
Git (분산형 버전 관리 시스템)

Git 은 Linux kernel 을 만든 것으로 유명한 Linus Torvalds 가 만든 강력한 버전 관리 시스템 (VCS)으로, 커밋 기준의 버전 관리를 가능하게 한다.

Git 은 공식적으론 Texinfo 기반 매뉴얼이 없고, 그 대신 Manpage 를 제공하며, 이미지가 들어가야 하거나 좀 긴 매뉴얼은 대부분 웹 페이지의 형태로 제공한다. Git 은 자유 소프트웨어지만 GNU 계열 소프트웨어는 아니기 때문에 GNU Coding Standard 를 따르진 않기 때문이다.

시작하기

Git 을 시작하게 되었다면 몇 가지 설정 및 기본적인 사용 방법을 익혀야 한다. 이곳에는 일단 그것에 대해서 적어두겠다.

Git 관련 매뉴얼을 찾는 방법

Git 의 모든 기능은 CLI 로 제공되며, 이들의 대부분은 매뉴얼이 함께 제공된다. 대부분은 ¦git commit¦ 과 같이 'git' 뒤에 명령 이름이 오는 형태로 사용된다. ¦git commit¦ 명령에 대한 매뉴얼을 보려면 ¦man git-commit¦ 혹은 ¦man git commit¦ 을 하도록 하라. 혹은 Emacs 의 org-mode 의 Man-page link 가 활성화 되어 있다면 man:git-commit.1 의 형태로도 참조 가능하다.

Git 의 전역 설정 정보를 설정하기

¦git config --global¦ 로 설정한 설정은 ~/.gitconfig 에 저장된다. 설정을 확인하고자 한다면 그 파일을 참고할 수 있다.

Git 을 시작하기 전에 일단 적당히 Git 에 대한 config 를 해두는 것이 좋다.

: git config --global user.name "Hong Gil-dong"
: git config --global user.email "gildong@example.com"

위는 commit 에 들어가는 사실상 최소한의 정보다. 위 두 정보가 없으면 Git 은 일단 적당히 hostname 과 username 으로 커밋해놓고, 경고를 준다. 모든 커밋에는 유저 이름과 이메일이 필요하기 때문이다. 위 정보는 커밋에 대해 '누가'에 관련된 정보다.

이런 경우 일단 commit 후에 고칠 수는 있다.

: git commit --amend --reset-author

위는 방금 전에 했던 커밋의 author 를 고치는 방법이다. 단, 앞서 했던 커밋 이후에 내용의 변경이 없어야 한다. Git 은 기본적으로 commit 을 하고 나면 그 커밋 노드가 변하지 않는 것을 원칙으로 하기 때문이다.

파일 이름이 CJK 계열인 경우 quote 되어 버리는 문제가 있는데, 이 경우엔 다음과 같이 파일 이름이 제대로 보이게 하기 위한 설정 정도는 해두는 것이 좋다.

: git config --global core.quotepath off

위를 다 하고 난 뒤에 전체가 어떤 식으로 ~/.gitconfig 에 반영되었는지 본다.

: git config --global --edit

참고로 ~/.gitconfig 를 ~/.config/git/config 로 옮기면 Git 는 ~/.config/git/config 를 사용한다.

Git 저장소를 시작하기 (init, add, commit)

일단 Git 저장소를 생성해야 Git 를 쓸 수 있다. 빈 저장소의 생성은 ¦git init¦ 으로 가능하다.

다음은 간단한 예시다.

: mkdir foobar; cd foobar
: git init . # foobar 에 저장소 시작. .git 이라는 디렉토리가 생성됨.
: nano README # 저장소 내부에 파일을 만들고 내용을 추가
: nano hello.c # ...
: git add . # 현재 디렉토리에 있는 모든 파일을 stage 함
: git commit # 최초 커밋

¦git init¦ 은 저장소를 시작한다. 디렉토리를 지정안하면 current working directory 를 저장소의 root 로 사용하며, 아무 옵션도 안주면 .git 이라는 디렉토리를 저장소에 바로 생성한다. 이때 생성되는 .git 디렉토리를 repository 라고 부른다.

일반적으로 파일이 보이는 상태를 **working directory** 라고 부른다. 참고로 ¦git init --bare¦ 으로 working directory 를 가지지 않는 repository 를 제작하는 것도 가능하며, 이는 보통 배포용 관리 저장소를 위해 사용된다.

¦git add¦ 를 하지 않으면 git 은 변경에 대해서 commit 을 하지 않으므로, Git 으로 관리되고 있는 디렉토리에서 새로운 파일이나 디렉토리를 *추가* 했거나, 파일을 *수정* 하거나 *삭제* 했을 경우 Git 에 그 변경을 추가하도록 ¦git add¦ 로 명시해줘야 한다.

다음은 몇 가지 예시다.

단순히 파일들을 추가 하기:
: git add file1 file2 file3

현재 디렉토리 내의 변경/추가된 모든 파일을 추가하기:
: git add .

디렉토리인 경우, 디렉토리 이하의 모든 파일을 다 추가한다:
: git add dir1

¦git add¦ 에 의한 추가를 staging 혹은 index 에 추가한다고 한다. 내부적으론 index 라는 용어로 불리며, 관습적으론 staging area, cache 라고도 불린다. index 는 commit 하기 전엔 저장소에 추가된 변경사항이 아니므로 주의하라.

¦git add¦ 는 add 한 시점의 content 를 index 에 넣는다. 즉, ¦git add¦ 하고 난 뒤에 파일을 수정했다거나 할 경우는 그 수정을 커밋에 반영하기 위해서는 다시 ¦git add¦ 를 해줘서 index 의 내용을 갱신해야 한다. ¦git add¦ 한 시점에서 index 가 저장되기에, 만약 ¦git add¦ 한 이후에 파일 수정을 했는데 ¦git add¦ 를 다시 하지 않고 ¦git commit¦ 했다면, Git 는 index 의 정보를 commit 하지, 현재 다시 수정된 정보를 commit 하진 않는다. 일반적으로 working directory 와 index 사이에 불일치가 있는 경우, Git 는 commit 과 동시에 경고 메시지를 표시한다.

같은 파일을 여러 번 수정하면서 그때마다 ¦git add¦ 를 하면 index 는 병합되어 적층된다. 즉, 마지막 ¦git add¦ 한 시점의 정보가 커밋에 반영되므로, 커밋 전에 여러 번 ¦git add¦ 하는 것은 딱히 add 후에 수정이 없는 한, 커밋의 결과에 영향을 미치지 않는다.

¦git commit¦ 은 index 에 추가된 파일들을 repository 에 추가해준다.

단순히 파일 추가는 없고, 수정만 있었을 경우는 commit 의 -a 옵션으로 수정된 파일에 대해서 add 를 지정한 것 처럼 할 수도 있다.

: git commit -a

저장소의 정보 보기 (status, diff, log)

¦git status¦ 는 앞으로 커밋 될 index 의 정보를 볼 수 있다.

: git status

현재 index 에 stage 된 정보와 가장 최근 커밋의 차이를 보려면 ¦git diff --cached¦ 를 할 수 있다.

: git diff --cached

¦git log¦ 로 지금까지 커밋들의 간략한 change log 를 보는 것이 가능하다.

```
: git log                  # 간략하게
: git log -p               # diff 를 함께 보여줌
: git log --stat --summary   # 유용한 여러 정보들과 함께 표시
```

옵션들(-p, --stat, --summary 등)은 중첩해서 사용할 수 있다. ¦git log¦ 가 보기 불편하다면 Gitk 같은 GUI 툴을 사용할 수도 있다.

저장소에 커밋하기

¦git commit¦ 은 현재의 index 에 추가된 snapshot 을 repository 에 새로운 노드로 만들어 저장하는 것이다. 이 노드는 cherry picking 이 가능하다. 이것은 branch 와 함께 이해되어야 한다.

commit 은 저장소에 그 시점을 정확히 저장해주는 역할을 한다. 기본적으로 커밋을 한 순간은 git 저장소에 보존되어 언제든지 다시 가져올 수 있다. 가져오기 위한 방법은 단순히 같은 branch 에서라면 cherry picking 도 있고, reset 같은 것도 있다. 어떤 시점에서 프로그램의 분기를 나누는 것 역시 가능한데 그것은 branch 로 행한다.

¦git commit¦ 을 하는 기본 work flow 는 다음과 같다.

파일의 추가나 이름 변경(rename)이 있었을 때는 다음과 같다.

: git add . # 일단 변경 사항을 index 에 추가
: git status # 앞으로 어떤 커밋이 일어날지 보고
: git commit # 실제 커밋
: git log # 필요하다면, 이전의 커밋들에 대해서 확인해둔다.

일반적으론 위의 형태가 적절하다만, 가끔은 더 간단한 경우도 있다.

단순히 파일 수정이나 삭제만 했고, 딱히 추가는 없었을 때는 다음과 같이 해도 된다.

: git status
: git commit -a # 수정된 파일은 자동으로 커밋.
: git log

커밋 메시지의 관습

git commit 을 할 때는 환경변수 EDITOR 에 지정된 에디터가 실행되어 커밋 메시지를 요구하는데, 메시지가 비어 있으면 커밋을 중지 한다. 커밋은 반드시 메시지가 있어야 한다. 이때, 메시지가 비어있다는 것의 판정에 빈 행과 # 로 시작하는 comment 행은 무시한다.

기본적으로 커밋은 맨 첫 줄에 50 글자 이하의 반각 영어로 commit 에 대한 요약을 적고, 그 다음 줄에 빈 줄을 하나 넣고 나서 이후로 자세한 commit 에 대한 상세를 적는다. Git 는 자동으로 맨 첫 줄을 summary(title) 로 취급한다.

커밋 로그는 원칙적으로 현재형(present tense)으로 적는 것이 추천된다. 즉, “Fixed typo error in foobar.txt” 라고 적는 것이 아닌, “Fix typo error in foobar.txt” 라고 적는 것이 좋다. 커밋은 그 커밋과 함께 작업을 flush 한다는 느낌으로 생각하기 때문이다.

간결한 커밋 (git commit -am blah)

commit 을 할 때 딱히 메시지 창을 띄우지 않고 메시지를 명령줄에서 바로 입력하여 바로 커밋 하도록 하려면 -m 옵션을 쓸 수도 있다.

: git commit -am "Trivial changes and fixing typo errors"

파일의 추가가 따로 없는 간단한 커밋의 경우, 위와 같이 -m 옵션과 -a 옵션을 함께 쓰면 간결하다.

방금 전의 커밋을 고치기 (git commit –amend)

바로 전 커밋을 고치고 싶다면 --amend 옵션을 쓸 수 있다.

: git commit --amend

방금 커밋한 것에 그다지 큰 수정이 아닌 경우 커밋을 두 번하면 이력이 더러워질 뿐이니, 마이너한 업데이트나 실수 수정은 이전 커밋을 지워버리고 새로 커밋하는게 낫다. 그렇게 하고싶을 경우 ¦git commit --amend¦ 를 하면 방금 전 커밋한 것이 대체된다.

분기 만들기 (branch, checkout)

Git 는 저장소를 만들면 기본적으로 master 라는 branch 를 만들고, 그것을 움직이면서 커밋 노드를 파생시켜 나간다. 도중에 실험적인 기능 등을 추가하기 위해서 현재 master 가 가리키는 커밋 노드는 그대로 놔두고, 그 커밋 노드에서부터 분기된 새로운 파생을 만들고 싶다면, branch 를 쓴다.

: git branch experi

위는 'experi' 라는 분기(branch)를 만든다. ¦git branch¦ 를 하면 현재 선택되어 있는 branch 가 무엇인지와, 어떤 branch 의 이용이 가능한지 보여준다. branch 를 이동하여 시점을 변경하고 싶으면 checkout 을 사용한다.

: git checkout experi

이 상태에서 파일을 추가하거나 수정하여 커밋하면, experi branch 가 커밋 노드를 만들어 뻗어나가며, master 쪽은 따로 커밋 노드를 만들어 뻗어나간다.

단, ¦git add¦ 로 추가해둔 index 의 정보는 딱히 branch 에 종속된 것은 아니므로 주의하라. index 는 애초에 commit 하기 전에는 .git repository 에 반영되는 것이 아니다. 단, 이러한 confliction 이 발생하는 상황에서 checkout 혹은 commit 하려고 하면 Git 가 알아서 경고 혹은 에러를 준다. _ 일반적으론 checkout 하기 전에 commit 을 먼저 해야한다_.

예를 들어서 foo.c 라는 파일이 v1 상태에서 ¦git branch experi¦ 를 하고, master 에서 foo.c 를 v2 상태로 만들고 commit 했다고 하자. experi branch 에서는 foo.c 는 여전히 v1 인데, experi branch 에서 이걸 수정해서 v2-ex 상태로 만든 상태에서 master 로 checkout 하려고 하면 Git 은 에러를 준다. 이것은, master 의 foo.c 는 v2 인데, checkout 하게 되면 그것이 v2-ex 로 덮어쓰기 되어버리기 때문이며, 이것을 회피하기 위해 Git 이 에러를 주는 것이다. 이것을 해결하기 위해서는 experi branch 에서 v2-ex 를 commit 하고 master 로 checkout 해야한다.

Git 의 branch 는 실제로는 DAG 의 한 노드를 가리키는 이름으로, 얼마든지 생성하고 삭제해도 repository 의 실제 저장 데이터 자체는 복잡해지지 않는다. graphical 하게는 Gitk 를 사용해서 branch 이름 위에서 remove this branch 를 하면 삭제된다. Gitk 자체는 브랜치의 생성도 간단히 가능하며, 단순히 커밋 노드에서 브랜치를 생성하기만 하면 된다. 매우 단순하고 깔끔하다.

분기를 통합하기 (merge)

¦git merge¦ 는 두 개의 branch 가 가리키는 노드 두 개를 통합한(merge) 노드를 생성한다. 통합은 confliction 을 동반하며, 이 confliction 은 보통은 수동으로 해결되어야 한다.

Git 의 노드는 기본적으로 branch 가 가리키면서 뻗어나가고, 기본적으로 다른 두 branch 는 서로 다른 커밋으로 뻗어나간다. 예로 experi branch 에서 어떤 기능이 구현되어서 잘 작동하면, 이것을 master 로 반영하고 싶을 때가 올 것이다. 그럴 때는 master branch 가 experi branch 를 merge 한다고 하며, 다음과 같이 한다.

: git checkout master # master branch 로 이동하고
: git merge experi   # experi 를 merge 한다.

이 경우 master 가 '이곳(이 쪽)' 이 되고, experi 가 '저곳(저 쪽)' 이 된다. merge 이후에도 experi 가 가리키는 노드 자체는 변화하지 않는다. 단지 DAG 가 변할 뿐이다.

따로 confliction 이 없으면 바로 merge 되어 commit 되고 난 뒤에 commit 된다. 문제는 confliction 이 있는 경우다.

분기 통합 상황의 가정

merge 기본적으로 파일들에 대하여 "최신의 상태"가 우선되어 채택되며, 분기가 있을 경우에 confliction 이 일어난다.

다음과 같은 경우를 생각하자.

- master 의 파일과 그 버전:
  - foo.c : v1
  - bar.c : v2
  - baz.c : v-master (이전 버전은 baz.c 의 v1)
  - qux.c : v2
  - (삭제됨) foobar.c : v1
- experi 의 파일과 그 버전:
  - foo.c : v2
  - bar.c : v1
  - baz.c : v-experi (이전 버전은 baz.c 의 v1)
  - (삭제됨) qux.c : v1
  - foobar.c : v1

master 문맥에서 git merge experi 를 하였을 때 각 파일은 이하와 같이 병합된다.

- foo.c : v2 (master 의 파일)
- bar.c : v2 (experi 의 파일)
- (conflict!) baz.c : v-merge (v-master 와 v-experi 가 diff 로 적혀짐)
- (conflict!) qux.c : master 의 v2 가 남아 있고, 커밋에서 그것이 삭제되었음을 말해줌
- (삭제됨) foobar.c : v1

confliction 을 해결해야하는 것은 baz.c 및 qux.c 이며, baz.c 의 경우 파일을 열어서 diff 된 부분을 처리하고, qux 는 삭제할 것인지 보존할 것인지를 정해야한다.

이처럼, merge 에서 experi 와 master 가 분기된 이후에 experi 와 master 둘 다 파일을 수정했거나(baz.c 의 경우), 어느 한 쪽은 수정하고 다른 쪽은 삭제해버린 파일(qux.c 의 경우)이 있을 경우에 confliction 이 발생하게 된다.

양측에서 수정한 파일을 통합하기

우선, baz.c 와 같이 양쪽에서 파일을 수정한 경우는 기본적으론 diff 의 형태로 새로운 버전(v-merge)이 생성된다.

baz.c (v-merge) 의 내용 예시:
```
#+begin_src c
main() {
<<<<<<<<< HEAD
  printf("master 쪽의 내용!");
=========
  printf("experi 쪽의 내용!");
>>>>>>>>> experi
}
#+end_src
```

위와 같이 기본적으론 diff 형식으로 새로운 버전이 생성된다.

이런 경우에 ¦git status¦ 를 해보면 다음과 같은 메시지를 볼 수 있다.

```
: $ git status baz.c
: 현재 브랜치 master
: 병합하지 않은 경로가 있습니다.
```

:   (충돌을 바로잡고 "git commit"을 실행하십시오)
:   (병합을 중단하려면 "git merge --abort"를 사용하십시오)
:
: 병합하지 않은 경로:
:   (해결했다고 표시하려면 "git add <파일>..."을 사용하십시오)
:   양쪽에서 수정: foo
:
: 커밋할 변경 사항을 추가하지 않았습니다 ("git add" 및/또는
: "git commit -a"를 사용하십시오)

해야 할 것은 baz.c 를 원하는 대로 수정하고, 그것을 index 에 추가한 후에 commit 하는 것이다.

: nano baz.c   # confliction 된 것을 확인하고 수동으로 수정함.
: git add baz.c # 수정한 내용을 index 에 staging 함
: git commit   # index 를 커밋.

단, 경우에 따라선 (충돌 없이) 다음과 같이 증가적으로 자동적으로 병합되어버리는 경우도 있다. 이 경우는 conflction 이 없는 것이므로 바로 merge 되어 commit 화면이 뜬다.

baz.c (v1) 의 내용:
#+BEGIN_SRC c
main() {
  printf("foobar");
}
#+END_SRC

위와 같은 원본 소스 코드가 있고, master branch 및 experi branch 에서 수정이 되었다고 한다.

baz.c (v-master) 의 내용:
#+BEGIN_SRC c
main() {
  printf("master add some codes BEFORE foobar");
  printf("foobar");
}
#+END_SRC

baz.c (v-experi) 의 내용:
#+BEGIN_SRC c
main() {
  printf("foobar");

```
  printf("experi add some codes AFTER foobar");
}
#+END_SRC
```

위와 같은 경우에 master 에서 experi 를 merge 할 경우, confliction 없이 다음과 같이 병합된다.

baz.c (v-merge) 의 내용:

```
#+BEGIN_SRC c
main() {
  printf("master add some codes BEFORE foobar");
  printf("foobar");
  printf("experi add some codes AFTER foobar");
}
#+END_SRC
```

물론, 이런 경우에도 confliction 을 주도록 하는게 가능하긴 하겠다만, 이것 자체는 병합 전에 diff 를 통해서 확인 가능한 형태다.

한 쪽에서는 삭제하고, 다른 한 쪽에서는 수정한 경우를 통합하기

qux.c 의 경우 experi 에서는 삭제되었지만 master 쪽에선 새로운 버전이 있는 상태다. 이런 경우는 일단은 master 의 파일이 남는다. (역으로 master 에서 삭제하고 experi 에서는 수정한 경우도 experi 쪽의 파일이 생기게 된다.) 참고로 한쪽에서만 삭제하고, 나머지 한쪽에서는 파일을 건드리지 않은 경우(foobar.c 의 경우)는 confliction 이 아니므로 삭제가 반영되게 된다. 삭제도 결국 '수정' 이기 때문이다.

이런 경우에 ¦git status qux.c¦ 를 해보면 다음과 같은 메시지를 볼 수 있다.

```
: $ git status qux.c
: 현재 브랜치 master
: 병합하지 않은 경로가 있습니다.
:   (충돌을 바로잡고 "git commit"을 실행하십시오)
:   (병합을 중단하려면 "git merge --abort"를 사용하십시오)
:
: 병합하지 않은 경로:
:   (해결했다고 표시하려면 알맞게 "git add/rm <파일>..."을 사용하십시오)
:         저 쪽에서 삭제: qux.c
:
: 커밋할 변경 사항을 추가하지 않았습니다 ("git add" 및/또는
```

: "git commit -a"를 사용하십시오)

역의 경우는 “저 쪽에서 삭제: qux.c” 가 “이 쪽에서 삭제: qux.c” 로 표시될 것이다. master 에서 할 수 있는 선택은 어느 경우라도 두 가지다. 1) qux.c 를 유지하는 경우, 2) qux.c 를 삭제하는 경우.

유지할 경우는 add 를 해주고 commit 하면 된다.

: git add qux.c
: git commit

삭제할 경우는 rm 을 하고 commit 하면 된다.

: git rm qux.c
: git commit

분기 통합 후의 작업

master 가 experi 를 병합했을 경우, 병합 직후의 master 가 가리키는 노드는 원래 master 가 가리키던 노드와 experi 가 가리키는 노드의 후손 노드가 된다. experi 는 여전히 원래 자신이 가리키던 노드를 가리키고 있는 상태다.

원한다면 experi branch 는 그대로 놔둬도 아무 문제가 없다. 다만 깔끔한 게 좋다면 사용하지 않는 branch 를 삭제한다는 선택지가 있다.

: git branch -d experi

단, 위는 master 로 checkout 된 상태에서 해야하며, 동시에 확실히 병합이 끝난 상태에서 해야한다. 병합의 커밋이 완료되지 않은 경우 “브랜치가 완전히 병합되지 않았습니다” 라고 에러를 줄 것이다.

merge 가 완료된 시점에서, experi branch 를 master 로 reset 해도 된다. 방법은 여러 가지 있는데, experi 쪽에서 master 를 merge 하는 게 가장 알기 쉽다.

: git checkout experi
: git merge master

위는 conflction 이 발생하지 않는다. 이는 experi 가 가리키는 노드가 master 가 가리키는 노드의 조상 노드이기 때문이다. 조상 노드와 자식 노드의 병합은 confliction 이 발생하지 않는다.

태그 붙이기 (tag)

소프트웨어는 흔히 '버전' 을 붙인다. Git 는 버전이라는 용어 대신 '태그' 라는 용어를 사용한다. 이 태그는 "움직이지 않는 branch" 로 이해할 수 있다. branch 는 commit 이나 reset 을 통해 DAG 위를 움직이지만, 태그는 어떤 노드에 한 번 지정되면 (명시적으로 태그를 삭제하지 않는 한) 계속해서 유지된다.

예를 들어서 현재 HEAD 노드에 ver.1 이라는 tag 를 붙이고자 한다면 다음과 같이 할 수 있다.

: git tag ver.1

branch 와 마찬가지로 태그도 얼마든지 붙일 수 있다.

노드의 해시를 지정해서 태그를 붙일 수도 있다.

: git tag ver.1 36db…6030

위를 하면 36db…6030 라는 해시를 가지는 노드에 태그 ver.1 가 붙게 된다.

가장 최근 태그에 근거해서 그로부터 커밋수 및 고유번호를 보여줄 수도 있다.

: git describe --tags

태그붙이는 것은 Gitk 에서도 가능하다.

참고로 가장 최근의 커밋에 뭔가 태그를 붙였다가, ¦git commit --amend¦ 를 한 경우, 그 태그는 사라진다. 이는 ¦git commit --amend¦ 는 사실상 가장 최근 커밋을 삭제한 후에 다시 커밋하여 노드를 대체하기 때문이다. 태그는 커밋 노드에 붙는 것이라, 노드가 사라지면 태그도 사라진다.

Git 로 협업하기

Git 는 버전 관리를 위한 도구이자 동시에 협업을 위한 도구다. 이 협업은 두 사람 이상의 공동 작업을 의미할 뿐만 아니라 한 사람이 여러 대의 컴퓨터에 Git 저장소를 두고 서로 동기화 하며 사용하는 경우도 이른다.

이곳에는 SSH 서버를 사용한 협업을 위한 몇 가지 기본을 적어둔다.

---
bare 저장소의 제작 (git init --bare)

Git 를 사용하면 보통은 동기화 전용 저장소를 하나 두고, 그 저장소와 동기화를 하면서 사용하는 것이 보통이다. 이 동기화 저장소는 여러 형태가 가능하다만, 가장 손쉽게는 컴퓨터 안에 Git 저장소를 하나 만들고, SSH 를 통해 동기화 하는 것이다.

주로 push 를 당하는 것이 전제되는 외부 Git 저장소는 bare 저장소로 만들 수 있는데, 이 경우 .git 에 해당하는 폴더만을 만든 것이 된다.

다음과 같이 bare 저장소를 만들 수 있다.

: git init --bare /path/to/storage/foobar.git

위는 저장 만을 담당하는 Git 저장소가 제작된다. bare 저장소는 폴더 이름 맨 끝에는 관습적으로 .git 을 붙인다. 단, 어디까지나 관습이므로 이를 지키지 않더라도 큰 문제는 되지 않는다.

bare 저장소는 working directory 가 존재하지 않는다. 따라서 bare 저장소의 업데이트는 pull 과 push 로만 가능하다. 외부 저장소에 working directory 가 있으면 성가실 수 있으므로 보통 외부 저장소는 bare 저장소로 만든다.

만약 bare 저장소가 아닌 경우, 예를 들어서 만약 /path/to/remote/repo 쪽의 master 가 checkout 된 상태라면 push 가 안된다. 다른 사람이 사용하고 있는 상태이기 때문이다. bare 저장소라면 이런 경우는 발생하지 않지만, 로컬에서 로컬로 push 할 경우 발생할 수 있다. 그럴 때는 /path/to/remote/repo 쪽에서 master 말고 다른 branch 로 checkout 해둔 상태로 만들어두면 된다. 혹은 저장소에서 ¦git config receive.denycurrentbranch ignore¦ 로 설정하여 현재 branch 라도 push 되도록 하는 방법도 있다.

receive.denycurrentbranch 를 ignore 해주는 방법은 가끔 Rsync 대신에 Git 저장소 자체로 동기화 할 때 사용되곤 하는 방법이다.

외부 저장소와 현재 저장소를 연결시키기 (remote)

이미 어느정도 내부적으로 진행된 프로젝트가 있어서, 그것으로부터 외부 저장소를 생성하거나 혹은 동기화를 하고싶다면, remote 를 써서 저장소를 연결해야 한다.

: git remote add origin /path/to/storage/foobar.git # 외부 저장소 추가
: git remote -v # 확인

origin 은 앞으로 외부 저장소를 가리키는 이름이 된다. master 와 마찬가지로 origin 은 보통 외부 저장소를 가리키는 관습적인 이름이다. 다른 이름을 쓰고 싶다면 다른 이름을 써도 문제 없다.

위는 로컬에 있는 경로인 /path/to/storage/foobar.git 를 지정해주고 있으나, SSH 상의 경로를 지정해주는 것도 가능하다.

: git remote add origin ssh://myuser@myserver:4321/path/to/hoge.git

위는 myserver 라는 host 의 myuser 라는 유저명으로 4321 포트를 사용해서 SSH 로 연결하여 /path/to/hoge.git 라는 경로에 있는 Git 저장소를 외부 저장소로 지정하고 있는 것이다.

¦git remote add¦ 를 한 후에 처음 ¦git push¦ 나 ¦git pull¦ 을 하면 다음과 같은 에러를 본다.

: $ git push
: fatal: 현재 브랜치 master에 업스트림 브랜치가 없습니다.
: 현재 브랜치를 푸시하고 해당 리모트를 업스트림으로 지정하려면
: 다음과 같이 하십시오.
:
: git push --set-upstream origin master
:

에러에서 제안하듯 upstream branch 를 다음과 같이 지정해줘야 한다.

: git push --set-upstream origin master

참고로 저장소를 clone 해서 오면 기본적으로 복사된 저장소의 remote/origin 이 원본 저장소가 된다.

저장소를 복제하기 (clone)

¦git clone¦ 으로 다른 저장소에서 clone 해서 새로운 저장소를 만들 수 있다.

: git clone /path/to/origrepo myrepo

인터넷을 통해 연결된 저장소에서 clone 하는 것도 가능하다.

: git clone https://git.example.com/hello/hello-world.git hello

위와 같은 형태로 clone 하면 자동으로 remote/origin 이 자동으로 지정된다.

다음과 같은 커맨드로 현재 저장소와 연결된 외부 저장소 리스트를 볼 수 있다.

: git remote -v

저장소에서 가지고 오기 (pull, fetch)

우선 clone 으로 origrepo 에서 myrepo 를 만들었다고 하자.

: git clone /path/to/origrepo myrepo

위는 myrepo 라는 디렉토리를 만들어서 clone 하는 작업을 하는 것이다. myrepo 의 주인이 origrepo 에 기여하기 위해서는 이렇게 clone 을 해서 자신의 버전을 만들고, myrepo 에서 수정을 한 후에, origrepo 쪽의 관리자에게 “내 repo 의 수정을 반영하지 않겠습니까?” 하고 메일 등의 방법으로 request 를 보내면 origrepo 쪽의 주인이 pull 하여 origrepo 쪽에 수정이 반영되게 된다:

: git pull /path/to/myrepo

이 요청을 보통 pull requset 라 부른다.

github 같은 git 를 사용한 소위 "소셜 코딩 사이트"는 기여하고 싶은 프로젝트에 대해 그 프로젝트의 repo 를 자신이 관리 가능한 github repo 로 clone(fork)하여서 pull request 를 보내는 형태로 이뤄진다. 물론, 자신이 관리하는 repo 에 컴퓨터의 local repo 쪽에서 업데이트를 할 때는 push 를 쓴다만, push 가 쓰이는 것은 사실상 이곳 뿐이고, 기본적으로는 모두 pull request 의 형태로 처리한다.

Git 는 저장소 단위로 동기화가 기본이다. 다시 말해서, 모든 변경을 동기화 하도록 되어있다. origin 이라는 이름으로 중앙저장소-like 한 것이 존재하긴 하다만, 설정의 차이이지 기본적으로 구성은 전부 같다.

¦git pull¦ 은 사실 ¦git fetch ; git merge origin/master¦ 와 동일함을 염두에 두도록 하라. 이때, ¦git fetch¦ 는 단순히 외부 저장소(origin) 로부터 origin 이하의 branch 를 동기화 시켜서 가져오는 것이다.

저장소로 변경점을 보내기 (push)

push 는 외부 저장소에 로컬 저장소의 변경 사항을 보내서 그것을 동기화 하도록 하는 것이다. 다만, push 는 어떤 의미로는 외부 저장소가 로컬을 pull 하도록 강제하는 것과 같기 때문에, confliction 의 해결이 전제되어야 한다. 따라서 실제로는 pull, commit 을 한번 하고 난 뒤에 push 를 해주는 것이 보통이다.

```
: git pull      # 일단 외부 저장소에서 변경 사항이 없는지 확인하고
: edit ...      # 변경 사항들을 확인하고 confliction 등을 해결해주고
: git commit -a  # 커밋하여 외부 저장소와 confliction 을 해소한다.
: git push      # 변경 사항을 외부 저장소에 push 한다.
```
