pro git 정리

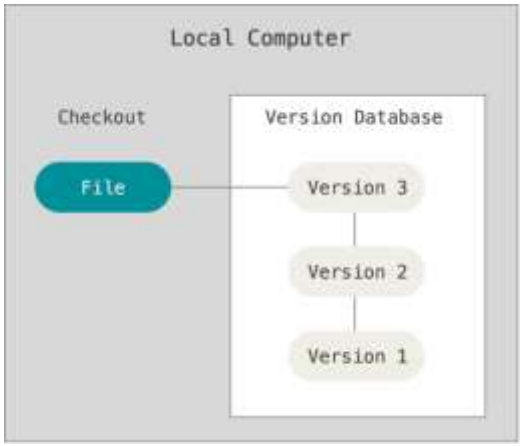
1. 시작하기
2. Git의 기초
3. Git 브랜치
4.

1.1 버전관리

1 버전 관리 시스템(VCS - Version Control System)

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 복구

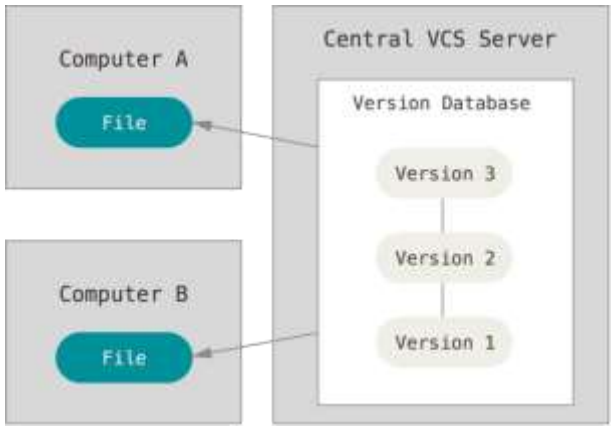
RCS(Revision Control System) ₩
– patch set(파일에서 변경되는 부분) 관리



2. 중앙집중식 버전 관리(CVCS)

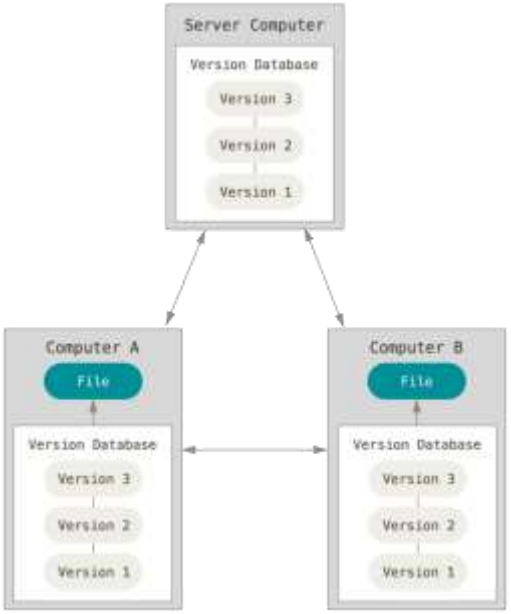
-중앙 서버에서 파일을 받아서 checkout(사용) 모두 누가 무엇을 하고 있는지 알 수 있다. 모든 클라이언트의 로컬 데이터베이스를 관리하는 것보다 VCS 하나를 관리하기가 쉽다.

결점 :
중앙 서버에 발생한 문제가 모두에게 영향
중앙 데이터베이스가 있는 하드디스크에 문제가 생기면 프로젝트의 모든 히스토리를 잃는다. (각 사람의 스냅샷은 유지)
로컬 VCS 시스템도 이와 비슷한 결점.



3 DVCS(분산 버전 관리 시스템)

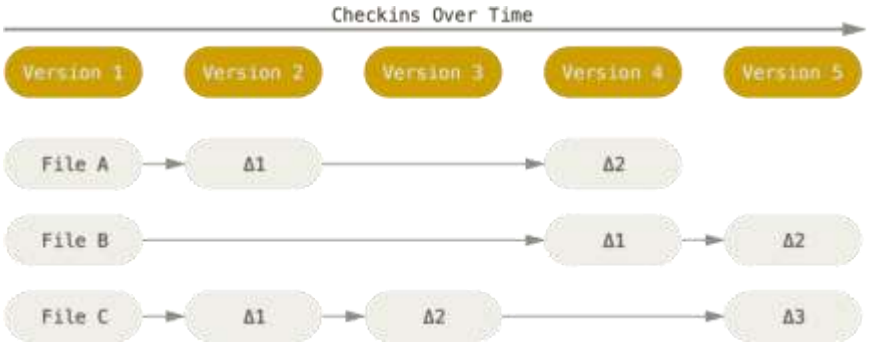
저장소를 히스토리와 더불어 전부 복제해서 사용.
서버에 문제가 생기면 각 클라이언트를 이용해 서버를 복원할 수 있다. Clone은 모든 데이터를 가진 진정한 백업.
게다가 대부분의 DVCS 환경에서는 리모트 저장소가 존재
다양한 그룹과 다양한 방법으로 협업이 가능



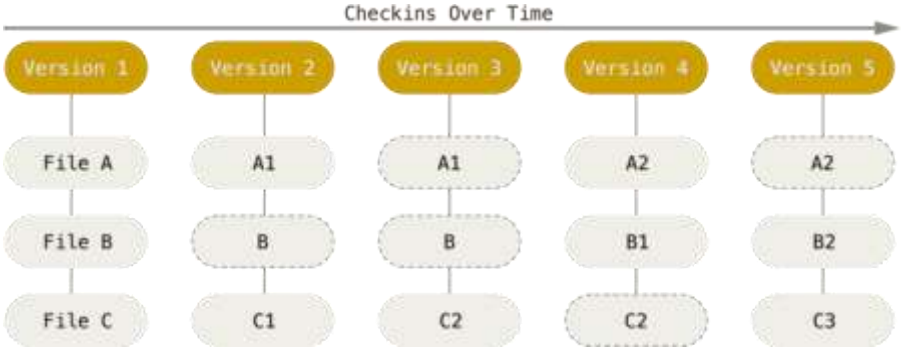
1.3 Git 기초

git의 핵심

VCS 시스템 대부분 파일의 목록을 관리.
CVS, Subversion, Perforce, Bazaar 등의 시스템은 각 파일의 변화를 시간순으로 관리하면서 파일들의 집합을 관리한다(보통 **델타 기반** 버전관리 시스템)
-- 각 파일에 대한 **변화를 저장**하는 시스템들. (patch set)



Git은 커밋하거나 프로젝트의 상태를 저장할 때마다 파일이 존재하는 그 순간을 중요하게 여긴다. 파일이 달라지지 않았으면 Git은 성능을 위해서 파일을 새로 저장하지 않고 이전 상태의 파일에 대한 링크만 저장.
Git은 데이터를 파일 시스템 **스냅샷의 스트림(연속)**처럼 취급한다 -- 시간 순으로 프로젝트의 **스냅샷**을 저장



1.3 Git 기초

git의 장점

1 로컬에서 거의 모든 명령을 로컬 파일과 데이터만 사용 – 빠름
  - 히스토리 조회시 서버 없이 조회 가능, 한달 전 파일 히스토리도 로컬에서 찾음
2 서버 접근이 필요하지 않기 때문에 오프라인 상태에서도 커밋가능

git의 무결성

1 데이터를 저장하기 전, 파일의 내용이나 디렉토리 구조를 이용해 **체크섬**을 구하고 그 체크섬으로 데이터를 관리
체크섬 : 40자 길이의 16진수 문자열. SHA-1 해시를 사용 (예 : 24b9da6552252987aa493b52f8696cd6d3b00373)
체크섬을 이해하는 Git 없이는 어떠한 파일이나 디렉토리도 변경할 수 없다.

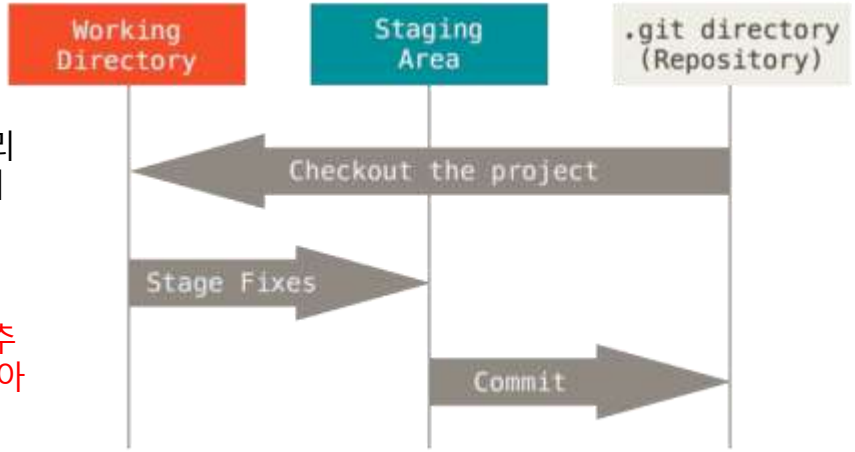
2 Git으로 무얼 하든 Git 데이터베이스에 데이터가 **추가** 된다. 되돌리거나 데이터를 삭제할 방법이 없다. 일단 스냅샷을 커밋하고 나면 데이터를 잃어버리기 어렵다.

Git을 사용하는 방법
- CLI → git의 모든 기능 지원
- GUI

1.3 Git의 기초

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

- **git 프로젝트의 세단계**



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

Staging Area : Git 디렉토리에 있다. 단순한 파일이고 곧 커밋할 파일에 대한 정보를 저장한다. 파일을 수정하고 Staging Area에 추가했다면 Staged 상태/ 그리고 Checkout 하고 나서 수정했지만, 아직 Staging Area에 추가하지 않았으면 Modified
(Git에서는 기술용어로는 "Index" 라고 하지만, "Staging Area")

Git 디렉토리 - Git이 프로젝트의 메타데이터와 객체 데이터베이스를 저장하는 곳 이 Git 디렉토리가 Git의 핵심이다. 다른 컴퓨터에 있는 저장소를 **Clone** 할 때 Git 디렉토리가 만들어진다. Git 디렉토리에 있는 파일들은 Committed 상태

1.5 git 설치

파일로 설치(프로젝트, 패키지) 로 git Bash, git GUI, Shell Integration 설치 가능

소스코드로 설치하기- 가장 최신 버전 설치가능
Git을 설치하려면 Git이 의존하고 있는 라이브러리인 autotools, curl, zlib, openssl, expat, libiconv등이 필요
문서를 다양한(doc, html, info) 형식으로 추가하려면 패키지들이 추가로 필요
(실행파일 이름이 다 다름 - docbook2X 패키지)

최신 배포 버전을 가져오기 , 컴파일, 설치 순서

+ Git BASH
Git for Windows provides a BASH emulation used to run Git from the command line. *NIX users should feel right at home, as the BASH emulation behaves just like the "git" command in LINUX and UNIX environments.
Git GUI
As Windows users commonly expect graphical user interfaces, Git for Windows also provides the Git GUI, a powerful alternative to Git BASH, offering a graphical version of just about every Git command line function, as well as comprehensive visual diff tools.
Shell Integration
Simply right-click on a folder in Windows Explorer to access the BASH or GUI.

1.6 git 최초 설정

git 최초 설정

/etc/gitconfig 파일:
시스템의 **모든 사용자와 모든 저장소**에 적용되는 설정이다.
git config --system 옵션으로 이 파일을 읽고 쓸 수 있다. (이 파일은 시스템 전체 설정파일이기 때문에 수정하려면 시스템의 관리자 권한이 필요하다.)

~/.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.

.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.)
Windows에서는 $HOME 디렉토리에서 .gitconfig 파일을 찾는다. /etc/gitconfig 파일은 그 경로에서 찾는다. (MSys 루트의 상대경로 – 설치시 결정됨)

1 Git을 설치하고 나서 가장 먼저 해야 하는 것은 사용자이름과 이메일 주소를 설정,
Git은 커밋할 때마다 이 정보를 사용한다. 한 번 커밋한 후에는 정보를 변경할 수 없다. (만약 프로젝트마다 다른 이름과 이메일 주소를 사용하고 싶으면 --global 옵션을 빼고 명령을 실행한다.)

2 사용자 정보를 설정하고 나면 Git에서 사용할 텍스트 편집기를 고른다

3 설정 확인 git config --list 명령을 실행하면 설정한 모든 것을 보여주어 바로 확인할 수 있다.
명령어에 대한 도움말이 필요할 때 도움말을 보는 방법은 두 가지로 동일한 결과를 볼 수 있다.
$ git help <verb> $ man git-<verb> → <verb> 예) config
간략히 살펴볼수도 있다. -h, --help 옵션을 사용

2.1 Git 저장소 만들기

1 기존 디렉토리를 Git 저장소로 만들기

$ cd /c/user/my_project
$ git init : 이 명령은 .git 이라는 하위 디렉토리를 만든다. .git 디렉토리에는 저장소에 필요한 뼈대 파일(Skeleton)이 들어 있다.
git add . (폴더로 옮긴 후, 모든 파일)
git commit -m "first commit"
git remote add origin https://github.com/yungity/repository이름.git
git push -u origin master

2 기존 저장소를 Clone 하기

- 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.
- git clone <url> 명령으로 저장소를 Clone [url은 https, git, ssh 프로토콜 등 사용가능]

$ git clone https://github.com/libgit2/libgit2
이 명령은 "libgit2" 라는 디렉토리를 만들고 그 안에 .git 디렉토리를 만든 후 저장소의 데이터를 모두 가져와서 자동으로 가장 최신 버전을 Checkout

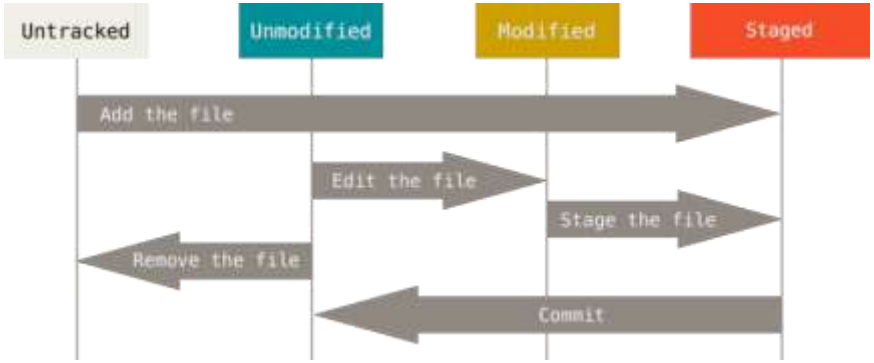
--- 디렉토리 이름을 지정하려면(여기에선 mylibgit)
$ git clone https://github.com/libgit2/libgit2 mylibgit

2.2 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 디렉토리: Tracked(관리대상임)와 Untracked(관리대상이 아님)

-- Tracked 파일 :
이미 스냅샷에 포함돼 있던 파일 git이 알고있는 파일
-Unmodified(수정하지 않음)와 Modified(수정함) 그리고 Staged(커밋으로 저장소에 기록할) 상태 중 하나
마지막 커밋 이후 아직 아무것도 수정하지 않은 상태Unmodified에서 어떤 파일을 수정하면 Git은 그 파일을 **Modified** 상태로 인식한다. 실제로 커밋을 하기 위해서는 이 수정한 파일을 Staged 상태로 만들고, Staged 상태의 파일을 커밋



-- Untracked 파일:
워킹 디렉토리에 있는 파일 중 스냅샷에도 Staging Area에도 포함되지 않은 파일 , 파일이 Tracked 상태가 되기 전까지는 Git은 절대 그 파일을 커밋하지 않는다.

파일의 상태 확인하기
* -- git status : 파일의 상태를 확인하는 명령 기본 브랜치가 master
(On branch master …… ,,
untracked files : …. : untracked 상태
"Changes to be committed" : staged 상태
new file : ….. : 새로 tracked된 파일이 staged 상태)

파일을 새로 추적하기
git add <파일/ 디렉토리 경로> : 명령으로 디렉토리에 있는 파일을 staged 상태로 만들 때 사용. untracked 새로 추적하고 관리하도록 한다.
커밋하면 git add 를 실행한 시점의 파일이 커밋되어 저장소 히스토리에 남는다.
디렉토리를 add 하면 아래에 있는 모든 파일들까지 재귀적으로 추가

2.2 수정하고 저장소에 저장하기

Modified 상태의 파일을 Stage 하기 – git add
$ git status에서
"Changes not staged for commit :,,,,,, Modified :,,,,, " : modified 상태, 아직 Staged 상태는 아니기 때문에 --> git add 필요

Staged 상태인 파일을 수정하면
$ git status에서
"Changes to be committed: :,,,,,,,: Modified :....." ,
"Changes not staged for commit: ,,,,,,, : Modified :....." 두 가지 상태로 표시됨.
git add 명령을 실행한 후에 또 파일을 수정하면 git add 명령을 다시 실행해서 최신 버전을 Staged 상태로 만들어야 한다. 최신 버전을 staged 상태로 만들지 않으면 그 전 add한 시점의 staged된 파일이 커밋됨

git status -s 또는 git status --short : 현재 변경한 상태를 짧게 보여줌.
왼쪽에는 Staging Area에서의 상태를, 오른쪽에는 Working Tree에서의 상태를 표시.
_M : modified
M_ : staged
A_ : staged (new file- tracked)
MM : staged, 그 상태로 또 수정됨.
?? : untracked

***git add :**
1. 수정한 파일을 Staged 상태로 만들 때도 사용
2. untracked 파일도 새로 tracked, staged 상태로
3. Merge 할 때 충돌난 상태의 파일을 Resolve 상태로 만들때도 사용

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. .gitignore 파일은 보통 처음에 만들어 두는 것이 편리
$ cat .gitignore
*.[oa] : ".o" 나 ".a" 인 파일 무시 (빌드 시스템이 만들어내는 오브젝트와 아카이브 파일)
*~ : ~ 로 끝나는 모든 파일을 무시 (Emacs나 VI 같은 텍스트 편집기가 임시로 만들어내는 파일)
+ log, tmp, pid 같은 디렉토리나, 자동으로 생성하는 문서 같은 것들도 추가 가능
+ https://github.com/github/gitignore

2.2 수정하고 저장소에 저장하기

Staged와 Unstaged 상태의 변경 내용을 보기
-- git diff 명령 : 어떤 내용이 변경됐는지 살펴봄,
워킹 디렉토리 (아직 staged 상태가 아닌 파일)에 있는 것과 Staging Area에 있는 것을 비교해 볼 수 있다.
$ git diff
diff --git a/CONTRIBUTING.md b/CONTRIBUTING.md

-- git diff --staged 옵션 (--cached 는 같은 옵션)
커밋하려고 Staging Area에 넣은 파일의 변경 부분을 보고 싶을 때 사용. 저장소에 커밋한 것과 Staging Area에 있는 것을 비교

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

변경사항 커밋하기
수정한 것을 커밋하기 위해 Staging Area에 파일을 정리. 커밋하기 전에 git status 명령으로 모든 것이 Staged 상태인지 확인할 수 있다. 그 후에 git commit 을 실행하여 커밋.
-- $ git commit
Git 설정에 지정된 편집기가 실행 (보통 Vim이나 Emacs : git config --global core.editor 명령으로 사용할 편집기 설정 가능)
커밋 메시지 생성, 편집기 종료, 새 커밋 완성.
(첫 라인 : messange 두번째 : git status 결과) : 지우고 새로 작성 할 수 있다.
-- git commit 에 -v 옵션
: 편집기에 diff 메시지도 추가
-- $ git commit –m 옵션 : commit message 내용을 인라인으로 첨부

commit 명령 정보 출력 : 브랜치 이름, 체크섬 이름, 수정한 파일 개수, 삭제/추가된 라인 개수
하는데 위 예제는 (master) 브랜치에 커밋했고 체크섬은 (463dc4f)이라고 알려준다. 그리고 수정한 파일이 몇 개이고 삭제됐거나 추가된 라인이 몇 라인인지 알려준다.
-- $ git commit -a 옵션 : Staging Area를 생략,
Git은 Tracked 상태의 파일을 자동으로 Staging Area에 넣는다. git add 명령을 실행 불필요,
but 추가하지 말아야 할 변경사항도 추가될 수 있기 때문에 주의

2.2 수정하고 저장소에 저장하기 - 삭제하기

파일 Staging Area에서 삭제 git에서 파일 제거하려면 Tracked 상태의 파일을 삭제한 후에(정확하게는 Staging Area에서 삭제) 한 후 커밋해 줘야함.

-- rm명령/git rm 명령 :

1 rm실행하면 삭제한 파일은 staged 상태가 아님. status에 deleted 표시

-- $ git status

"Changes not staged for committed: ..... deleted: ....."

2 git rm실행하면 삭제한 파일은 staged 상태로 됨. status에 deleted 표시

$ git rm README

-- $ git status

"Changes to be committed: ..... deleted: README"

-- > 그 후 커밋.

파일은 삭제되고 Git은 이 파일을 더는 추적하지 않는다.

*** 이미 파일을 수정했거나 staging area에 추가된 경우**

-- -f 옵션 : 이미 파일을 수정했거나 Index에(역주 - Staging Area을 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.

*** Staging Area에서만 제거하고 워킹 디렉토리에 있는 파일은 지우지 않고 남겨두기**

하드디스크에 있는 파일은 그대로 두고 Git만 추적하지 않게 한다. (.gitignore 파일에 추가하는 것을 빼먹었거나 대용량 로그 파일이나 컴파일된 파일인 .a 파일 같은 것을 실수로 추가했을 때)

-- --cached 옵션을 사용하여 명령을 실행한다.

$ git rm --cached README

*** 여러 개의 파일이나 디렉토리를 한꺼번에 삭제**

git rm 명령에 file-glob 패턴을 사용

예 $ 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 삭제하라는 뜻

2.2 수정하고 저장소에 저장하기

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. 하지만 아는 방법이 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
```

git mv 명령은 일종의 단축 명령어 ($ mv README.md README $ git rm README.md $ git add README)

2.3 커밋 히스토리 조회하기

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.

-- git log : 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.

git log 명령은 매우 다양한 옵션을 지원

* -p 는 각 커밋의 diff 결과를 보여줌

: commit message 밑에 diff -- 보여줌 직접 diff를 실행한 것과 같은 결과를 출력하기 때문에 동료가 무엇을 커밋했는지 리뷰하고 빨리 조회하는데 유용

* -2`가 있는데 최근 두 개의 결과만 보여주는 옵션
* --stat 옵션으로 각 커밋의 통계 정보를 조회

: 어떤 파일이 수정됐는지, 얼마나 많은 파일이 변경됐는지, 또 얼마나 많은 라인을 추가하거나 삭제했는지 보여준다. 요약정보는 가장 뒤쪽에 보여준다.

* --graph 옵션 :브랜치와 머지 히스토리를 보여주는 아스키 그래프를 출력
* --pretty 옵션 → 형식 옵션 선택 : (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. format 옵션이다. 나만의 포맷으로 결과를 출력하고 싶을 때 사용- 포맷을 정확하게 일치시킬 수 있기 때문에 Git을 새 버전으로 바꿔도 결과 포맷이 바뀌지 않는다.) git log --pretty=format 에 쓸 몇가지 유용한 옵션 : https://git-scm.com/book/ko/v2/Git의-기초-커밋-히스토리-조회하기#pretty_format

조회 제한조건

출력 형식과 관련된 옵션을 살펴봤지만 git log 명령은 조회 범위를 제한하는 옵션들도 있다. 히스토리 전부가 아니라 부분만 조회 -2 옵션은 원래 –n 옵션

* --since 나 --until 같은 시간을 기준으로 조회하는 옵션 , 이 옵션은 다양한 형식을 지원 날짜, 상대적인 기간 설정.
* --author 옵션으로 저자를 지정하여 검색
* --grep 옵션으로 커밋 메시지에서 키워드를 검색

- -S 옵션 : 코드에서 추가되거나 제거된 내용 중에 특정 텍스트가 포함되어 있는지를 검색한다. 예를 들어 어떤 함수가 추가되거나 제거된 커밋만을 찾아볼 수 있다
- -- path이름 : 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

* --no-merges Merge 커밋을 제외한 순수한 커밋을 확인해보는 명령

2.4 - 되돌리기

되돌리기
일을 하다보면 모든 단계에서 어떤 것은 되돌리고(Undo) 싶을 때가 있다. 이번에는 우리가 한 일을 되돌리는 방법을 살펴본다.

* $git commit --amend 옵션을 사용하여 커밋을 재작성 다시 커밋하고 싶으면 파일 수정 작업을 하고 Staging Area에 추가한 다음 사용함.
: 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. 이전의 커밋을 완전히 새로 고쳐서 새 커밋으로 변경하는 것을 의미한다. 이전의 커밋은 일어나지 않은 일이 되는 것이고 당연히 히스토리에도 남지 않는다.
한 번 되돌리면 복구할 수 없기에 주의 : 되돌린 것은 복구할 수 없다.

파일 상태를 Unstage로 변경하기
Staging Area와 워킹 디렉토리 사이를 넘나드는 방법을 설명한다. 두 영역의 상태를 확인할 때마다 변경된 상태를 되돌리는 방법을 알려주기 때문에 매우 편리
* -- $git reset HEAD <file>
: git reset 명령은 매우 위험하다. --hard 옵션과 함께 사용하면 더욱 위험하다. 하지만 위에서 처럼 옵션 없이 사용하면 워킹 디렉토리의 파일은 건드리지 않는다.

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 checkout -- <file> : 원래 파일로 덮어썼기 때문에 수정한 내용은 전부 사라짐. 주의

+ 변경한 내용을 쉽게 버릴수는 없고 하지만 당장은 되돌려야만 하는 상황이라면 Stash와 Branch를 사용 (뒤에 브랜치에 나옴)

2.5 리모트 저장소

리모트 저장소 :
- 인터넷이나 네트워크 어딘가에 있는 저장소,
- 저장소는 여러 개가 있고 저장소 별로 읽고 쓰기, 읽기만 등 다름.
- 협업은 리모트 저장소를 관리하면서 데이터를 거기에 Push 하고 Pull 하는 것.
- 리모트 저장소를 관리한다는 것은 저장소를 추가, 삭제하는 것뿐만 아니라 브랜치를 관리하고 추적할지 말지 등을 관리
- **원격 저장소라 하더라도 로컬 시스템에 위치할 수도 있다.** "remote" 저장소여도 사실 같은 로컬 시스템에 존재할 수 있다.

리모트 저장소 확인하기
* -- git remote : 현재 프로젝트에 등록된 리모트 저장소를 확인
: 단축 이름을 보여준다.
- 저장소를 Clone 하면 `origin`이라는 리모트 저장소가 자동으로 등록되기 때문에 `origin`이라고 보여줌.
• -v 옵션 : 단축이름과 URL을 보여줌

리모트 저장소가 여러 개라면
- git remote 를 이용하면 등록된 전부를 보여줌, 여러 사람과 함께 작업하는 리모트 저장소가 여러 개라면 여러 개가 나옴.
다른 사람이 기여한 내용(Contributions)을 쉽게 가져올 수 있음.
- 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 에서

리모트 저장소 추가하기
git remote add [단축이름] [url] 명령 : 기존 워킹 디렉토리에 새 리모트 저장소를 쉽게 추가.
```
$ git remote add pb https://github.com/paulboone/ticgit
$ git remote -v
pb  https://github.com/paulboone/ticgit (fetch)
pb  https://github.com/paulboone/ticgit (push)
```
→이제 URL 대신에 **pb 라는 이름을 사용해서 로컬 저장소에 없는 것을 가져 올 수 있음**
+
```
$ git fetch pb
.....
 * [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 확인
+ 브랜치를 어떻게 사용하는지는 Git 브랜치 에서

2.5 리모트 저장소

리모트 저장소 추가하기
git remote add [단축이름] [url] 명령 : 기존 워킹 디렉토리에 새 리모트 저장소를 쉽게 추가.
$ git remote add pb https://github.com/paulboone/ticgit
$ git remote -v
pb https://github.com/paulboone/ticgit (fetch)
pb https://github.com/paulboone/ticgit (push)
→이제 URL 대신에 **pb 라는 이름을 사용해서 로컬 저장소에 없는 것을 가져 올 수 있음**
+ 예 pb
$ git fetch pb
.....
 * [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 확인
+ 브랜치를 어떻게 사용하는지는 Git 브랜치 에서

리모트 저장소를 Pull 하거나 Fetch 하기
$ git fetch [remote-name] : 리모트 저장소에서 로컬에 없는 데이터를 모두 가져올 때 사용. 그 후 리모트 저장소의 모든 브랜치를 로컬에서 접근할 수 있어서 언제든지 Merge를 하거나 내용을 살펴볼 수 있다.

* -- Clone 명령 :
- 자동으로 리모트 저장소를 단축이름 "origin"으로 추가. 그 후 git fetch origin 명령을 실행 시 마지막으로 가져온 이후에 수정된 것을 모두 가져옴. 하지만 merge는 안 해서 수동으로 해 줘야함.
- 자동으로 로컬의 master 브랜치가 리모트 저장소의 master 브랜치를 **추적하도록** 한다(물론 리모트 저장소에 master 브랜치가 있다는 가정에서).
* -- git pull 명령 : merge까지 자동,
: clone한 서버 리모트 저장소 브랜치에서 데이터를 가져올 뿐만 아니라 자동으로 로컬 브랜치와 Merge 시킬 수 있음. + git 브렌치에서 설명

2.5 리모트 저장소

리모트 저장소에 PUSH하기
* -- $ git push [리모트 저장소 이름] [브랜치 이름]
Clone 한 리모트 저장소에 쓰기 권한이 있고, Clone 하고 난 이후 아무도 Upstream 저장소에 Push 하지 않았을 때(같은 버전일 때)만 사용, 여러 명이 clone했을 때, 다른 사람이 Push 한 후에 Push 하려고 하면 Push 할 수 없다. 먼저 다른 사람이 작업한 것을 가져와서 Merge 한 후에 Push 해야함

리모트 저장소 이름 바꾸거나 리모트 저장소 삭제하기
* -- $ git remote rename [현재 이름] [바꿀 이름]
: 로컬에서 관리하던 리모트 저장소의 브랜치 이름도 바뀐다 pb/master → paul/master

리모트 저장소 살펴보기
* -- $ git remote show [리모트 저장소 이름] : 리모트 저장소의 구체적인 정보를 확인.
- 리모트 저장소의 URL과 추적하는 브랜치와 가져온 모든 리모트 저장소 정보 출력.

$ 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.

2.6 태그

태그 : 누군가 저장소에서 Clone 하거나 Pull을 하면 모든 태그 정보도 함께 전송
Git의 태그의 종류 : Lightweight 태그와 Annotated 태그로 두 종류가 있다.
- Lightweight 태그는 브랜치와 비슷한데 브랜치처럼 가리키는 지점을 최신 커밋으로 이동시키지 않는다. 단순히 특정 커밋에 대한 포인터일 뿐이다. 임시로 생성하는 태그거나 정보를 유지할 필요가 없는 경우
- 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

* -- $ git tag : 이미 만들어진 태그가 있는지 확인, 검색 패턴을 사용하여 태그를 검색 가능 예 $ git tag -l "v1.8.5*" 8.5버전 태그만

1 annotated 태그
$ git tag –a : annotated 태그 만들기
$ git tag –m : 태그를 저장할 때 메시지를 함께 저장할 수 있다. 명령을 실행할 때 메시지를 입력, 입력하지 않으면 Git은 편집기를 실행시킨다.
$ git show [태그이름] :태그정보와 커밋 정보 확인
: 커밋 정보를 보여주기 전에 먼저 태그를 만든 사람이 누구인지, 언제 태그를 만들었는지, 그리고 태그 메시지가 무엇인지 보여준다.

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

나중에 태그하기
* $ git tag -a [태그이름] [커밋 체크섬] :
예전 커밋에 대해서도 나중에 태그를 붙일 수 있다. 특정 커밋에 태그하기 위해서 명령의 끝에 커밋 체크섬을 명시한다(긴 체크섬을 전부 사용할 필요는 없다).

2.6 태그

태그 공유하기
git push origin [태그 이름]
git push 명령은 자동으로 리모트 서버에 태그를 전송하지 않는다. 태그를 만들었으면 서버에 별도로 Push 해야 한다. 브랜치를 공유하는 것과 같은 방법으로 할 수 있다. `

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

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

2.7 git alias

자주 사용하는 명령은 Alias을 만들어 사용할 수 있다. (단축키같이)
git config 사용
$ git config --global alias.ci commit : git commit 대신 git ci 만으로도 커밋 수행해라
$ git config --global alias.unstage 'reset HEAD --' : git unstage 명령으로 파일을 Unstaged 상태로 변경하라
$ git config --global alias.last 'log -1 HEAD' : 최근 커밋 확인

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

외부 명령어 실행
！를 제일 앞에 추가하면 외부 명령을 실행
$ git config --global alias.visual '!gitk' : git visual 이라고 입력하면 gitk 가 실행된다.

3.1 Git 브랜치란

- 브랜치란 : 코드를 통째로 복사하고 나서 원래 코드와는 상관없이 독립적으로 개발하는 것
- Git의 최고 장점. 브랜치는 매우 가볍고 브랜치 사이를 이동 가능하고 작업 후 나중에 Merge하는 것을 권장.

- git의 데이터 저장
- git은 chage set이나 변경 사항 대신 일련의 스냅샷으로 기록

- 커밋하면 Git은 현 staging area에 있는 데이터의 스냅샷에 대한 포인터, 저자나 커밋 메시지 같은 메타 데이터, 이전 커밋에 대한 포인터 등을 포함하는 커밋 개체