

1 Git & GitHub 튜토리얼

출처: <http://riniblog.egloos.com/viewer/1024993> [Studio Rini]

Git 을 보통 어떻게 사용하는지 간략한 Flow 를 보겠습니다.

1. 새 프로젝트를 생성, 프로젝트 폴더에 git 을 설정한다.
2. 개발한다. 개발 중간중간에 개발한 내용을 commit 한다.
3. 버전 1 을 완성한다.
4. 새 기능을 넣는 사람들은 feature branch 를, 버그를 고치는 사람들은 bugfix branch 를... 등등 branch 를 나누어 개발한다.
5. 심각한 버그가 발생하면 별도의 hotfix branch 를 만들어서 빠르게 고친다.
6. 각 branch 의 개발이 끝나면 master branch 로 merge 한다.

꼭 이렇게 사용한다는 보장은 없지만 어떤 느낌으로 사용해야할 지 감은 오실 거라 생각합니다.

Git 설치

윈도우즈 기준으로 설명합니다.

GitHub 계정이 있는 경우 (또는 만들 생각이 있을 경우)

- [Windows 용 GitHub](#) 를 설치한다.
- GitHub 를 실행하고 로그인한다. (계정이 없으면 회원가입한다.)
- 로그인되면 Git Shell 을 연다.

GitHub 계정이 없는 경우

- [msysgit](#) 을 설치한다.
- Git Bash 를 실행한다.
- 다음을 적는다.

```
git config --global user.name <이름>
```

```
git config --global user.email <이메일>
```

※ 공백이 있는 문자열을 인수로 적을 때는 따옴표를 사용합니다.

```
ex) git config --global user.name "John Doe"
```

Shell 명령어

Git 은 기본적으로 shell 에서 실행됩니다.

그러므로 기초적인 명령어들은 알아두는 것이 좋습니다.

- ls

현재 디렉토리에 있는 파일 및 디렉토리들을 출력합니다.

-a 옵션으로 숨김 파일을 볼 수 있습니다.

- pwd (Print Working Directory)

현재 디렉토리 경로를 출력합니다.

- cd (Change Directory)

현재 디렉토리를 변경합니다. 이동할 경로를 입력받습니다.

경로에서 .은 현재 디렉토리, ..은 상위 디렉토리를 의미합니다.

- touch

파일을 생성합니다. 파일 명을 입력받습니다.

- mkdir (MaKe DIRectory)

디렉토리를 생성합니다. 디렉토리 명을 입력받습니다.

- rm (ReMove)

파일을 제거합니다. 제거할 파일 명을 입력받습니다.

와일드 카드 문자를 사용할 수 있습니다. ?는 임의의 한 문자, *은 임의의 문자열입니다.

-f 옵션을 주면 파일을 강제로 지웁니다.

-r 옵션을 주면 디렉토리를 지울 때, 그 서브 디렉토리까지 재귀적으로 제거합니다.

그래서 rm -rf 는 상당히 위험한 명령어로 언급됩니다. (?)

- rmdir (ReMove DIRectory)

디렉토리를 제거합니다. 디렉토리 명을 입력받습니다.

디렉토리가 비어있지 않으면 지울 수 없습니다.

- mv (MoVe)

파일/디렉토리를 이동시키거나 이름을 변경합니다.

이동할 파일/디렉토리 명과 이동될 경로를 입력합니다.

또는 기존 이름과 바꿀 이름을 입력합니다.

- vi, vim

vi, vim 에디터를 엽니다.

vi, vim 은 유닉스에서 사용하는 텍스트 에디터입니다.

GitHub 의 Git Shell 에서는 vi, vim 을 쓸 수 없습니다.

vi, vim 의 단축키는 [여기](#)를 참조하세요.

※ 파일 명이나 경로 명을 입력하는데 전체 이름이 기억이 나지 않을 때, 적당히 친 다음 Tab 을 누르면 자동으로 완성해줍니다. 비슷한 이름이 여러 개 있을 경우 Tab 을 여러 번 누르면 다른 이름으로 바뀝니다.

Git 사용하기

Git 을 본격적으로 사용해보겠습니다.

초기화

cd 와 mkdir 로 적당한 곳에 디렉토리를 만들고 들어갑니다.

저는 홈 디렉토리(윈도우즈에서 사용자 디렉토리)에 hellogit 이라는 디렉토리를 만들었습니다.

디렉토리 안에서 다음 명령어를 칩니다.

```
git init
```

이제 이 디렉토리는 git 으로 관리할 수 있습니다.

※ 이렇게 하면 디렉토리에 .git 이라는 폴더가 생성됩니다. 여기에 git 의 데이터가 저장됩니다. ls -a 를 해보세요.

관리할 파일 추가

디렉토리에 새로운 파일을 추가합니다.

```
touch README.md  
git add README.md
```

README.md 라는 파일을 추가해주었습니다.

그리고 그 파일을 git 에서 관리하도록 add 시켜주었습니다.

(이렇게 새 파일을 add 해주는 것을 추적(Track)이라고 합니다.)

```
git status
```

이 명령어를 치면 현재 상태를 볼 수 있습니다.

만일 add 되지 않은 파일이 있다면 붉은색으로 Untracked files 또는 Changes not staged for commit 목록이 나옵니다.

README.md 는 add 되었기 때문에 초록색으로 Changes to be committed 목록에 나옵니다.

만일 add 할 파일이 많은 경우 그냥 git add *이라고 적으면 모든 파일, 디렉토리가 add 됩니다.

git add *라고 적어도 몇 가지 쓸모없는 파일들은 add 되지 않도록 하고 싶다면, .gitignore 라는 이름의 파일을 만들어야 합니다.

파일의 내용은 무시할 파일명 패턴들입니다.

```
# 이 문자로 시작하면 주석행  
# 특정 확장자 파일들 무시  
*.sln  
*.sdf
```

```
# 특정 디렉토리 내 파일들 무시
Debug/
Release/
Project/*.vcproj
# []를 쓰면 괄호 안의 문자들 중 하나를 의미
[Bb]in/           # Bin 또는 bin
```

많이 쓰이는 종류의 .gitignore 은 인터넷에 찾으면 많이 나오고, gitignore.io 같은 사이트도 있습니다.

커밋 Commit

관리할 파일들을 모두 add 시켜놓았으면, 이제 커밋을 해봅시다.

커밋은 간단히 말해 현재 상태를 저장하는 것입니다.

```
git commit
```

이렇게 적으면 vim 이 열리면서 커밋 메시지를 입력받습니다.

i 를 눌러 편집 모드에 들어가고, 메시지를 입력한 다음, Esc 를 누르고, :wq 로 저장 후 빠져나오기를 합니다.

이 과정이 번거롭기 때문에 보통은 이렇게 사용합니다.

```
git commit -m <커밋 메시지>
```

git commit -m "First commit" 이런 식으로 커밋하시면 됩니다.

커밋 메시지는 상세할수록 좋습니다.

git log 로 커밋 로그를 볼 수도 있습니다.

커밋한 이후에 개발하면서 파일이 변경되면 또 git add 를 해주어야 합니다.

Tracked 는 된 상태지만 Staged 는 되지 않은 상태이기 때문입니다.

git add 는 파일을 Tracked 상태로 만들기도 하지만 동시에 Staged 상태로 만들기도 합니다.

커밋은 Staged 된 파일만 저장하기 때문에 git add 를 하지 않으면 변경된 내용이 없다고 판단해버립니다.

브랜치 Branch

개발을 하다보면 기존에 개발하던 부분과 다른 부분의 개발을 해야하는 상황이 발생합니다.

예를 들어 기존 코드를 리팩토링한다고 생각해봅시다. 리팩토링이 끝나기 전까지는 프로그램에 기능을 추가하는 것이 어렵습니다.

만일 여러 명이서 작업하고 있었다면, 이것은 전체가 올스톱되는 상황을 만듭니다.

이럴 때, 서로 독립적으로 평행하게 개발을 진행하기 위해서 브랜치를 만듭니다.

한 브랜치는 기능을 추가하는 개발을 하고,

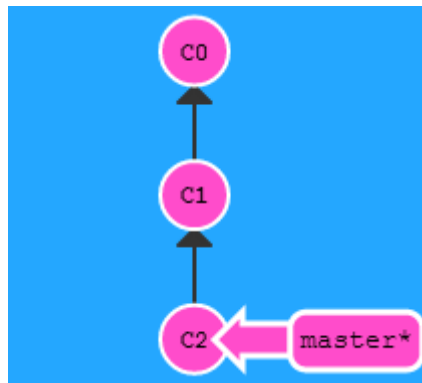
한 브랜치는 기존 코드를 리팩토링하는 개발을 진행하는 것입니다.

브랜치가 나뉘어있으면, 둘 사이의 충돌이 일어나지 않습니다. 일종의 평행세계입니다.

물론 나중에 하나로 묶는 과정에서 충돌이 일어나기도 하지만 그 전까지는 평화롭습니다. (?)

기본적으로 git 을 초기화하면 master 브랜치가 생깁니다. 그리고 commit 을 하면 이 master 브랜치에서 커밋이 일어납니다.

그림으로는 이렇습니다.

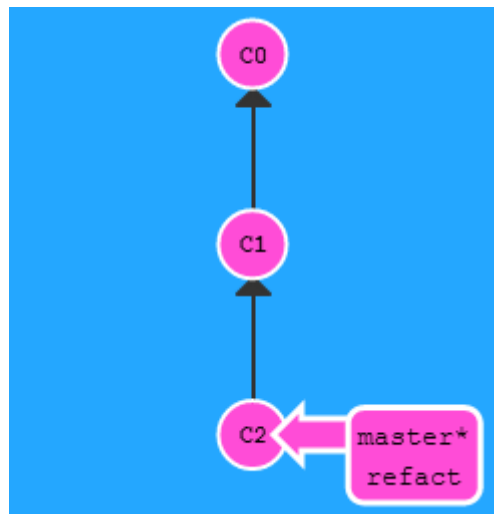


커밋을 하면 이전 커밋에서 바뀐 부분만 저장됩니다. 그래서 이전 커밋을 이렇게 줄줄이 소시지처럼 가르키게 됩니다.

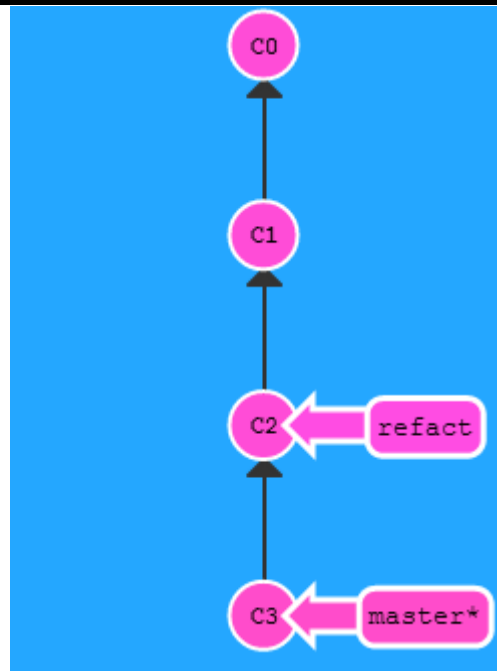
새 브랜치를 만들어봅시다.

`git branch refactor`

브랜치를 만들면 현재 브랜치와 같은 곳에 브랜치가 생성됩니다.



현재 브랜치는 master 로 유지됩니다. 이 상태에서 커밋을 하면

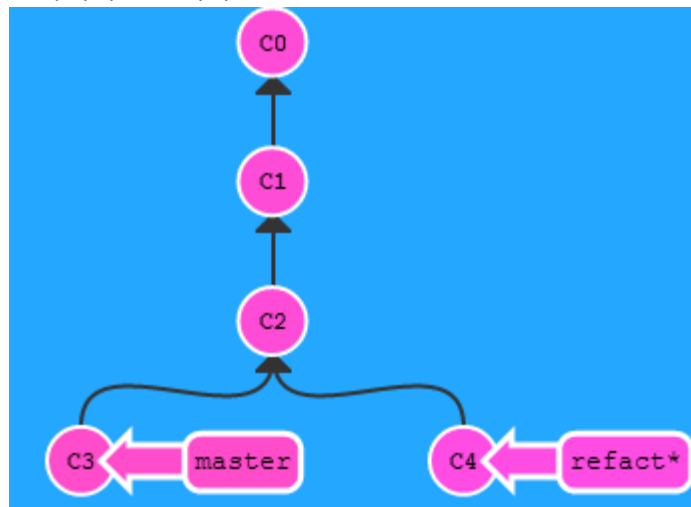


이렇게 나누어집니다.

현재 브랜치를 refactor 브랜치로 바꾸어봅시다.

```
git checkout refactor
```

브랜치를 바꾸고 커밋하면 아래처럼 됩니다.



둘은 서로 독립적입니다. master 에서 변경이 일어나도 refactor 에는 전혀 영향을 주지 못합니다. 그 반대로 마찬가지입니다.

여담으로 브랜치를 만들고 checkout 하는 것을 한 번에 하는 방법이 있습니다.

```
git checkout -b <브랜치 이름>
```

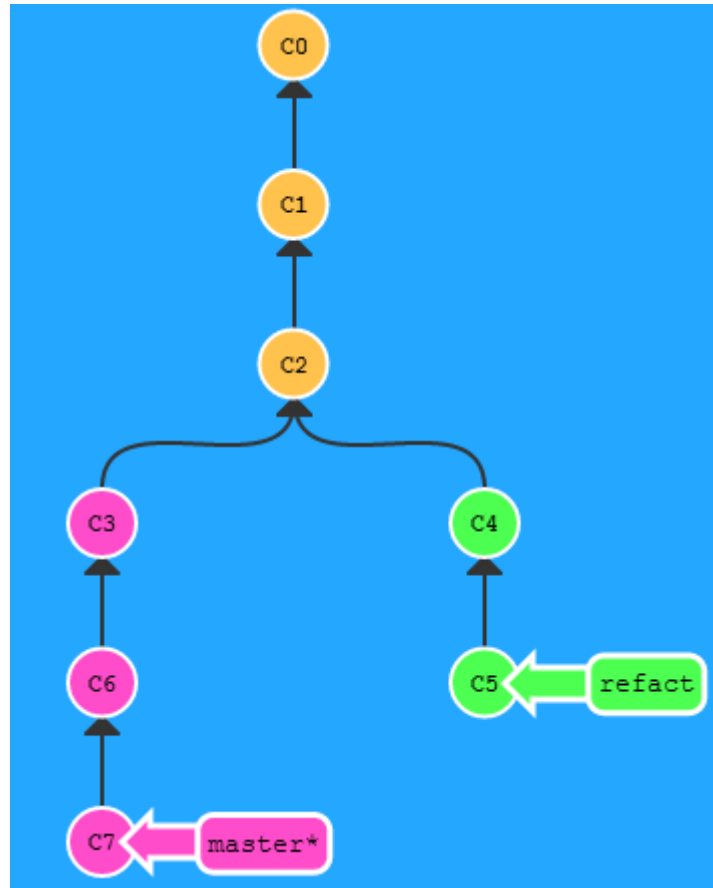
브랜치를 지우고 싶을 경우에는 -d 를 씁니다.

```
git branch -d <브랜치 이름>
```

병합 merge

이렇게 브랜치를 나누어 개발하는 중에 한 브랜치의 개발이 끝났습니다. 그리고 기존 브랜치에 그 개발상황을 적용시키려고 합니다.

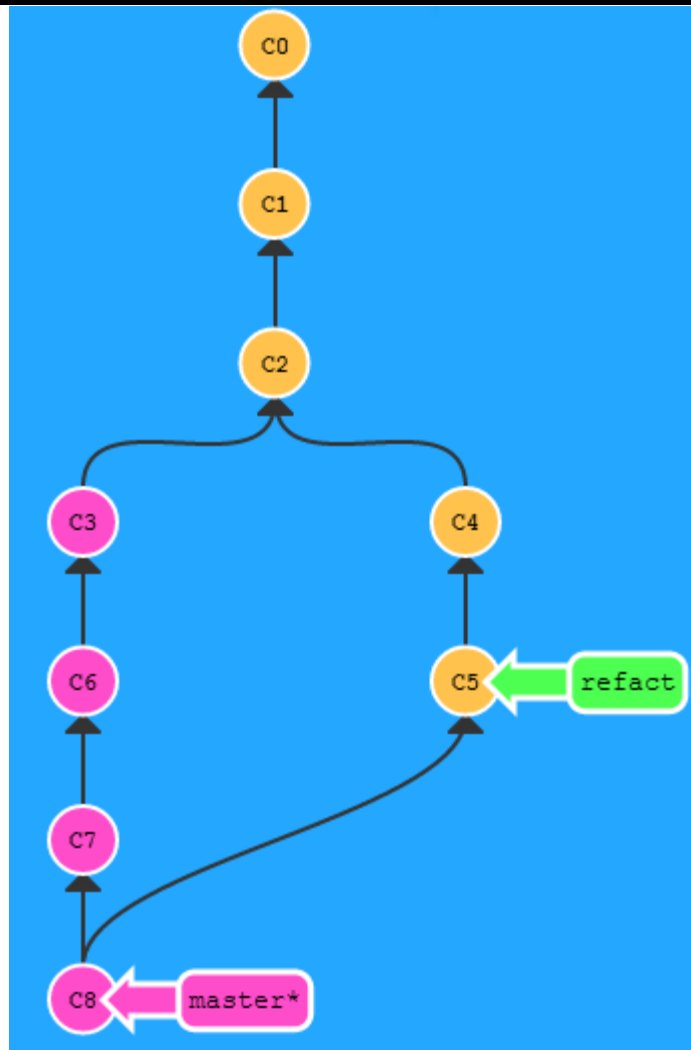
이럴 때 사용하는 것이 merge 입니다.



master 브랜치에 refact 브랜치를 병합하려고 합니다.

이럴 땐 현재 브랜치가 master 브랜치인 상태에서 다음 명령어를 씁니다.

```
git merge refact
```



새로운 커밋이 생성되며, 이는 refactor 의 내용과 master 의 내용이 합해진 형태가 됩니다.

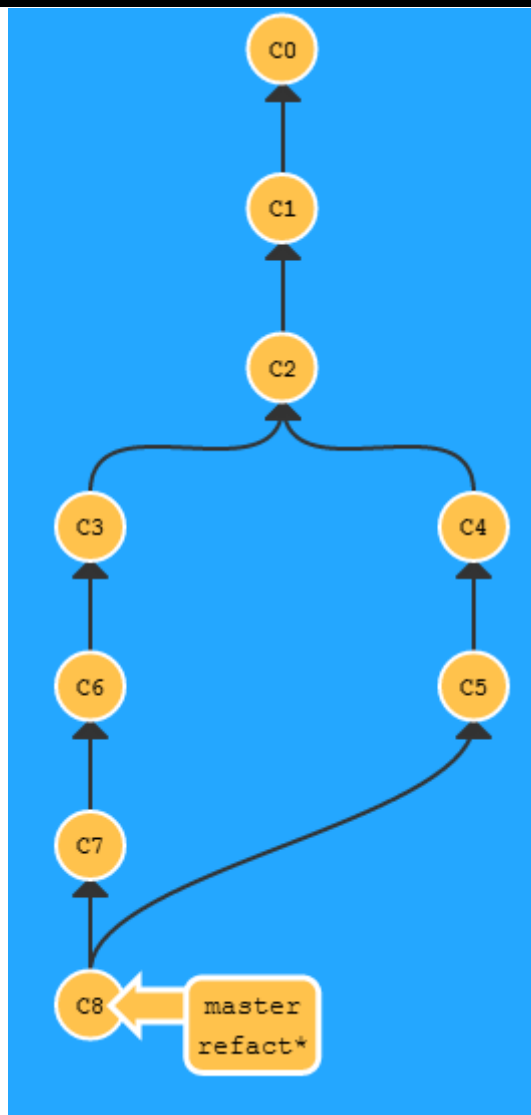
이 과정에서 충돌이 일어나기도 합니다.

예를 들어 a.txt 가 master, refactor 둘 모두에서 변경되었다면, 두 버전 중 하나를 선택해주어야 합니다.

a.txt 를 열면 충돌된 부분이 표시되어있기 때문에 그 부분을 수정하고 다시 `git add a.txt` 를 해서 커밋하면 됩니다.

(새로운 커밋이 생기지는 않고 머지 커밋이 갱신됩니다.)

여담으로 이 상태에서 refactor 브랜치에 master 를 merge 하면 이렇게 됩니다.



refect 가 master 가 있는 곳으로 이동합니다.

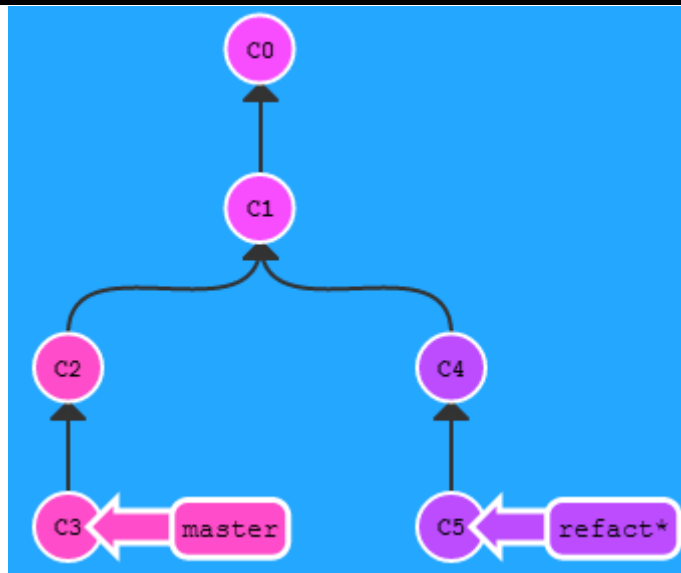
이를 fast-forward 라고 합니다.

리베이스 Rebase

merge 는 보시다시피 병합된 브랜치의 커밋 히스토리가 남습니다.

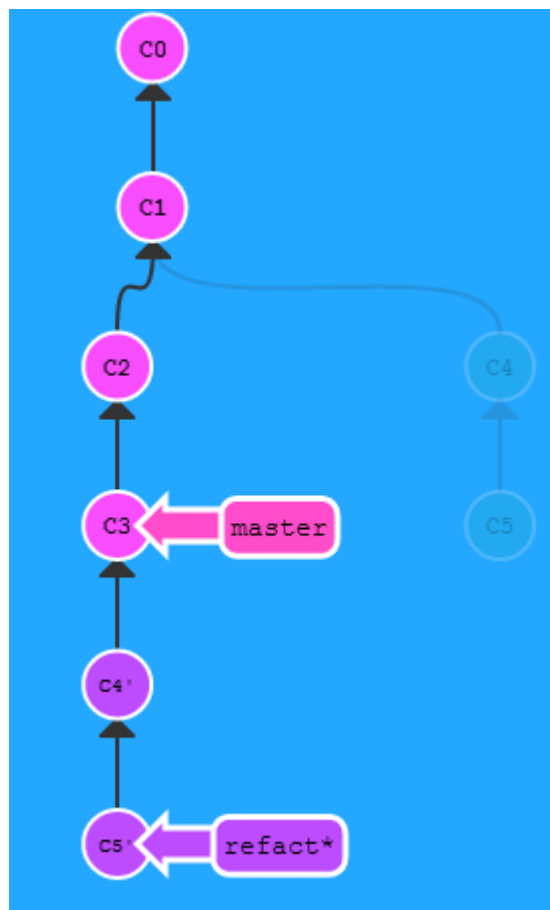
이렇게 merge 를 계속 하다보면 커밋 히스토리가 지저분해집니다.

rebase 는 merge 와 똑같이 브랜치를 병합하지만 히스토리가 일자로 깨끗합니다.

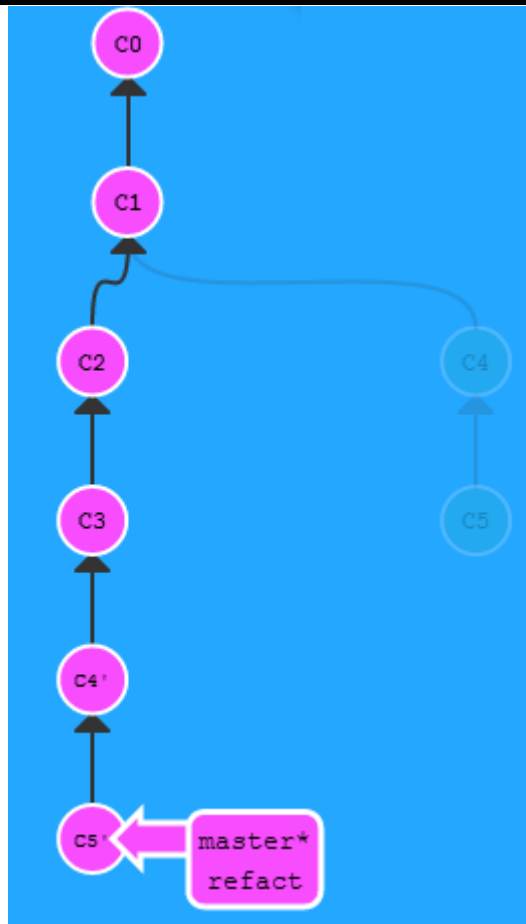


좀전과 같은 상황입니다. refactor 브랜치를 선택한 상태에서 다음 명령어를 입력합니다.

```
git rebase master
```



이렇게 하면 master 브랜치 작업 이후에 refactor 작업을 한 것처럼 병합이 이루어집니다.
master 브랜치를 fast-forward 하면



이렇게 됩니다. merge 보다 깔끔합니다. 그래서 여러 경우에는 merge 보다 rebase 를 더 추천합니다.
(자세한 내용은 저도 아직 잘 모르겠습니다.)

복구

개발 도중에는 코드가 엉켜서 도저히 나아가기 힘든 상황에 놓이기도 합니다.
이럴 때 git 을 사용해 과거의 코드를 다시 가져올 수 있습니다.

```
git checkout [<커밋 이름>] -- <파일 명>
```

[]는 생략 가능한 부분입니다.

커밋 이름을 생략하면, 현재 가르키는중인 커밋(HEAD)을 사용합니다.

커밋 이름은 `git log` 를 하면 나오는 커밋의 해시값입니다.

전부 다 적을 필요는 없고 앞에 네 글자만 적어도 대부분 됩니다.

해시값 말고 branch 이름도 가능합니다.

그 외에 HEAD 라는 상수도 사용할 수 있는데, 이는 현재 가르키고 있는 커밋을 말합니다.

커밋 이름 뒤에 ^를 붙이면 그 커밋의 부모(바로 전) 커밋을 의미하고,

~<숫자>를 붙이면 그 커밋의 <숫자>번째 부모(전) 커밋을 의미합니다.

예를 들어 최근 커밋한 a.txt 로 복구하고 싶다면 `git checkout -- a.txt` 라고 적으면 됩니다.

또는 f5d4 커밋때의 모든 파일을 가져오고 싶다면 `git checkout f5d4 -- *`라고 적으면 됩니다.

커밋 자체를 취소하고 싶을 경우도 있습니다.

- 방금 한 커밋을 수정하고 싶을 경우

```
git commit --amend
```

메시지를 잘못 적었거나, 깜빡하고 add 하지 못한 파일이 있거나 할 때,

git add 를 하고나서 git commit --amend -m <새 메시지>를 적으면 방금 했던 커밋이 수정됩니다.

- 커밋 자체를 취소하고 싶은 경우

```
git reset <돌아갈 커밋 이름>
```

```
git revert <취소할 커밋>
```

reset 과 revert 의 작동 방식은 조금 다릅니다. reset 은 돌아갈 커밋 이름으로 적은 커밋으로 롤백합니다.

바로 전으로 돌아가려면 git reset HEAD^나 git reset HEAD~1 을 적으면 됩니다.

revert 는 취소할 커밋의 변경 내용을 정확히 거꾸로 수행한 새로운 커밋을 만듭니다.

결과적으로는 둘 다 커밋하기 전 상태로 돌아갑니다.

대신 reset 은 커밋했던 히스토리가 남지 않을 수도 있습니다.

revert 는 커밋했던 히스토리가 남습니다.

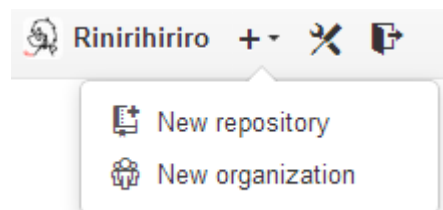
보통 원격 저장소에는 reset 대신 revert 를 씁니다.

** GitHub

오래 기다렸습니다. 드디어 Git 으로 GitHub 를 사용하는 방법을 알아보겠습니다.



GitHub 가 아닌 다른 서비스라고 해도 사용 방법은 크게 다르지 않습니다.

Repository 생성




로그인을 하고 New repository 로 새로운 저장소를 만듭니다.


Owner **Repository name**

PUBLIC   **Rinirihiriro** / **GitTutorial** ✓

Great repository names are short and memorable. Need inspiration? How a

Description (optional)

☒  **Public**
Anyone can see this repository. You choose who can commit.

☐  **Private**
You choose who can see and commit to this repository.

☒ **Initialize this repository with a README**
This will allow you to `git clone` the repository immediately. Skip this step if

Add .gitignore: **None** | Add a license: **None** ⓘ

Create repository

저장소의 이름을 정하고, Description 에는 필요한 설명을 적습니다.

Github 에서 Private(개인) 저장소를 만드려면 프리미엄 계정으로 업그레이드해야합니다.

우리는 GitHub 에서 만든 저장소를 가져올 것이기 때문에 README 를 생성하도록 하겠습니다.

.gitignore 와 라이선스는 자신의 입맛에 따라 선택하시면 됩니다.

Create repository 를 누르면 저장소가 만들어집니다.

원격 저장소 가져오기

이렇게 만든 저장소를 이용하려면 우선 우리 컴퓨터로 가져와야합니다. 이를 clone 이라고 합니다. shell 을 열고 적당한 곳에 가서 아래 명령을 칩니다.

```
git clone <저장소 URL>
```

저장소 URL 은 오른쪽 아래에 있습니다.

HTTPS clone URL

https://github.com

You can clone with [HTTPS](#), [SSH](#),
or [Subversion](#). ⓘ

Clone in Desktop



Download ZIP

clone 하게되면 해당 디렉토리에 저장소 이름과 똑같은 폴더가 생성됩니다. 안에는 이미 git 이 설정되어있는 저장소가 그대로 복사되어 들어와 있습니다.

만약 윈도우즈용 GitHub 를 설치했다면 아래의 Clone in Desktop 을 클릭해서 바로 clone 할 수도 있습니다. 이 때 디폴트 경로는 사용자 디렉토리 아래의 Documents 아래의 GitHub 입니다.

원격 저장소에 저장하기

컴퓨터에서 한동안 작업하면서 커밋도 하고 이것저것 했을 겁니다. 그럼 그 결과를 다시 원격 저장소에 보내주어야 합니다.

```
git push <원격 저장소 이름> <보내줄 브랜치>
```

우리는 원격 저장소를 가져온 경우이기 때문에 원격 저장소 이름은 기본적으로 origin 으로 설정되어있습니다.

보내줄 브랜치는 보통 master 를 적고, 다른 브랜치를 작업중이었다면 그 브랜치를 적어줍니다.

이 경우에는 git push 라고만 적어도 작동합니다.

이렇게 하면 저장소에 지금까지의 내역이 저장됩니다.

※ GitHub 로 설치한 shell 에서는 문제 없이 push 되지만 msysgit 으로 push 할 경우 github 계정을 인증하는 단계가 필요합니다.

한 가지 중요한 점은, 누군가가 push 를 했다면 다른 사람들은 push 를 할 수 없습니다. push 를 하려면 업데이트된 저장소를 불러온 다음 병합해서 push 해야합니다.

원격 저장소에서 업데이트 받기

우리가 우리 컴퓨터에서 작업하는 동안에 다른 사람이 작업한 내용을 원격 저장소에 push 할 수도 있습니다. 이렇게 되면 원격 저장소 내용을 받아와서 업데이트를 해주어야 합니다.

이 때 사용하는 명령어는 pull 입니다.

```
git pull
```

pull 을 하면 fetch 와 merge 가 이루어집니다.

fetch 는 원격 저장소의 데이터를 가져오는 작업입니다.

git fetch 만 실행하면 새로운 데이터를 가져오고 origin/master 브랜치가 생깁니다.

이 브랜치는 원격 저장소의 master 를 나타내며, 변경할 수는 없고 참조만 가능합니다.

보통은 이 원격 저장소의 master 와 이 컴퓨터의 master 를 merge 해서 계속 작업합니다.

pull 했을 때 일어나는 충돌은 merge 와 똑같은 방식으로 처리합니다.

※ git log --oneline --decorate 를 입력하면 로그를 조금 더 예쁘게(?) 볼 수 있습니다.

--decorate 옵션을 사용하면 HEAD, origin/master, master 등의 브랜치와 태그들을 볼 수 있습니다.

로컬 저장소를 원격 저장소로 만들 때

방금은 원격 저장소를 만든 다음 로컬 저장소로 복사해왔습니다.

이번에는 반대로 로컬 저장소를 만들고, 그것을 원격 저장소로 전송하겠습니다.

만일 git init 으로 저장소를 만든 경우에는 원격 저장소 이름을 설정해주어야 합니다.

```
git remote
```

이것만 입력하면 원격 저장소 목록이 나옵니다.

```
git remote add <저장소 이름> <저장소 URL>
```

이렇게 입력하면 새로운 저장소를 추가할 수 있습니다.

그 다음 push 합니다.

```
git push -u <저장소 이름> <보내줄 브랜치>
```

여기서 -u 옵션을 쓴 이유는, 이렇게 하면 git push 와 git pull 을 인수 없이 사용할 수 있기 때문입니다.

※ push 를 하려면 적어도 한 번은 커밋을 해야합니다.

이 이후는 원격저장소를 가져온 경우와 똑같이 활용 가능합니다.