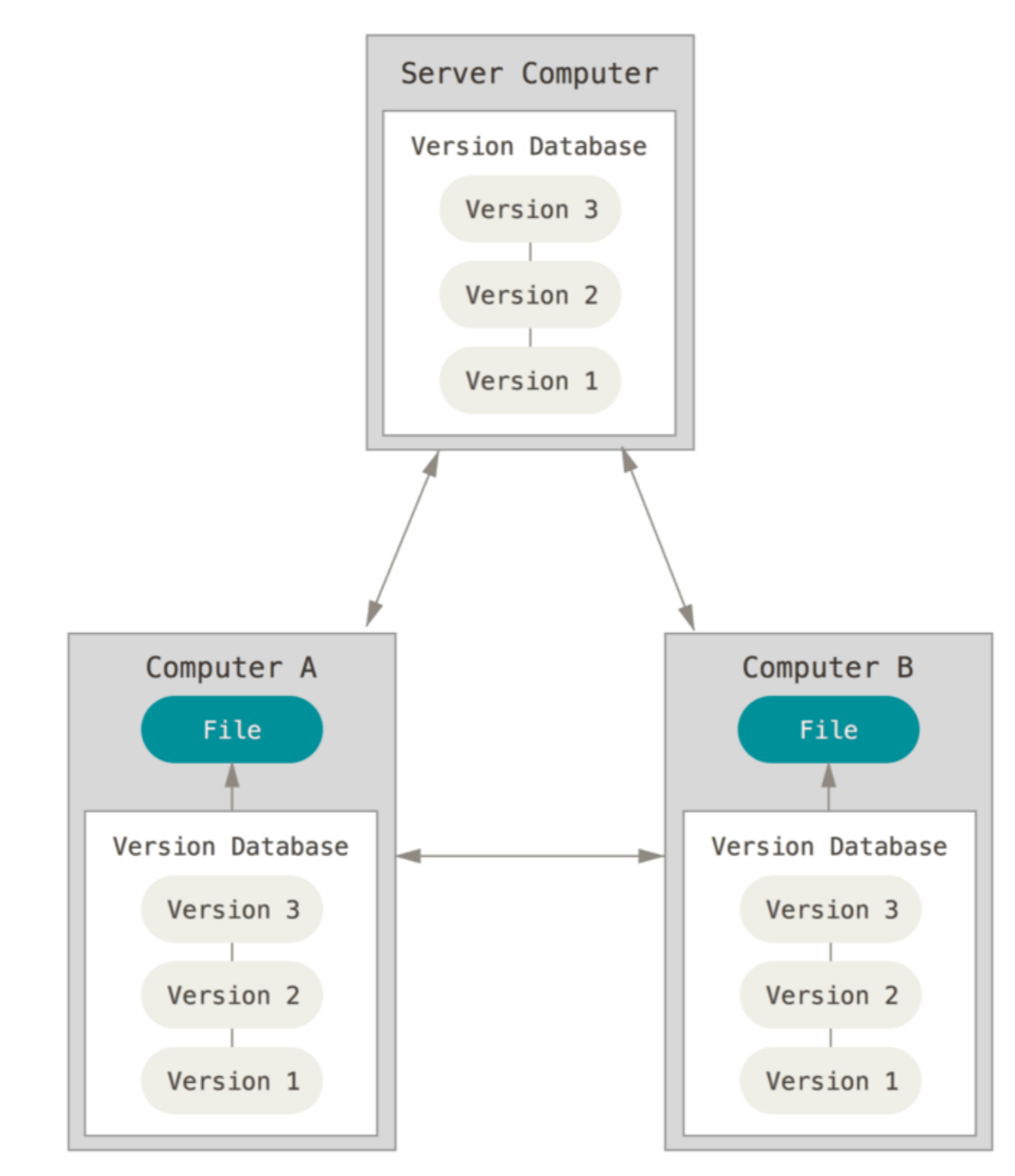
<https://gangju.tistory.com/12>

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

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

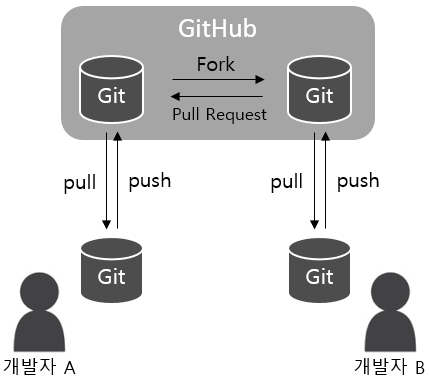
**- 분산 버전 관리 시스템(Distributed Version Control System; DVCS)**

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



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

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

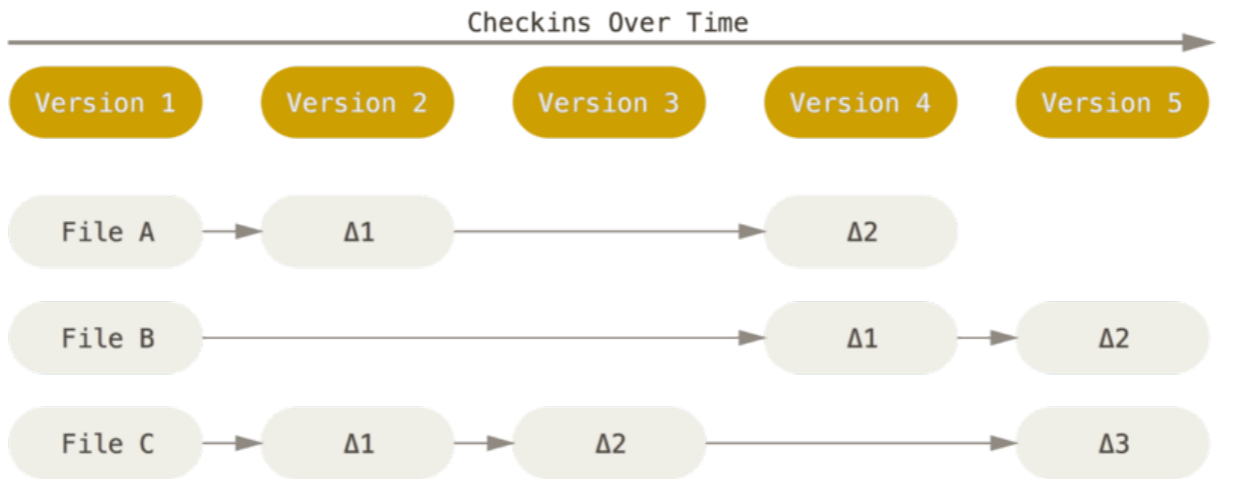


**2. Git 기초**

**Git의 핵심**

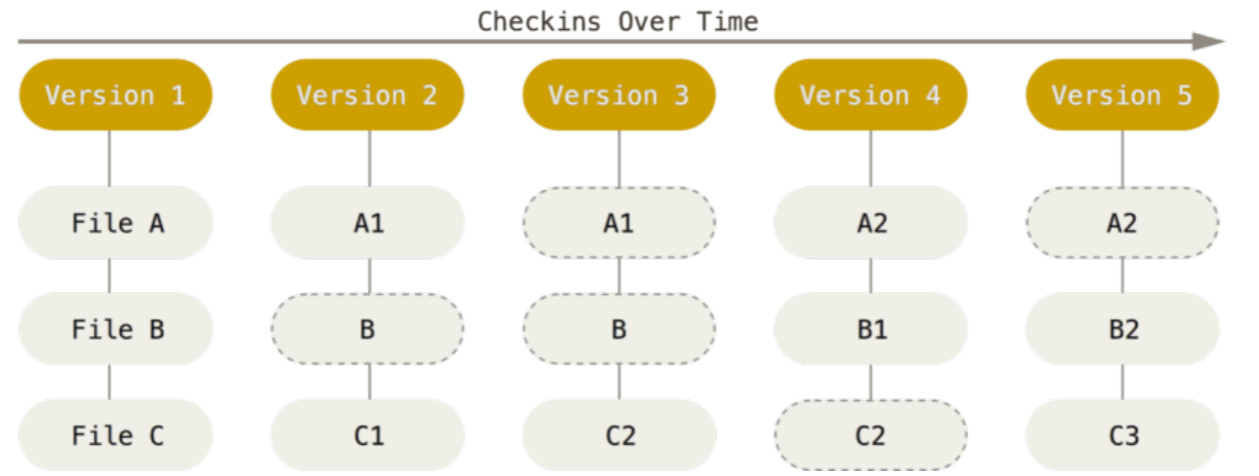
**1. 차이(델타)가 아닌 스냅샷**

- 대부분의 VCS 시스템 (파일의 차이를 관리)



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

- Git(스냅샷을 관리)



Git은 데이터를 파일 시스템 스냅샷으로 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.

**2. 거의 모든 명령을 로컬에서 실행**

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

**3. 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

실제로 Git은 파일을 이름으로 저장하지 않고 해당 파일의 해시로 저장하여 식별한다.

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

**체크섬**(checksum)은 [중복 검사](https://ko.wikipedia.org/wiki/%EC%98%A4%EB%A5%98_%EC%A0%95%EC%A0%95_%EB%B6%80%ED%98%B8)의 한 형태로, [오류 정정](https://ko.wikipedia.org/wiki/%EC%98%A4%EB%A5%98_%EA%B2%80%EC%B6%9C_%EB%B6%80%ED%98%B8)을 통해, 공간([전자 통신](https://ko.wikipedia.org/wiki/%EC%A0%84%EC%9E%90_%ED%86%B5%EC%8B%A0))이나 시간([기억 장치](https://ko.wikipedia.org/wiki/%EA%B8%B0%EC%96%B5_%EC%9E%A5%EC%B9%98)) 속에서 송신된 자료의 무결성을 보호하는 단순한 방법이다.

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

**SHA**(Secure Hash Algorithm, 안전한 해시 알고리즘) 함수들은 서로 관련된 [암호학적 해시 함수](https://ko.wikipedia.org/wiki/%EC%95%94%ED%98%B8%ED%95%99%EC%A0%81_%ED%95%B4%EC%8B%9C_%ED%95%A8%EC%88%98)들의 모음이다.

SHA-1은 SHA 함수들 중 가장 많이 쓰이며, [TLS](https://ko.wikipedia.org/wiki/%ED%8A%B8%EB%9E%9C%EC%8A%A4%ED%8F%AC%ED%8A%B8_%EB%A0%88%EC%9D%B4%EC%96%B4_%EB%B3%B4%EC%95%88), [SSL](https://ko.wikipedia.org/wiki/SSL), [PGP](https://ko.wikipedia.org/wiki/PGP_(%EC%86%8C%ED%94%84%ED%8A%B8%EC%9B%A8%EC%96%B4)), [SSH](https://ko.wikipedia.org/wiki/%EC%8B%9C%ED%81%90%EC%96%B4_%EC%85%B8), [IPSec](https://ko.wikipedia.org/wiki/IPSec" \o "IPSec) 등 많은 보안 프로토콜과 프로그램에서 사용되고 있다.

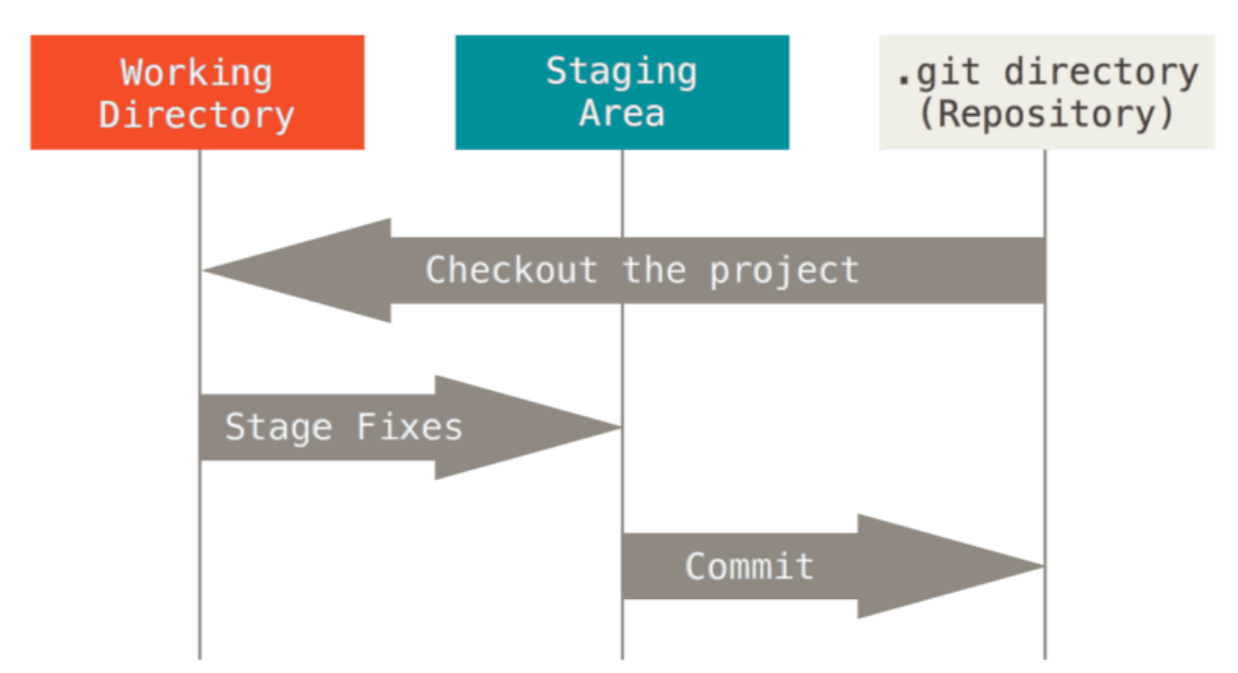
SHA-1은 SHA-0의 압축 함수에 비트 회전 연산을 하나 추가한 것이다.

**4. Git은 데이터를 추가할 뿐**

Git으로 무얼 하든 데이터를 추가한다. 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. 9장을 보면 Git이 데이터를 어떻게 저장하고 손실을 어떻게 복구해야 할지 알 수 있다.

**5. 세 가지 상태**



Git파일 관리 상태

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

- Modified : 수정한 파일을 아직 로컬 데이터베이스에 커밋하지 않음

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

Git 프로젝트의 세 가지 단계

- Git directory : Git이 프로젝트의 메타데이터와 객체 데이터베이스를 저장하는 곳. 다른 컴퓨터에 있는 저장소를 Clone할 때 Git 디렉토리가 만들어짐.

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

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

**정리**

1. 워킹 디렉토리에서 파일을 수정한다.

2. Staging Area에 파일을 Stage해서 커밋할 스냅샷을 만든다.

3. Staging Area에 있는 파일들을 커밋해서 Git 디렉토리에 영구적인 스냅샷으로 저장한다.

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

1. **Git의 기초**
2. **Git 저장소 만들기**

**- 기존 디렉토리를 Git 저장소로 만들기**

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

$ git init

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

$ git add \*.c

$ git add README

$ git commit -m 'initial project version'

**- 다른 서버에 있는 저장소를 Clone**

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

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

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

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

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

1. **수정하고 저장소에 저장하기**

- **워킹 디렉토리 내 파일**

**Tracked**

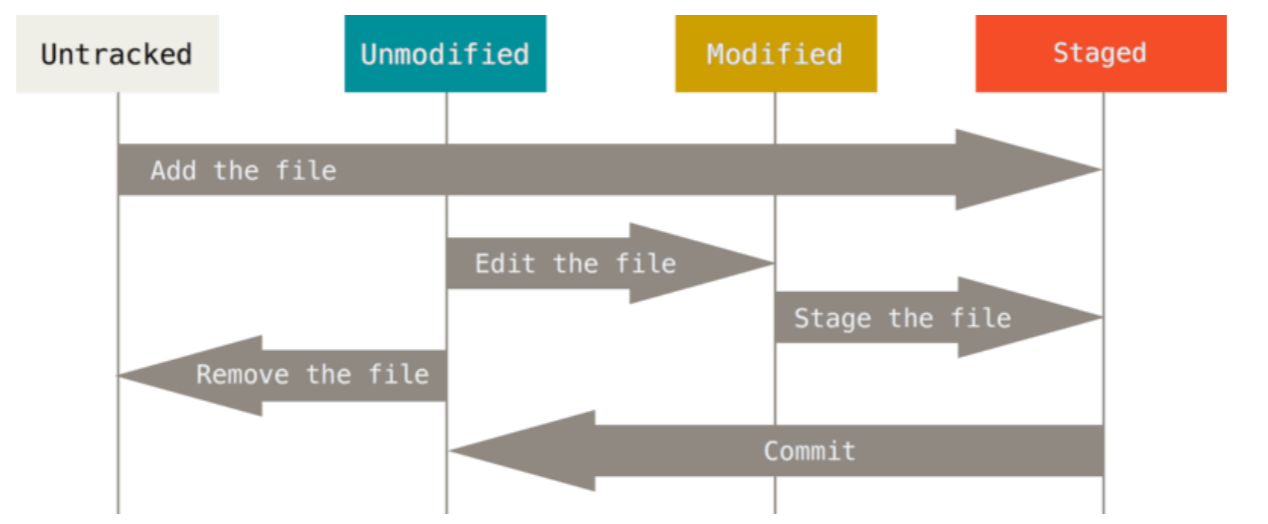
-Unmodified

-Modified

-Staged

**Untracked**

**- 파일의 라이프사이클**



**- 파일의 상태 확인하기 : git status 명령**

Clone 이후 **git status** 명령 실행

$ git status

On branch master

nothing to commit, working directory clean

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

🡪 Untracked 파일은 아직 없어서 목록에 나타나지 않음

🡪 현재 브랜치를 알려줌

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

$ vim README

$ git status

On branch master

Untracked files:

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

README

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

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

**- 파일을 새로 추적하기**

**git add** 명령으로 파일을 새로 추적 가능

$ git add README

**git status** 명령을 다시 실행시 README파일이 Tracked 상태이면서 Staged 상태인 것 확인 가능

$ git status

On branch master

Changes to be committed:

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

new file: README

- “Changes to be committed” : Staged 상태라는 것을 의미한다.

**- Modified 상태의 파일을 stage하기**

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

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

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

**- 파일 상태를 짤막하게 확인하기**

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

$ git status -s

M README

MM Rakefile

A lib/git.rb

M lib/simplegit.rb

?? LICENSE.txt

- ?? : Untracked

- A : Staged 상태로 추가한 파일 중 새로 생성한 파일

- M : 수정한 파일

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

M README

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

M lib/simplegit.rb

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

MM Rakefile

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

**- 파일 무시하기**

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

$ cat .gitignore

\*.[oa]

\*~

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

**- Staged와 Unstaged 상태의 변경 내용을 보기**

- **git status** : Staged파일 / Unstaged파일 구분

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

**- git diff** : 수정했지만 아직 staged 상태가 아닌 파일을 비교해 볼 수 있다

$ git diff

diff --git a/benchmarks.rb b/benchmarks.rb

index 3cb747f..da65585 100644

--- a/benchmarks.rb

+++ b/benchmarks.rb

@@ -36,6 +36,10 @@ def main

@commit.parents[0].parents[0].parents[0]

end

+ run\_code(x, 'commits 1') do

+ git.commits.size

+ end

+

run\_code(x, 'commits 2') do

log = git.commits('master', 15)

log.size

: 워킹 디렉토리에 있는 것과 Staging Area에 있는 것을 비교한다. 그래서 수정한 후 아직 Stage하지 않은 것을 보여준다.

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

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

$ git diff --cached

diff --git a/README b/README

new file mode 100644

index 0000000..03902a1

--- /dev/null

+++ b/README2

@@ -0,0 +1,5 @@

+grit

+ by Tom Preston-Werner, Chris Wanstrath

+ http://github.com/mojombo/grit

+

+Grit is a Ruby library for extracting information from a Git repository

1. 커밋 히스토리 조회하기
2. 되돌리기
3. 리모트 저장소
4. 태그
5. Git Alias