|  |  |
| --- | --- |
| **Program / Project Name:** | Git |
| **Checklist / Template Completed by:** | TBD |
| **Date Completed:** | Click here to enter a date. |

**Tips for Git**

Revision: 0.1

**Detailed Revision History**

|  |  |  |  |
| --- | --- | --- | --- |
| **Rev** | **Date** | **Editor(s)** | **Description of change** |
| 0.1 | 20-MAY-2019 | Sang-Gu Kang | Initial Draft |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

**Table of Contents**

[1 Install 5](#_Toc29491382)

[1.1 Git 5](#_Toc29491383)

[1.2 TortoiseGit 5](#_Toc29491384)

[1.2.1 Download 5](#_Toc29491385)

[1.2.2 Install 5](#_Toc29491386)

[2 사용법 10](#_Toc29491387)

[2.1 Command 10](#_Toc29491388)

[2.2 History 관리(롤백, 리버트, 리셋, 리베이스) 23](#_Toc29491389)

[2.2.1 정확한 작업 취소 방법 선택 23](#_Toc29491390)

[2.2.2 적절한 명령어 선택 플로우 차트 24](#_Toc29491391)

[2.2.3 실험 작업을 위한 브랜치 사용 24](#_Toc29491392)

[2.2.4 단계적 리베이싱 26](#_Toc29491393)

[2.2.5 잃어버린 작업 찾기 개요 28](#_Toc29491394)

[2.2.6 파일 복구하기 30](#_Toc29491395)

[2.2.7 커밋 작업하기 33](#_Toc29491396)

[2.3 상황별 용도 38](#_Toc29491397)

[2.3.1 리모트 저장소 38](#_Toc29491398)

[2.3.2 태그 38](#_Toc29491399)

[2.3.3 브랜치 39](#_Toc29491400)

[2.3.4 되돌리기 Reset과 checkout 39](#_Toc29491401)

[2.3.5 원격 브랜치 삭제 40](#_Toc29491402)

[2.4 브랜치 워크플로우 40](#_Toc29491403)

[2.4.1 Long-Running 브랜치 40](#_Toc29491404)

[2.4.2 토픽 브랜치 40](#_Toc29491405)

[3 Definition and Explanation 41](#_Toc29491406)

[3.1 HEAD 41](#_Toc29491407)

[4 Tools 41](#_Toc29491408)

[4.1 Beyond Compare 41](#_Toc29491409)

[5 Useful Tip 41](#_Toc29491410)

[5.1 Overlay Icon 보이게 하기 41](#_Toc29491411)

# What is Git

## 스테이지와 커밋



# Install

## TortoiseGit

TortoiseGit 설치파일과 언어팩을 다운로드한 후 설치한다.

한글 설정을 놓쳤다면 settings에서 설정해 준다.



사용자 정보를 설정해 준다



# 사용법

## 작업 되돌리기

### 작업트리에서 수정한 파일 되돌리기

git checkout -- <file name>

### staging 되돌리기

git reset HEAD <file name>

HEAD 다음에 파일 이름을 지정하지 않으면 스테이지에 있는 모든 파일을 되돌린다.

### 최신 커밋 되돌리기

git reset HEAD^

HEAD^는 현재 HEAD가 가리키는 브랜치의 최신 커밋을 가리킨다. 최근 3개의 커밋을 취소하려면 $git reset HEAD~3처럼 사용한다. 커밋 전에 했던 스테이징도 함께 취소가 된다.

**Tips>** git reset 명령의 옵션  
reset 명령은 사용하는 옵션에 따라 되돌릴 수 있는 단계가 다르다.

|  |  |
| --- | --- |
| **명령** | **설명** |
| --soft HEAD^ | 최근 커밋을 하기 전 상태로 작업 트리를 되돌린다. **스테이징 상태**가 된다. |
| --mixed HEAD^ | 최근 커밋과 스테이징을 하기 전 상태로 작업트리를 되돌린다. 옵션 없이 git reset 명령을 사용할 경우 이 옵션을 기본으로 작동한다. 워킹 트리 상태가 된다. |
| --hard HEAD^ | 최근 커밋과 스테이징, **파일 수정을 하기 전 상태**로 작업 트리를 되돌린다. 이 옵션으로 **되돌린 내용은 복구할 수 없다**. |

### 특정 커밋으로 되돌리기

git reset <커밋 해시>

reset에서 커밋 해시를 사용해 되돌릴 때 주의해야 할 점이 있다. 예를 들어 reset A를 입력한다면 이 명령은 A 커밋을 삭제하는 것이 아니라 최근 커밋을 A로 리셋한다. 즉 A 커밋을 삭제하는 것이 아니라 A 커밋 이후에 만들었던 커밋을 삭제하고, A 커밋으로 이동하겠다는 의미이다. reset HEAD^가 최신 커밋을 삭제한다는 점에서 다르다.

### 커밋 삭제하지 않고 되돌리기

git revert

나중에 사용할 것을 대비해서 커밋을 되돌리더라도 취소한 커밋을 남겨두어야 할 때가 있는데 이때는 git reset이 아닌 git revert라는 명령을 사용한다.

reset의 경우에는 취소할 커밋의 해시가 아니라 되돌아갈 커밋 해시를 지정하고, revert의 경우에는 취소할 커밋 해시를 지정한다는 점이 다르다.

revert하면 새로운 버전(커밋 해시)이 생성되고 이전 버전(커밋 해시)들은 밑에 그대로 남게 되며, 당연히 새로운 버전에는 되돌리려는 커밋 이전 내용이 반영되어 있다.

### 실수로 reset한 경우 원복하기

git reset --hard HEAD~~

잘못 지웠음을 인지했다면 아래와 같이 해서 되돌릴 수 있다.

git reset --hard ORIG\_HEAD

## 브랜치 정보 확인하기

### 브랜치 정보 확인

--branches 옵션을 사용하면 각 브랜치의 커밋을 함께 볼 수 있다.

$ git log --oneline --branches  
b38f4fc (HEAD -> apple) apple content 4

2a428e9 (master) master content 4

d560214 (ms, google) work 3

8fbd8d9 work 2

1bfec66 work 1

브랜치와 커밋의 관계를 좀 더 보기 쉽게 그래프 형태로 표시하려면 git log 명령에 --graph 옵션을 함께 사용한다. apple 브랜치의 최신 커밋은 ‘apple content 4’인데, 점선을 따라 ‘apple content 4’ 커밋의 부모를 찾아가 보면 ‘work 3’ 커밋을 만나게 된다. 즉 apple 브랜치에서는 ‘work 3’ 커밋 다음에 ‘apple content 4’ 커밋이 만들어졌다는 뜻이다. master 브랜치까지 본다면, master 브랜치나 apple 브랜치는 ‘work 3’ 커밋까지는 같고 그 이후부터 브랜치마다 다른 커밋을 만들었다는 사실을 알 수 있다.  
$ git log --oneline --branches --graph

\* b38f4fc (HEAD -> apple) apple content 4

| \* 2a428e9 (master) master content 4

|/

\* d560214 (ms, google) work 3

\* 8fbd8d9 work 2

\* 1bfec66 work 1

### 브랜치 사이의 차이점 보기

브랜치마다 커밋이 점점 쌓여갈수록 브랜치 사이에 어떤 차이가 있는지 일일이 확인하기 어려워진다. 이때 브랜치 이름 사이에 마침표 두개를 넣는 명령으로 차이점을 쉽게 확인할 수 있다. 마침표 왼쪽에 있는 브랜치를 기준으로 오른쪽 브랜치와 비교한다. 브랜치 이름과 마침표는 공백 없이 이어준다.

$ git log master..apple

commit b38f4fc4752f7766e74b84dedb42093d658dbf1c (HEAD -> apple)

Author: SangguKang <kangsanggu74@gmail.com>

Date: Thu Feb 20 19:46:07 2020 +0900

apple content 4

이렇게 하면 master 브랜치에는 없고 apple 브랜치에만 있는 커밋 즉, ‘apple content 4’ 커밋을 보여준다.

$ git log apple..master

commit 2a428e94494327c1f8c90e1d81fe5d75856349a3 (master)

Author: SangguKang <kangsanggu74@gmail.com>

Date: Thu Feb 20 19:13:54 2020 +0900

master content 4

반대로 apple 브랜치를 기준으로 master와 비교하면 apple 브랜치에는 없고 master 브랜치에만 있는 ‘master content 4’ 커밋을 보여준다.

## 브랜치 병합하기

### 서로 다른 파일 병합

$ git br

master

\* o2

$ git log --branches --graph --oneline

\* 9dcecfb (HEAD -> o2) o2 work 2

| \* 27b0246 (master) master work 2

|/

\* 1d4f12f work 1

$ git checkout master

Switched to branch 'master'

$ git merge o2

Merge made by the 'recursive' strategy.

o2.txt | 1 +

1 file changed, 1 insertion(+)

create mode 100644 o2.txt

$ git br

\* master

o2

$ git log --oneline --branches --graph

\* 57b5c87 (HEAD -> master) Merge branch 'o2'

|\

| \* 9dcecfb (o2) o2 work 2

\* | 27b0246 master work 2

|/

\* 1d4f12f work 1

$ ll

total 3

-rw-r--r-- 1 MSI 01 197121 10 2월 21 06:37 master.txt

-rw-r--r-- 1 MSI 01 197121 6 2월 21 06:38 o2.txt

-rw-r--r-- 1 MSI 01 197121 2 2월 21 06:26 work.txt

Tips> master에서 분기한 브랜치에서 작업을 한 후 master에서 merge를 할 때 master 브랜치에 아무 변화가 없다면 fast-forward라는 메시지가 나타나며, 따로 커밋 메시지 창이 열리지 않는다. 이것을 빨리 감기 병합(fast-forward merge)이라고 한다.

### 같은 문서의 다른 위치를 수정했을 때 병합

각 브랜치에서 같은 파일의 같은 줄을 수정했을 때 브랜치를 병합하면 충돌이 발생한다.

kan63697@19WCP72 MINGW64 /c/Works/02\_project\_ind/41\_Utils/01\_Git/manual-4 (master)

$ git merge o2

Auto-merging work.txt

CONFLICT (content): Merge conflict in work.txt

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

충돌이 생긴 문서는 자동으로 병합될 수 없으므로 사용자가 직접 충돌 부분을 해결한 후 커밋해야 한다.

kan63697@19WCP72 MINGW64 /c/Works/02\_project\_ind/41\_Utils/01\_Git/manual-4 (master|MERGING)

$ vim work.txt

#title

content

<<<<<<< HEAD

master content 2

=======

o2 content 2

>>>>>>> o2

#title

content

<<<<<<< HEAD와 가운데 가로줄 ======= 사이의 내용은 현재 브랜치, 즉 master 브랜치에서 수정한 내용이다. 가로줄 =======과 >>>>>>> o2 사이의 내용은 o2 브랜치에서 수정한 내용이다. 양쪽 브랜치의 내용을 참고하면서 직접 수정해야 한다.

Tips> 무료 병합 및 충돌 해결 프로그램

|  |  |
| --- | --- |
| 프로그램 이름 | 설명 |
| P4Merge | 무료익 직관적이며 사용이 편리하고 병합 기능이 뛰어나다. 단축키가 지원되지 않는 단점이 있다. |
| Meld | 무료에 오픈 소스이다. 파일을 비교하는 것뿐만 아니라 직접 편집할 수 있다. |
| Kdiff3 | 무료이고 사용이 편리하고 병합 기능이 뛰어나지만 한글이 깨져보일 수 있다. |

### 병합이 끝난 브랜치 삭제하기

브랜치를 삭제하려면 master 브랜치에서 해야 한다.

kan63697@19WCP72 MINGW64 /c/Works/02\_project\_ind/41\_Utils/01\_Git/manual-4 (master)

$ git br -d o2

Deleted branch o2 (was 2923745).

master 브랜치에 병합하지 않은 브랜치를 삭제하려면 오류 메시지가 나타난다. 이럴 경우 옵션 -d 대신 -D를 사용해서강제로 브랜치를 삭제할 수 있다.

삭제한 브랜치는 같은 이름으로 다시 브랜치를 만들면 예전에 작업했던 내용이 그대로 나타난다. 즉, 브랜치를 삭제한다는 것은 완전히 저장소에서 없애는 것이 아니라 깃의 흐름 속에서 감추는 것이다.

## 브랜치 관리하기

### 브랜치에서의 checkout과 reset

$ git init test

Initialized empty Git repository in C:/Works/02\_project\_ind/41\_Utils/01\_Git/test/.git/

$ cd test/

$ vim c1.txt

$ git add .

warning: LF will be replaced by CRLF in c1.txt.

The file will have its original line endings in your working directory

$ git commit -m "c1"

[master (root-commit) 497eeb3] c1

1 file changed, 1 insertion(+)

create mode 100644 c1.txt

$ git log

commit 497eeb3efc7b4ab0be37ab2e2af6b254fbff638d (HEAD -> master)

Author: SangguKang <kangsanggu74@gmail.com>

Date: Fri Feb 21 16:18:22 2020 +0900

c1

$ git branch sub

$ git log --oneline

497eeb3 (HEAD -> master, sub) c1

$ vim c2.txt

$ git add .

warning: LF will be replaced by CRLF in c2.txt.

The file will have its original line endings in your working directory

$ git commit -m "c2"

[master 7233345] c2

1 file changed, 1 insertion(+)

create mode 100644 c2.txt

$ git commit -m "c2"

[master 7233345] c2

1 file changed, 1 insertion(+)

create mode 100644 c2.txt

$ git log --oneline

7233345 (HEAD -> master) c2

497eeb3 (sub) c1

$ git checkout sub

gitSwitched to branch 'sub'

$ git log --oneline

497eeb3 (HEAD -> sub) c1

$ vim s1.txt

$ git add .

warning: LF will be replaced by CRLF in s1.txt.

The file will have its original line endings in your working directory

$ git commit -m "s1"

[sub 2355447] s1

1 file changed, 1 insertion(+)

create mode 100644 s1.txt

$ git log --oneline

2355447 (HEAD -> sub) s1

497eeb3 c1

$ git log --oneline --branches

2355447 (HEAD -> sub) s1

7233345 (master) c2

497eeb3 c1

$ git reset 7233345

Unstaged changes after reset:

D c2.txt

$ git log --oneline --branches --graph

\* 7233345 (HEAD -> sub, master) c2

\* 497eeb3 c1

git checkout 명령을 사용하면 HEAD를 제어해서 브랜치를 이동할 수 있다. git reset 명령을 사용하면 HEAD가 가리키고 있는 브랜치의 최신 커밋을 원하는 커밋으로 지정할 수 있다. 이때 어떤 브랜치에 있는 커밋이든 지정할 수 있으며, 명령을 수행한 뒤 브랜치와 연결이 끊긴 커밋은 삭제된다.

### 수정중인 파일 감추기 및 되돌리기

git stash

브랜치에서 파일을 수정하고 커밋하지 않은 상태에서 급하게 다른 파일을 커밋해야 할 경우가 있다. 이럴 때 아직 커밋하지 않고 작업 중인 파일들을 잠시 감춰둘 수 있다. 그리고 당장 필요한 작업들을 끝낸 후 다시 감춰둔 파일들을 꺼내 오면 된다.

$ git init st

Initialized empty Git repository in C:/Works/02\_project\_ind/41\_Utils/01\_Git/st/.git/

$ cd st/

$ vim f1.txt

$ git add .

warning: LF will be replaced by CRLF in f1.txt.

The file will have its original line endings in your working directory

$ git commit -m "f1"

[master (root-commit) 4891f0d] f1

1 file changed, 1 insertion(+)

create mode 100644 f1.txt

$ vim f2.txt

$ git add .

warning: LF will be replaced by CRLF in f2.txt.

The file will have its original line endings in your working directory

$ git commit -m "f2"

[master 3d969f0] f2

1 file changed, 1 insertion(+)

create mode 100644 f2.txt

$ vim f1.txt

$ vim f2.txt

$ git st

On branch 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: f1.txt

modified: f2.txt

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

$ git stash

warning: LF will be replaced by CRLF in f1.txt.

The file will have its original line endings in your working directory

warning: LF will be replaced by CRLF in f2.txt.

The file will have its original line endings in your working directory

Saved working directory and index state WIP on master: 3d969f0 f2

On branch master

nothing to commit, working tree clean

$ ll

total 2

-rw-r--r-- 1 kan63697 1049089 4 Feb 21 19:09 f1.txt

-rw-r--r-- 1 kan63697 1049089 4 Feb 21 19:09 f2.txt

$ git stash list

stash@{0}: WIP on master: 3d969f0 f2

$ git stash pop

On branch 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: f1.txt

modified: f2.txt

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

Dropped refs/stash@{0} (1146960deb3c68d7891e0ccb150b2c1d2a73733a)

## 지역 저장소를 원격 저장소에 연결하기

### 원격 저장소에 연결하기

여러 방법 중 아래와 같은 방법으로 연결해 보겠다.



$ git remote add origin <https://github.com/SangguKang/test-1.git>

이 명령은 원격저장소(remote)에 origin을 추가(add)하겠다고 git에게 알려주는 것이다. 여기서 origin은 깃허브 저장소 주소(<https://github.com/SangguKang/test-1.git>)를 가리킨다. 깃허브 저장소 주소를 그대로 쓰면 너무 길기 때문에 origin이라는 단어로 줄여서 remote에 추가하는 것이다.

Tips> 기본 브랜치를 master라고 하는 것처럼 기본 원격 저장소에는 origin이라는 이름을 사용한다.

잘 연결되었는지 확인  
$ git remote -v

origin https://github.com/SangguKang/test-1.git (fetch)

origin https://github.com/SangguKang/test-1.git (push)

## 원격 저장소에 올리기 및 내려받기

### 원격 저장소에 파일 올리기

$ **git push -u origin master**

Enumerating objects: 3, done.

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

Writing objects: 100% (3/3), 209 bytes | 69.00 KiB/s, done.

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

To https://github.com/SangguKang/test-1.git

\* [new branch] master -> master

Branch 'master' set up to track remote branch 'master' from 'origin'.

여기에서 -u 옵션은 지역 저장소의 브랜치를 원격 저장소의 master 브랜치에 연결하기 위한 것으로 처음에 한번만 사용하면 된다.

## 깃허브에 SSH 원격 접속하기

SSH에서는 기본적으로 private key와 public key를 한쌍으로 묶어서 컴퓨터를 인증한다. public key는 외부로 공개되는 키이고, private key는 사용자 컴퓨터에 저장되는 키이다. 사용자 컴퓨터에서 SSH 키 생성기를 실행하면 private key와 public key가 만들어진다.

SSH 원격 접속은 private key와 public key를 사용해 현재 사용하고 있는 기기를 깃허브에 인증하는 방식이다.

### SSH 키 생성하기

$ cd ~

$ ls

AppData/

'Application Data'@

Contacts/

Cookies@

Desktop/

Documents/

Downloads/

Favorites/

IntelGraphicsProfiles/

Links/

'Local Settings'@

Music/

'My Documents'@

NetHood@

NTUSER.DAT

ntuser.dat.LOG1

ntuser.dat.LOG2

NTUSER.DAT{016888bd-6c6f-11de-8d1d-001e0bcde3ec}.TM.blf

NTUSER.DAT{016888bd-6c6f-11de-8d1d-001e0bcde3ec}.TMContainer00000000000000000001.regtrans-ms

NTUSER.DAT{016888bd-6c6f-11de-8d1d-001e0bcde3ec}.TMContainer00000000000000000002.regtrans-ms

ntuser.ini

ntuser.pol

ODBA/

Pictures/

PrintHood@

Recent@

'Saved Games'/

Searches/

SendTo@

'Start Menu'@

Templates@

Videos/

'VirtualBox VMs'/

$ ssh-keygen

Generating public/private rsa key pair.

Enter file in which to save the key (/c/Users/kan63697/.ssh/id\_rsa):

/c/Users/kan63697/.ssh/id\_rsa already exists.

Overwrite (y/n)? y

Enter passphrase (empty for no passphrase):

Enter same passphrase again:

Your identification has been saved in /c/Users/kan63697/.ssh/id\_rsa.

Your public key has been saved in /c/Users/kan63697/.ssh/id\_rsa.pub.

The key fingerprint is:

SHA256:9vOro2CyqXJVKnRkfVlby2rca+iC5tWFs0yaHTFHc/Q kan63697@19WCP72

The key's randomart image is:

+---[RSA 3072]----+

| . o. +.o |

| o . o = + . |

| o . + + E|

| . . . . B |

| . . o S O o |

| . o . X \* . |

| o. o.+ O o |

|. . =oo...+ |

| o..oo. .oooo. |

+----[SHA256]-----+

$ cd ~/.ssh/

$ ls -al

total 21

drwxr-xr-x 1 kan63697 1049089 0 Aug 6 2019 ./

drwxr-xr-x 1 kan63697 1049089 0 Feb 21 20:21 ../

-rw-r--r-- 1 kan63697 1049089 2602 Feb 21 21:14 id\_rsa

-rw-r--r-- 1 kan63697 1049089 570 Feb 21 21:14 id\_rsa.pub

-rw-r--r-- 1 kan63697 1049089 621 Oct 28 10:04 known\_hosts

### 깃허브에 퍼블릭 키 전송하기

SSH 방식으로 접근하려면 먼저 사용자 컴퓨터에 만들어져 있는 퍼블릭 키를 깃허브 서버로 전송한 다음 저장한다.

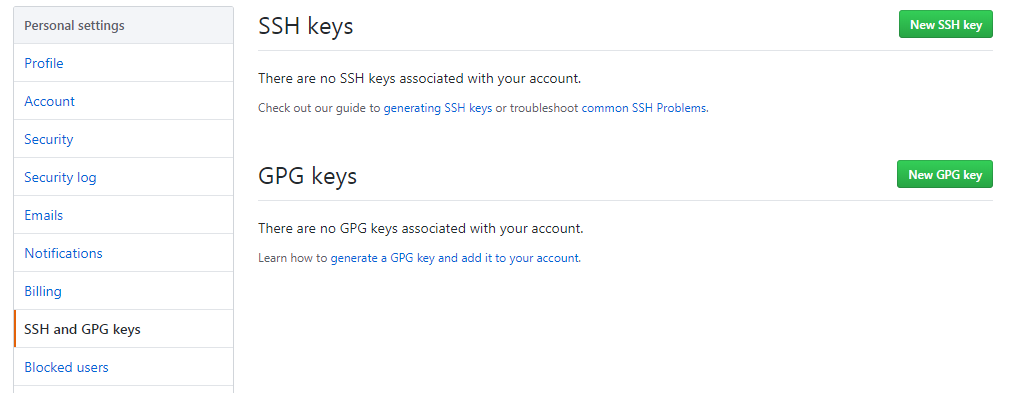
사용자 컴퓨터에서 깃허브 저장소에 접속하면 사용자 컴퓨터에 있는 프라이빗 키와 깃허브 서버에 있는 퍼블릭 키를 비교한다. 퍼블릭 키와 프라이빗 키는 한 쌍이므로 두 개의 키가 서로 맞으면 사용자 컴퓨터와 깃허브 저장소가 연결된다.

$ cd ~/.ssh

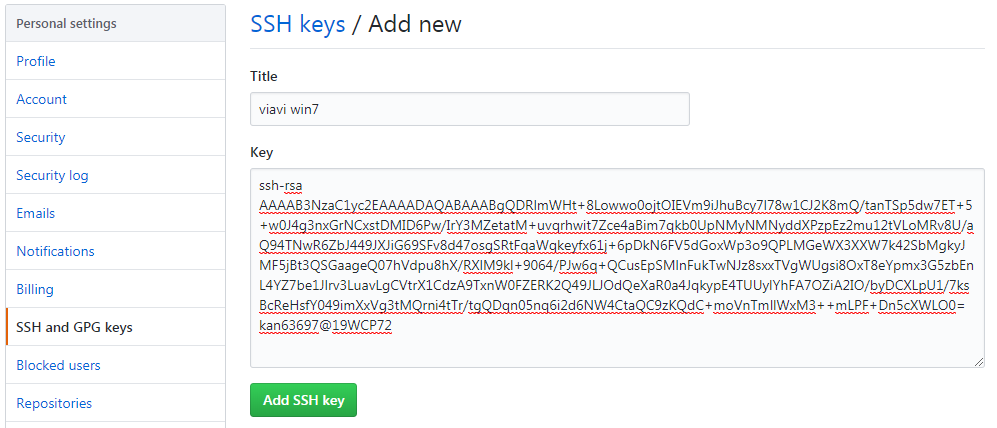
$ cat id\_rsa.pub

ssh-rsa  kan63697@19WCP72

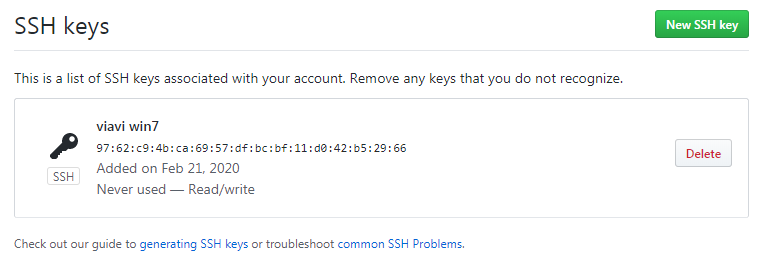
ssh-rsa부터 시작해서 끝까지 copy한다.



New SSH key 버튼을 누른다. SSH 중 퍼블릭 키는 여러 개를 등록할 수 있다.



Add SSH key 버튼을 누른다.



### SSH 주소로 원격 저장소 연결하기

기존에 http로 연결된 리모트 저장소는 사용하지 못한다. 새롭게 깃허브 서버에서 생성한 다음 주소만 SSH로 연결시켜주면 된다. 혹은 기존 리모트 저장소는 그대로 이용하되 로컬 저장소를 새로 만들어서 리모트 연결을 SSH로 다시 해 준다.

$ git remote add origin git@github.com:SangguKang/test-1.git

## 여러 컴퓨터에서 원격 저장소 함께 사용하기

### 원격 저장소 복제하기

git clone <주소> <로컬 디렉토리>

원격 저장소의 주소를 복사한다. 원하는 위치에 clone한다. 이때, git\_home이라는 디렉토리가 없다면 자동으로 만들어진다. 현재 디렉토리에 clone하려면 git\_home 대신마침표.를 입력한다.

$ git clone git@github.com:SangguKang/test-1.git git\_home

Cloning into 'git\_home'...

remote: Enumerating objects: 9, done.

remote: Counting objects: 100% (9/9), done.

remote: Compressing objects: 100% (4/4), done.

remote: Total 9 (delta 0), reused 6 (delta 0), pack-reused 0

Receiving objects: 100% (9/9), done.

원격저장소를 복제하면 자동으로 지역저장소와 원격저장소가 연결된다.

## 원격 브랜치 정보 가져오기

git pull 명령은 원격저장소의 최신커밋을 지역 저장소에 합쳐준다. 하지만 최신 커밋을 합치기 전에 **원격저장소에 어떤 변화가 있는지 먼저 살펴봐야 한다**. 이럴 때는 **원격 브랜치에서 정보만 먼저 가져올 수 있다**.

### 원격 master 브랜치

터미널에서 log를 확인해 보자.

$ git log --oneline

ec00b4d (HEAD -> master, origin/master, origin/HEAD) add d

4848100 add c

0a7ac96 Create f2.txt

64bd55f add b

9fb93da add a

HEAD->master는 이 커밋이 지역 저장소의 최종 커밋이라는 뜻이고, origin/master는 원격 저장소의 최종 커밋이라는 뜻이다.

파일을 새롭게 만들고 커밋해 보자.

$ vim f3.txt

$ git add f3.txt

warning: LF will be replaced by CRLF in f3.txt.

The file will have its original line endings in your working directory

$ git commit -m "create f3.txt"

[master 6acb952] create f3.txt

1 file changed, 1 insertion(+)

create mode 100644 f3.txt

$ git log --oneline

6acb952 (HEAD -> master) create f3.txt

ec00b4d (origin/master, origin/HEAD) add d

4848100 add c

0a7ac96 Create f2.txt

64bd55f add b

9fb93da add a

HEAD->master는 방금 커밋한 ‘create f3.txt’를 가리킨다. origin/master, origin/HEAD는 아직 ‘add d’커밋을 가리키고 있다.

git status 명령을 입력해 보자. 현재 master 브랜치가 origin에 있는 원격 master 브랜치의 버전보다 하나 앞서 있는 것을 알 수 있고, git push 명령으로 지역 저장소의 커밋을 원격 저장소로 올리라고 알려주고 있다.

$ git st

On branch master

Your branch is ahead of 'origin/master' by 1 commit.

(use "git push" to publish your local commits)

nothing to commit, working tree clean

push한 후에 master 브랜치와 origin/master 브랜치가 가리키는 커밋이 같아졌다.

$ git push

Enumerating objects: 4, done.

Counting objects: 100% (4/4), done.

Delta compression using up to 4 threads

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

Writing objects: 100% (3/3), 297 bytes | 42.00 KiB/s, done.

Total 3 (delta 0), reused 1 (delta 0)

To github.com:SangguKang/test-1.git

ec00b4d..6acb952 master -> master

$ git log --oneline

6acb952 (HEAD -> master, origin/master, origin/HEAD) create f3.txt

ec00b4d add d

4848100 add c

0a7ac96 Create f2.txt

64bd55f add b

9fb93da add a

### 원격 브랜치 정보 가져오기 - git fetch

git fetch

fetch는 ‘불러오다, 가져오다’라는 뜻이다. 그래서 git fetch 명령은 원격 저장소의 정보를 가져오는 기능이 있다. pull 명령이 원격 저장소의 커밋을 가져와서 무조건 지역 저장소와 합친다면, fetch 명령은 원격 브랜치에 어떤 변화가 있는지 그 정보만 가져온다. 팀 작업을 할 때 다른 사람이 수정한 소스를 한번 더 훑어 보고 지역 저장소와 합치고 싶다면 풀 대신 페치를 사용해서 커밋을 가져온 다음 지역 저장소와 합치면 된다.

git fetch 명령을 입력하면 원격 저장소에서 무언가 가져올 것이다.

$ git fetch

remote: Enumerating objects: 4, done.

remote: Counting objects: 100% (4/4), done.

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

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

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

From github.com:SangguKang/test-1

ec00b4d..6acb952 master -> origin/master

log 를 보면 HEAD->master만 보이고 원격저장소의 origin/master는 보이지 않는다. 원격저장소의 최신 커밋 정보를 가져왔지만 아직 지역 저장소에 합치지 않아서 원래의 최신 커밋만 나타나기 때문이다.

$ git log --oneline

ec00b4d (HEAD -> master) add d

4848100 add c

0a7ac96 Create f2.txt

64bd55f add b

9fb93da add a

git status 명령으로 확인해 보면 현재 브랜치가 origin/master 브랜치에 비해 1개의 커밋이 뒤쳐져 있다고 나온다. 즉, 원격 저장소의 최신 커밋 하나가 아직 지역 저장소에 반영되지 않았다는 뜻이다. git pull 명령을 사용하면 지역 저장소를 업데이트할 수 있다고 알려주고 있다.

$ git status

On branch master

Your branch is behind 'origin/master' by 1 commit, and can be fast-forwarded.

(use "git pull" to update your local branch)

nothing to commit, working tree clean

페치로 가져온 원격저장소 정보는 origin/master 브랜치가 아닌 FETCH\_HEAD라는 브랜치로 가져온다. 이 브랜치로 가져온 정보는 지역 저장소에 바로 반영되지 않는다.

페치해서 가져온 최신 커밋을 살펴보고 싶다면 FETCH\_HEAD 브랜치로 체크아웃해서 확인한다.

$ git checkout FETCH\_HEAD

Note: checking out 'FETCH\_HEAD'.

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

HEAD is now at 6acb952 create f3.txt

$ git log --oneline

6acb952 (HEAD, origin/master, origin/HEAD) create f3.txt

ec00b4d (master) add d

4848100 add c

0a7ac96 Create f2.txt

64bd55f add b

9fb93da add a

$ git st

HEAD detached at FETCH\_HEAD

nothing to commit, working tree clean

$ ls -al

total 7

drwxr-xr-x 1 MSI 01 197121 0 2월 23 17:04 ./

drwxr-xr-x 1 MSI 01 197121 0 2월 23 16:14 ../

drwxr-xr-x 1 MSI 01 197121 0 2월 23 17:05 .git/

-rw-r--r-- 1 MSI 01 197121 12 2월 23 16:19 f1.txt

-rw-r--r-- 1 MSI 01 197121 7 2월 23 16:14 f2.txt

-rw-r--r-- 1 MSI 01 197121 3 2월 23 17:04 f3.txt

이 내용을 살펴보고 원격 브랜치의 최신 커밋을 지역 저장소에 합칠지 말지를 결정하면 된다.

페치한 후에 최신 커밋을 현재 브랜치에 합치려면 git pull 명령을 사용해서 원격저장소의 소스를 내려받을 수도 있고, git merge 명령으로 FETCH\_HEAD에 있던 커밋을 병합할 수도 있다.

$ git checkout master

Previous HEAD position was 6acb952 create f3.txt

Switched to branch 'master'

Your branch is behind 'origin/master' by 1 commit, and can be fast-forwarded.

(use "git pull" to update your local branch)

$ git merge FETCH\_HEAD

Updating ec00b4d..6acb952

Fast-forward

f3.txt | 1 +

1 file changed, 1 insertion(+)

create mode 100644 f3.txt

$ git log --oneline

6acb952 (HEAD -> master, origin/master, origin/HEAD) create f3.txt

ec00b4d add d

4848100 add c

0a7ac96 Create f2.txt

64bd55f add b

9fb93da add a

살펴본 바와 같이 git pull 명령은 git fetch 명령과 git merge FETCH\_HEAD 명령 두 개를 합친 것과 같은 기능을 한다.

Tips> 페치로 가져온 브랜치 한 번에 병합하기

페치한 뒤 병합할 때 원격 master 브랜치에 있는 커밋이라면,  
$ git merge origin/master

다른 브렌치에 있는 커밋이라면,  
$ git merge origin/브랜치 이름

하지만, 매번 브랜치 이름을 써야 한다면 번거롭다. 다음과 같이 명령하면 페치한 뒤 지역 저장소에 반영하지 않은 최신 커밋을 병합할 수 있다.  
$ git merge FETCH\_HEAD

## 협업에서 브랜치 사용하기

### 새로운 브랜치 푸시하기

$ git checkout -b f

Switched to a new branch 'f'

$ vim f1.txt

$ git add --update

$ git commit -m "features1"

[f 5d9189b] features1

1 file changed, 1 insertion(+)

$ git push origin f

Enumerating objects: 5, done.

Counting objects: 100% (5/5), done.

Delta compression using up to 4 threads

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

Writing objects: 100% (3/3), 304 bytes | 33.00 KiB/s, done.

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

remote:

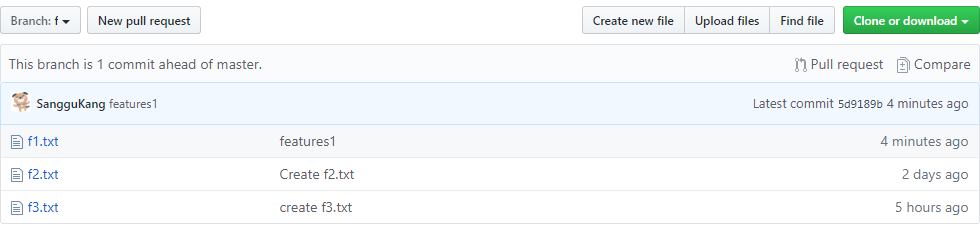
remote: Create a pull request for 'f' on GitHub by visiting:

remote: https://github.com/SangguKang/test-1/pull/new/f

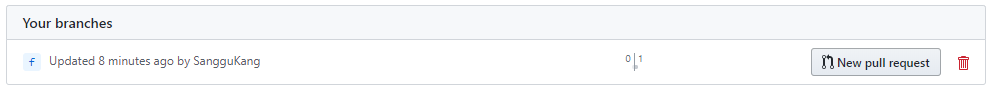
remote:

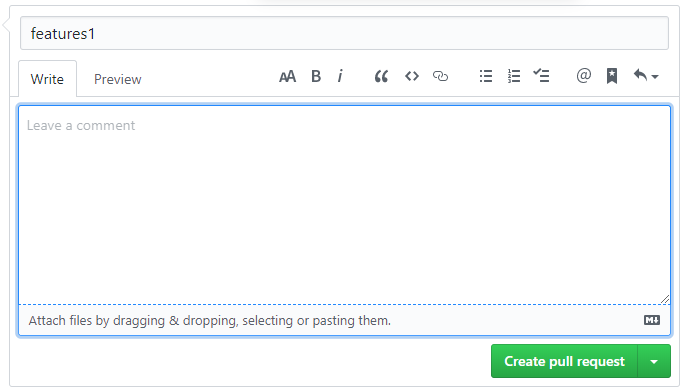
To github.com:SangguKang/test-1.git

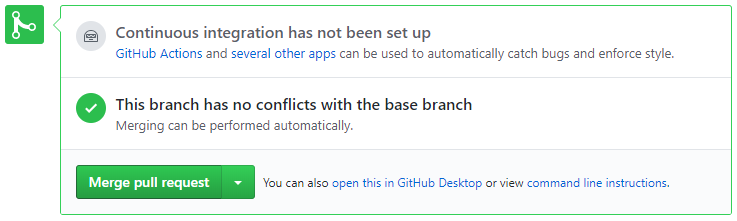
\* [new branch] f -> f

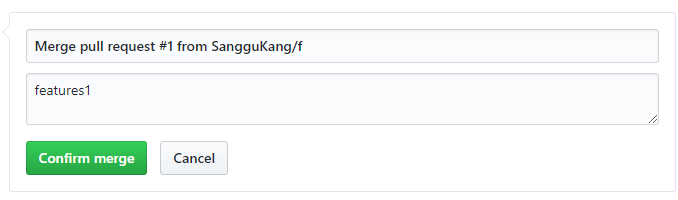


### 풀 리퀘스트로 푸시한 브랜치 병합하기





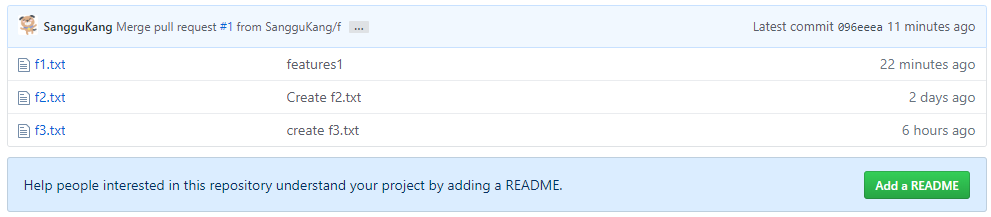


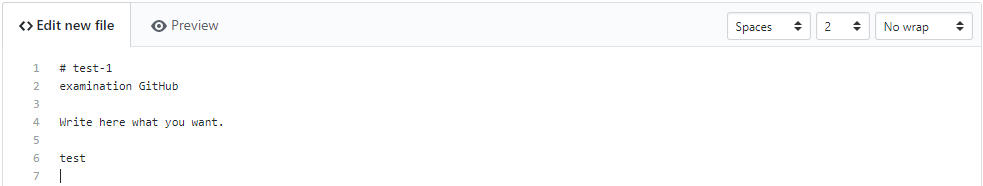


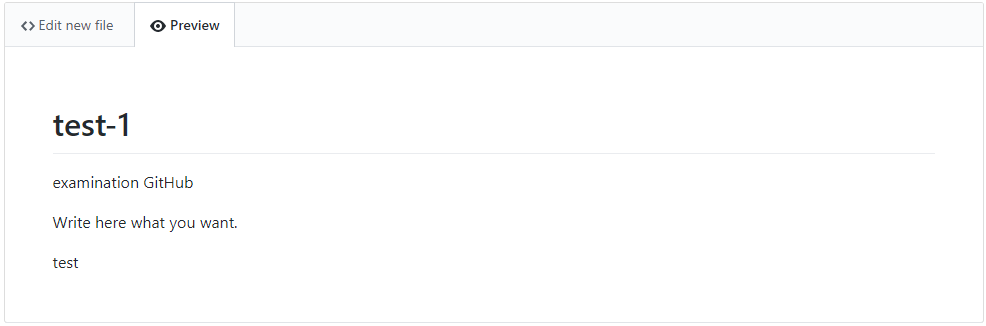
## 깃허브에서 개발자와 소통하기

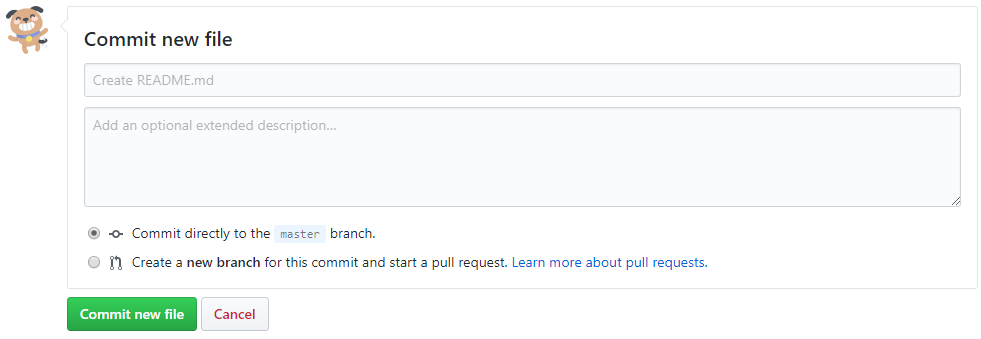
### README 파일 작성하기

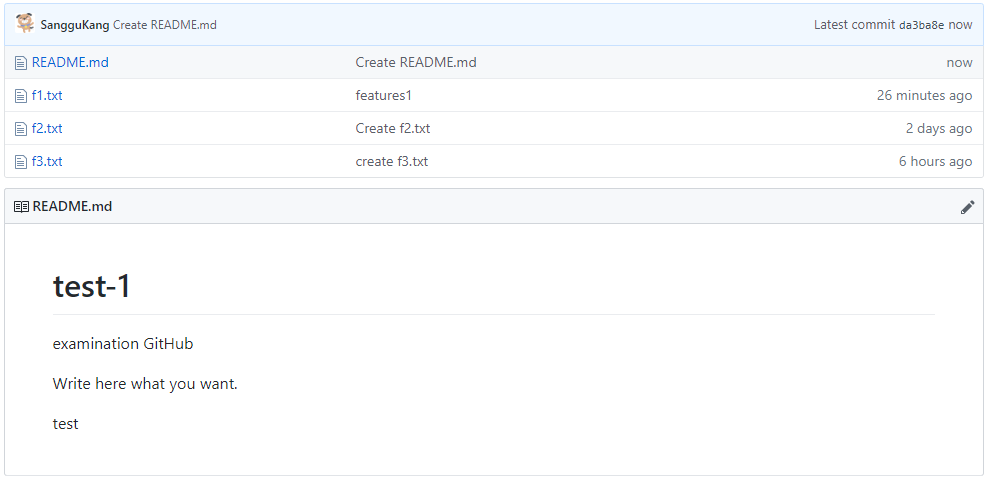
README 파일은 마크다운(markdown) 문법을 사용하기 때문에 확장자는 .md를 사용한다.



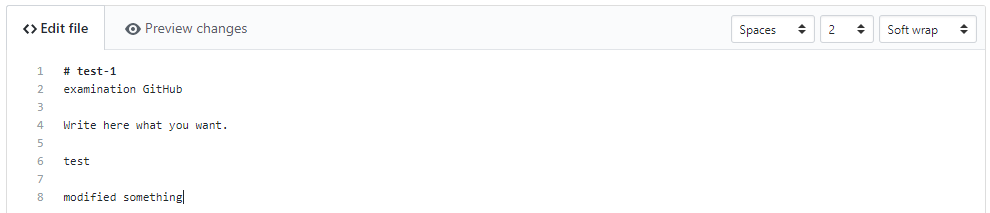


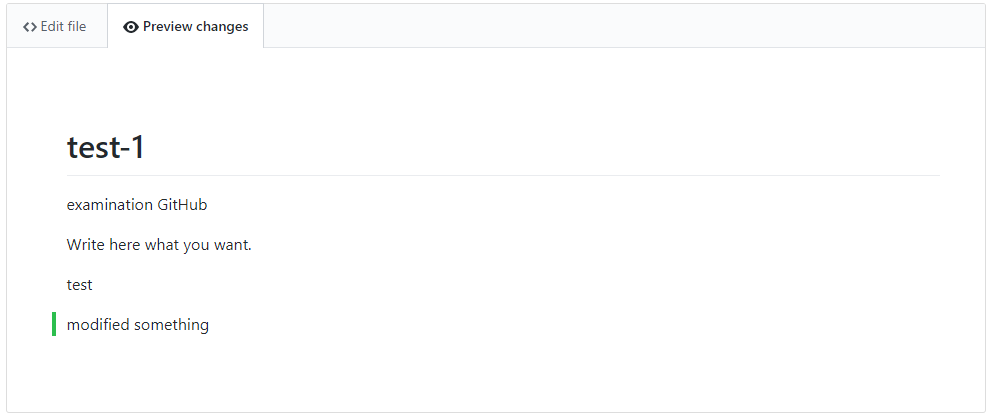




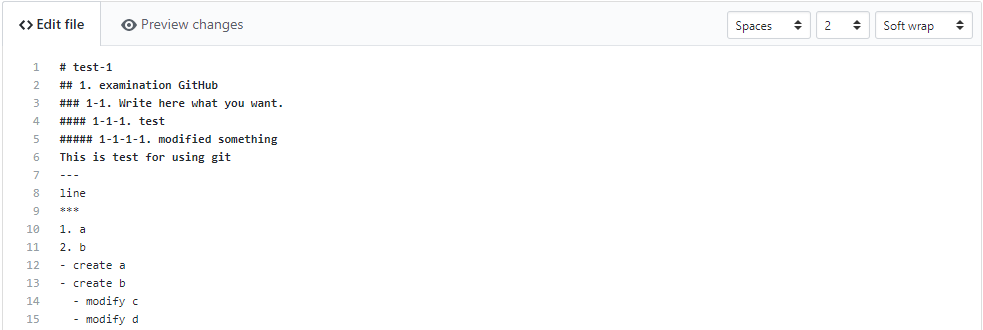


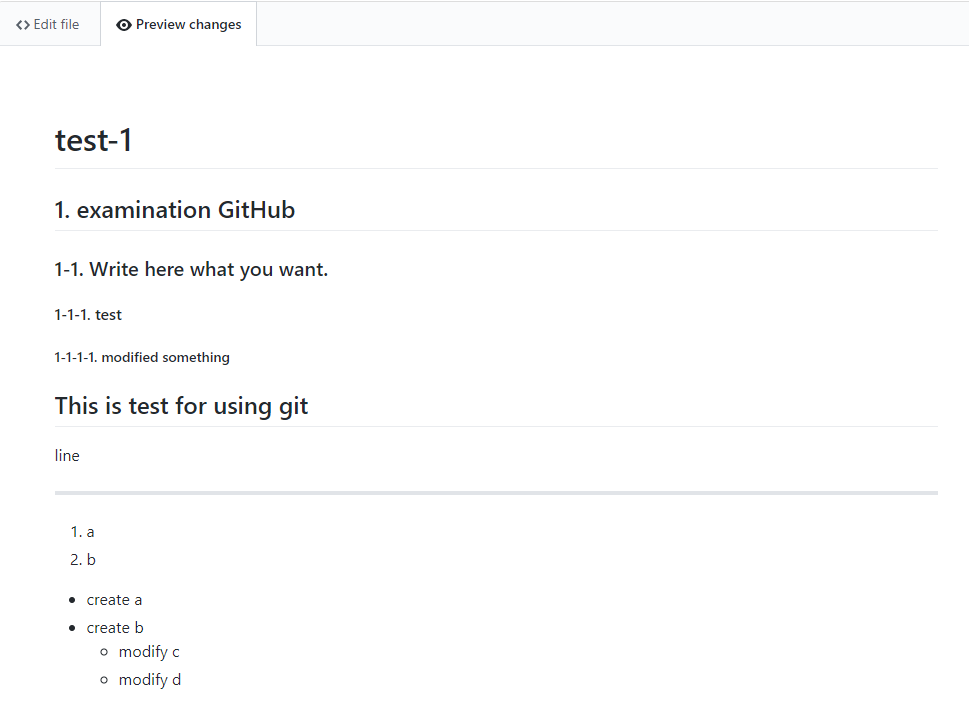
README 파일을 수정하려면 오른쪽에 있는 연필 아이콘을 누른다.





마크다운 문법을 적용하면 좀 더 좋은 문서를 만들 수 있다.





# 고급 사용법

## Submodule

Git 저장소 안에 다른 Git 저장소를 디렉토리로 분리해 넣는 것이 서브모듈이다. 다른 독립된 Git 저장소를 clone해서 내 Git 저장소 안에 포함할 수 있으며 각 저장소의 커밋은 독립적으로 관리한다.

## rebase

### 어디에 사용할까?

* 커밋을 다른 브랜치로 옮기는 일을 할 수 있다.
* 이미 해버린 커밋을 수정하는데에도 유용하게 사용할 수 있다.
* 브랜치를 깔끔하게 유지하고, 브랜치를 이어 붙이는 등 커밋에 관련된 대부분의 것을 할 수 있다.

협업을 할 때 반드시 git rebase를 알아야 하고, 잘 쓸 수 있어야 비로소 git의 기능을 잘 활용하는 것이라고 할 수 있다.

# Todo Review

## fetch, pull, cherry-pick

## merge, rebase(option)

# 사용법

## Command

git < > [ ] …

#### add

[file name]: 특정한 파일만 준비영역에 추가하기  
[directory name/\*]: 특정한 경로의 모든 파일을 준비영역에 추가하기  
[--all]: 작업폴더에 있는 모든 파일을 준비영역에 추가하기. 현재 git에 기록되지 않은 모든 파일도 준비 영역에 추가된다. 이는 매우 탐욕스런 명령이다. 주의가 필요하다.

[.]: git add .  
작업 트리에서 수정한 파일을 한꺼번에 올릴수 있다. (=git add --all)

파일을 새로 추적할 수 있다(untracted -> tracted). git add 명령은 파일을 새로 추적할 때(새로 만든 파일을 add하여 tracted상태로 만들고자 할 때)도 사용하고 수정한 파일(이미 tracted된 파일이지만 modified된 상태의 파일)을 Staged 상태로 만들 때도 사용한다. Merge할 때 충돌난 상태의 파일을 Resolve 상태로 만들때도 사용한다.

git add README.md process-diagram.png -> 2개의 파일을 추가  
git add branch-naming-rules.png -> 1개의 파일을 추가  
git add –all -> 모든 파일을 추가  
git add ca5g/\* -> ca5g 디렉토리 안의 모든 파일을 추가  
git add \*.svg -> svg 확장자를 가진 모든 파일을 추가

[--update]: git에 기록된 적이 있는 모든 파일 중 마지막 커밋 이후 수정된 파일을 준비 영역에 추가  
git add --update

[--patch]: 한 파일의 일부만을 준비 영역으로 보내 커밋 준비를 한다.  
git add --patch <file name>

[-i]: 대화형 모드

#### branch

브랜치를 생성하거나 삭제하고, 브랜치의 목록을 보여준다.

[-v]: 브랜치 목록과 함께 마지막 커밋 메시지도 함께 보여준다.  
git banch -v

[--merged]: merged된 브랜치 목록을 보여준다.  
git branch –merged

[--no-merged]: merge하지 않은 브랜치 목록을 보여준다.  
git branch –no-merged

[-d]: 브랜치 삭제.  
git branch -d testing

[-D]: merge하지 않은 브랜치를 강제로 삭제한다.  
git branch -D testing

[-vv]: 추적 브랜치가 현재 어떻게 설정되어 있는지 확인하려면

[--all]: 저장소의 모든 브랜치 목록을 볼 때 사용. 로컬 브랜치와 원격 브랜치 목록을 같이 볼 수 있다.  
목록에서 remotes/origin으로 시작하는 목록을 볼 수 있는데, remotes는 ‘local이 아니다’라는 뜻이고, origin은 기본 규약에서 ‘나의 사본은 여기에서 복제된 것’의 의미이다.

[--list]: 모든 로컬 브랜치의 목록을 얻는다.

[--remotes]: 모든 원격 브랜치의 목록을 얻는다.  
실제 원격 브랜치 이름에는 remotes가 붙지 않는다. 그저 해당 브랜치가 어떤 종류인가를 설명해 주는 정보일 뿐이다. 그래서 사용할 수 있는 원격 브랜치의 이름 목록을 얻으려면 –remotes를 사용하면 된다.

Remote branch생성은 [리모트 저장소](#_리모트_저장소) 참조.

git branch <branch name>  
브랜치를 새로 만든다.

#### checkout

다른 브랜치로 이동할 때 사용한다.

[-b]: 브랜치를 만들면서 checkout까지 한번에 한다.  
git checkout -b <new branch name>  
git checkout -b iss53

git checkout -b <new branch name> <branch parent>  
git checkout -b iss53 master

리모트 트래킹 브랜치에서 시작하는 새 브랜치를 만들려면  
git checkout -b serverfix origin/serverfix  
git checkout -b tmAnalysis origin/feature/HA-3653-tm-analyzer

[-d]: 브랜치 삭제.  
git branch -d hotfix

[--track]: 원격 브랜치의 로컬 사본을 생성한다.  
git checkout --track <remote\_name/branch>  
git checkout –track origin/video-lessons

[--filename]: 작업 폴더 안의 파일 수정사항을 폐기한다. 수정사항은 준비 영역에 추가되거나 커밋되지 않은 상태. checkout으로 되돌린 내용은 다시 복구할 수 없다.  
git checkout -- signalTm.cpp

#### cherry-pick

git cherry-pick -x <commit>

특정 브랜치에 잘못 추가한 커밋을 올바른 브랜치로 옮기려고 할 때, 다른 브랜치의 커밋을 현재 브랜치에도 추가하고 싶을 때 사용한다.

#### clean

워킹 디렉토리 청소하기. 작업하고 있던 파일을 stash하지 않고 단순히 그 파일들을 치워버리고 싶을 때.

[-f]: 강제로 그냥 지운다.

[-n]: 가상으로 실행해 보고 어떤 파일들이 지워질지 알려달라는 뜻.  
git clean -d -n

#### clone

[url]



#### commit

[--amend]: 한줄 커밋 후에 추가적인 설명이 필요할 때, 커밋을 개정할 수 있다. commit –m ‘message’으로 커밋한 후에 commit --amend를 입력하면 에디터가 뜨는데 이곳에 내용을 입력하면 된다.  
git add --all  
git commit -m “CH05: Adding technical edits.”  
git commit --amend  
[VIM Editor에 커밋 내용 입력]

[-m]: 메시지와 함께 커밋.

#### config

[--global] 사용자 이름과 이메일을 설정한다.  
git config --global user.name “SangguKang”  
git config --global user.email [kangsanggu74@gmail.com](mailto:kangsanggu74@gmail.com)

git config --global alias.st status  
git config --global alias.br branch

user name과 email을 pull/push할 때마다 입력하지 않도록 변경   
git config credential.helper store --global  
git pull or git push  
name과 email입력(한번은 입력해야 한다) -> 이후부터는 입력이 필요 없다

[--list] 설정한 모든 것을 보여준다.



git config --global core.editor “vim”  
git의 편집기 설정

#### diff

단순히 파일이 변경됐다는 사실이 아니라 어떤 내용이 변경됐는지 살펴보려면 git status 대신 git diff 명령을 사용해야 한다. Patch처럼 어떤 라인을 추가했고 삭제했는지가 궁금할 때 사용한다.

이 명령은 working directory에 있는 것과 Staging Area에 있는 것을 비교한다. 그래서 수정하고 아직 Stage하지 않은 것을 보여준다.  
**git diff**

만약 commit하려고 Staging Area에 넣은 파일의 변경부분을 보고 싶으면 **git diff –staged** 옵션을 사용한다. 이 명령은 저장소에 commit한 것과 Staging Area에 있는 것을 비교한다.

[--check]: 공백문자를 검사해서 공백문자에 대한 오류를 확인할 수 있다.  
git diff --check



Commit 하기 전 마지막으로 실수한 것이 없는지 확인할 수 있는 기회를 제공한다.

#### fetch

리모트 저장소의 데이터를 모두 로컬로 가져오지만, 자동으로 Merge하지는 않는다(git pull 명령 사용은 merge시킬 수 있다).  
원격 브랜치 목록은 자동으로 최신상태로 유지되지 않는다. 목록을 업데이트하려면 fetch 명령어를 사용한다.  
git fetch origin

#### grep

[-n]: 찾을 문자열이 위치한 라인 번호도 같이 출력한다.  
git grep -n html

[--count]: 어떤 파일에서 몇 개나 찾았는지 알고 싶을 때 사용한다.  
git grep –count html

#### init

.git이라는 하위 디렉토리를 만든다. .git 디렉토리에는 저장소에 필요한 뼈대 파일(Skeleton)이 들어 있다.

git init

git init <디렉토리 이름>: 새로운 디렉토리를 만들고 저장소를 초기화하는 과정까지 한번에  
$ **git init manual-2**

Initialized empty Git repository in C:/Works/02\_project\_ind/41\_Utils/01\_Git/manual-2/.git/

$ cd manual-2/

$ ls -al

total 4

drwxr-xr-x 1 kan63697 1049089 0 Feb 20 20:28 ./

drwxr-xr-x 1 kan63697 1049089 0 Feb 20 20:28 ../

drwxr-xr-x 1 kan63697 1049089 0 Feb 20 20:28 .git/

#### log

저장소의 commit 히스토리를 시간 순으로 보여줌.

[-p]: 각 commit의 diff 결과를 보여준다.

[-2]: 최근 두 개의 결과만 보여준다.

[--stat]: 각 commit의 어떤 파일이 수정됐는지, 얼마나 많은 파일이 변경됐는지, 또 얼마나 많은 라인을 추가하거나 삭제했는지의 통계 정보를 보여준다. 커밋에 관련된 파일까지 함께 살펴볼 때 사용한다.  
git log --stat

[--oneline]: log 메시지를 한 줄로 보기

[--pretty=oneline]: 각 commit을 한 라인으로 보여준다.

[--decorate]: 브랜치가 어떤 커밋을 가리키는지 알 수 있다. git log –oneline --decorate

기타 다른 많은 옵션들은 pro git 등을 참조한다.



[--graph]: graphical하게 보여준다.  
git log --oneline --graph

#### merge

새로운 브랜치(origin/serverfix)의 내용을 merge하려면, git merge origin/serverfix

From\_branch를 to\_branch와 merge한다.  
git merge <from\_branch> <to\_branch>

#### mv

#### push

git push <remote repo name> <branch name>  
git push origin master

원격 저장소에 트랙킹된 저장소에서 git push 명령만 사용하면 알아서 원격 저장소에 push 된다.

git push origin :<branch-name>  
**git push origin :feature01**  
주의) 원격 저장소의 브랜치를 삭제.

git push origin <branch-name>  
**git push origin feature01**  
새로운 브랜치 저장소를 원격에 생성.

git push --delete <branch-name>  
git push --delete restoring\_old\_commit  
원격 저장소 삭제

#### pull

서버로부터 데이터를 가져와서 현재 로컬 브랜치와 서버의 추적 브랜치를 merge한다.

git pull <remote> <branch>  
git pull origin feature/HA-3653-tm-analyzer

#### rebase

remote 브랜치에 커밋을 깔끔하게 적용하고 싶을 때 사용한다. 이미 공개 저장소에 push한 커밋을 rebase하지 마라.

git rebase [branch]

git rebase [basebranch] [topicbranch]

example>

git rebase master  
CONFLICT solve  
git add myfile.txt  
git rebase --continue (commit 대신)  
git checkout master  
git merge issue3

#### reflog

저장소 로컬 사본에서 일어난 모든 일의 히스토리를 목록으로 볼 수 있다. 클론 이후부터 시작해서 코드에 영향을 주지 않는 브랜치 체크아웃 같은 행위도 포함된다.  
git reflog

#### remote

이미 설정된 원격 연결이 있는지 알아본다.  
$ git remote --verbose  
origin git@gitlab.com:SangguKang/book\_unix\_linux\_utility.git (fetch)  
origin git@gitlab.com:SangguKang/book\_unix\_linux\_utility.git (push)  
-> origin: 원격 연결의 별명  
-> 첫 번째 줄은 새 작업을 받아오는 장소(fetch)를 나타내고, 두 번째 줄은 새 작업을 업로드할 장소(push)를 나타낸다.

[-v] 단축 이름과 URL을 함께 볼 수 있다.

[show <리모트 저장소 이름>] 예)git remote show origin

[rename]

[remove]

[add]: 현재 작업중인 프로젝트에 팀의 저장소를 추가한다.

[rm]: 원격 저장소 삭제  
git remote rm <remote-name>

원격 저장소 연결  
$ git remote add origin https://github.com/SangguKang/test-1.git

#### reset

돌아가려는 커밋으로 저장소는 재설정되고 해당 커밋 이후의 이력은 사라진다. 이미 push한 상태라면 reset을 사용하지 말고 revert만 사용해야 한다.

git reset <옵션> <돌아가고 싶은 커밋>  
옵션: --hard, --soft, --mixed  
git reset –hard a3bbb3c

* staged area(준비영역)에서 파일을 not staged 영역으로 되돌릴 때. 다시 말해 실수로 staged area에 너무 많은 파일을 add했는데 일부 파일을 unstaged area로 내려서 의미있는 일부만 commit할 수 있다. reset으로 unstaged area로 내려간 파일의 수정사항을 되돌리는 것은 아니다.  
  git reset HEAD <file name>: git reset HEAD signal.cpp

#### revert

이전 작업을 삭제. 커밋 히스토리는 그대로 두며, revert한 사실이 명시된 새로운 커밋이 생긴다. 브랜치는 게재됐고 작업 폴더는 정리된 상태이다.

git revert <돌아가고 싶은 커밋>: git revert a3bbb3c

이전에 작성한 커밋을 안전하게 지우고 싶을 때 사용한다.

#### rm

staged된 파일을 unstaged하게 할 때 사용한다.

왠만하면 사용하지 말 것!

실수로 지워진 파일을 복구하려면,  
git reset –hard HEAD


#### show

git show [commit]: 하나의 커밋의 상세 정보를 확인한다. 태그 정보와 커밋 정보를 모두 확인할 수 있다.  
git show <SHA-1>  
git show 1c002d  
git show <branch-name>  
git show master

git show tag: 해당 태그가 적용된 커밋의 상세 정보를 본다.  
git show import

#### show-branch

git show-branch

#### stash

아직 완료하지 않은 일을 커밋하지 않고 나중에 다시 돌아와 작업을 하고 싶을 때 사용한다. Stash는 modified이면서 tracked 상태인 파일과 Staging Area에 있는 파일들을 보관해 두는 장소이다.

git stash

git stash list

git stash pop

git stash drop

#### status

파일의 상태를 확인한다.

[-s] or [--short] 현재 변경한 상태를 짤막하게 보여준다.

#### submodule

두 프로젝트를 서로 별개로 다루면서도 그 중 하나를 다른 하나 안에서 사용하려고 할 때 사용. 각 저장소의 커밋은 독립적으로 관리한다.

[add URL]: git 저장소를 서브모듈로 추가  
git submodule add <http://github.com/chaconinc/DbConnector>

[init]:

[update]:

git submodule status

#### tag

git tag <tag name> <커밋해쉬>  
git tag import ee6426b  
  
위와 같이 새로 추가한 tag는 git tag를 이용해 모든 태그 목록을 확인할 수 있고, 태그된 커밋을 선택해서 git show <tag name>을 이용해 해당 커밋을 검토할 수 있다.

[-a] annotated tag 만들기.

git tag: 모든 태그 목록을 본다.

example>

git tag apple  
git log --decorate

## History 관리(롤백, 리버트, 리셋, 리베이스)

### 정확한 작업 취소 방법 선택



|  |  |  |
| --- | --- | --- |
| **원하는 것…** | **설명** | **해결책** |
| 작업 폴더 안의 파일의 수정사항을 폐기한다 | 수정사항은 준비영역에 추가되거나 커밋되지 않았다 | git checkout -- filename |
| 작업 폴더의 모든 저장되지 않을 수정사항을 폐기한다. | 파일은 준비 영역에 추가됐지만, 커밋되지는 않았다 | git reset --hard |
| 특정 커밋을 제외한 여러 커밋을 합친다 |  | git reset commit |
| 기록되지 않은 파일을 포함해 모든 저장되지 않은 파일을 삭제한다 | 수정된 파일은 커밋되지 않았다 | git clean -fd |
| 준비 영역의 모든 수정사항과 특정 커밋 전까지 커밋된 작업을 삭제하되, 작업 폴더의 새 파일은 삭제하지 않는다 |  | git reset --hard commit |
| 어떤 작업을 삭제하되, 커밋 히스토리는 그대로 둔다(롤포워드 roll forward) | 브랜치는 게재됐고 작업 폴더는 정리된 상태다 | revert commit |
| 브랜치 히스토리에서 하나의 커밋 삭제하기 | 수정사항은 커밋됐고, 작업 폴더는 정리됐고 브랜치는 게재되지 않았다 | rebase --interactive commit |
| 이전 작업을 계속 진행하되, 다른 커밋과 합친다 | squash 옵션을 선택한다 | rebase --interactive commit |

### 작업을 위한 Git 명령어

|  |  |
| --- | --- |
| **명령어** | **용도** |
| git checkout -b branch | branch라는 이름의 새 브랜치를 생성한다 |
| git add filename(s) | 저장소에 커밋하기 위해 파일을 준비 영역에 추가한다. |
| git commit | 준비 영역에 추가된 수정사항을 저장소에 저장한다 |
| git checkout branch | 현재 작업 브랜치를 특정한 브랜치로 교체한다 |
| git merge branch | 브랜치 branch의 커밋을 현재 브랜치에 통합한다 |
| git branch --delete | 로컬 브랜치를 삭제한다 |
| git branch -D | 커밋이 다른 곳에 통합되지 않은 로컬 브랜치를 삭제한다 |
| git clone URL | 원격 저장소의 로컬 사본을 생성한다 |
| git log | 현재 브랜치의 커밋 히스토리를 읽는다 |
| git reflog | 현재 브랜치의 확장 히스토리를 읽는다 |
| git checkout commit | 특정 커밋을 체크아웃하고 떨어진 HEAD 상태가 된다 |
| git cherry-pick commit | 하나의 브랜치 커밋을 다른 브랜치로 복사한다 |
| git reset --merge ORIG\_HEAD | 최신 병합을 통해 적용된 모든 커밋을 현재 브랜치에서 삭제한다 |
| git checkout -- filename | 수정됐지만 커밋되지 않은 파일을 복구한다 |
| git reset --hard HEAD | 수정된 모든 파일을 이전에 저장된 상태로 복구한다 |
| git reset commit | 특정 커밋 이후에 커밋된 모든 수정사항을 준비영역에서 삭제한다 |
| git rebase --interactive commit | commit 이후 추가된 커밋을 수정하거나 합친다 |
| git rebase - continue | 병합 충돌을 해결하라고 리베이싱 과정을 계속 진행한다 |
| git revert commit | 특정 커밋에 저장된 수정사항을 무력화한다. 이를 실행하면 공유 가능한 역전 커밋의 히스토리가 생성된다 |
| git log --oneline --graph | 현재 브랜치의 히스토리 그래프를 출력한다 |
| git revert --mainline 1 commit | 하나의 병합 커밋을 역전시킨다 |
| git branch --contains commit | 특정 커밋 객체를 포함하는 모든 브랜치 목록을 본다 |
| git revert --no-commit last\_commit\_to\_keep .. newest\_commit\_to\_reject | 여러 커밋을 역전시키고 되돌려지는 커밋 하나당 하나의 객체를 생성하는 대신 하나의 역전 커밋을 생성한다 |
| git filter-branch | 파일을 저장소에서 영원히 삭제한다 |
| git reflog expire | 확장 히스토리를 잊고 오직 저장된 커밋 메시지만 사용한다 |
| git gc --prune=now | 가비지 컬렉터를 실행해 모든 커밋되지 않은 수정사항을 로컬 메모리에서 삭제한다 |

### 적절한 명령어 선택 플로우 차트

#### 파일 수정 사항 제거



### 실험 작업을 위한 브랜치 사용

티켓 브랜치로부터 브랜치를 하나 더 만들어 실험적인 수정사항을 적용하자([예제1-1]). 결과가 만족스럽다면 실험 브랜치를 티켓 브랜치에 병합하면 된다([예제 1-2]). 실험 브랜치는 병합하고 나면 삭제해도 된다([예제 1-3]).

예제1-1 실험 브랜치를 사용해 수정사항 테스트


예제1-2 실험 브랜치를 메인 브렌치에 병합


예제 1-3 실험 브랜치 삭제


### 단계적 리베이싱

일반적으로 리베이스는 하나의 브랜치에서 그것의 부모 브랜치에서 만들어진 커밋을 최신 상태로 업데이트하는 과정이라고 한다.

#### 리베이스 시작하기

|  |
| --- |
| git checkout feature  git rebase master |

#### 리베이스 중에 발생한 삭제된 파일에 대한 충돌

|  |
| --- |
| First, rewinding head to replay your work on top of it…  Applying: CH10: Stub file added with notes copied from video recording lessons.  Usng index info to reconstruct a base tree…  A ch10.asciidoc  Falling back to patching base and 3-way merge…  CONFLICT (modify/delete): ch10.asciidoc deleted in HEAD and modified in CH10:  Stub file added with notes copied from video recording lessons.. Version CH10:  Stub file added with notes copied from video recording lessons. Of ch10.asciidoc  left in tree.  Failed to merge in the changes.  Patch failed at 0001 CH10: Stub file added with notes copied from video recording  lessons.  The copy of the patch that failed is found in:  /Users/emmajane/Git/1234000002182/.git/rebase-apply/patch  When you have resolved this problem, run “git rebase --continue”.  If you prefer to skip this patch, run “git rebase --skip” instead.  To check out the original branch and stop rebasing, run “git rebase --abort”. |

When you have resolved this problem, run “git rebase --continue”.

앞의 문구는 무엇을 해야 할지를 설명해 준다.  
1. 병합 충돌을 해결한다.  
2. 병합 충돌이 해결된 것 같으면 다음과 같은 명령어를 실행한다.  
git rebase --continue

기본 파일 비교 도구를 사용해 충돌이 일어난 파일을 열어서 1단계를 해결한다.  
git mergetool ch10.asciidoc

더 이상 병합 충돌 표시가 없으면 병합 도구를 끝내고 Git이 안내해주는 다음 단계로 넘어간다.  
git rebase --continue

|  |
| --- |
| ch10.asciidoc: needs merge  You must edit all merge confilicts and then mark them as resolved using git add |

Git에 문제가 무엇인지 명령어 status를 사용해 물어본다.  
git status

|  |
| --- |
| rebase in progress; onto 6ef4edb  You are currently rebasing branch ‘ch10’ on ‘6ef4edb’.  (fix conflicts and then run “git rebase --continue”)  (use “git rebase --skip” to skip this patch)  (use “git rebase --abort” to check out the original branch)  Unmerged paths:  (use “git reset HEAD <file>… “ to unstage)  (use “git add/rm <file>…” as appropriate to mark resolution)  deleted by us: ch10.asciidoc  no changes added to commit (use “git add” and/or “git commit -a”) |

수정사항을 적용하지 않으려면  
git reset HEAD ch10.asciidoc

충돌을 해결했다면  
git add book.asciidoc  
git status

|  |
| --- |
| rebase in progress; onto 6ef4edb  You are currently rebasing branch ‘ch10’ on ‘6ef4edb’.  (all conflicts fixed: run “git rebase --continue”)  Changes to be commited:  (use “git reset HEAD <file>… “ to unstage)  modified: book.asciidoc |

git rebase --continue

### 잃어버린 작업 찾기 개요

git에서 커밋된 작업을 완전히 삭제하기란 매우 어렵지만, 작업을 잃어버리는 일은 쉬운 편이다. 어떤 작업을 잃어버렸을 때, 가장 처음으로 할 일은 그 작업이 저장됐던 커밋을 찾는 것이다.

이때 유용한 명령이 git log와 git reflog이다(2.1 Command 참조).

명령어 log와 reflog 모두 저장소에 저장된 특정 상태의 커밋 ID를 보여준다. 이 커밋 ID를 찾기만 하면 상태를 확인하고 특정한 시점의 코드 베이스 상태를 임시로 복구할 수 있다.

현재 상태: br-master, b4faee3



저장소에서 특정 커밋 확인하기: git checkout 74dad73






특정 커밋을 체크아웃하면 해당 브랜치의 연결된 히스토리로부터 떨어지게 된다. 현재 상태를 저장하려면 새로운 브랜치를 체크아웃하고 현재 상태를 새 브랜치에 저장하면 된다. git checkout -b restoring\_old\_commit



수정할 것이 있으면 수정을 한다.



추가나 수정이 끝나면 새 브랜치를 기존 브랜치에 통합한다.



병합이 완료되면 임시 브랜치를 삭제해 로컬 저장소를 정리한다.



### 파일 복구하기

지워서는 안되는 파일이나 수정해서는 안되는 파일을 수정한 경우.

* 커밋 전이고, 준비 영역 전이라면



* 이미 준비영역에 추가했다면  
  -. 삭제 -> git add -> git status  
    
  -. 복구하고 싶은 파일을 준비영역에서 삭제  
    
  -. File이 작업 영역에 있으므로 checkout  
    
  -. 작업 폴더의 수정사항을 모두 취소하고, 이전 커밋에 저장된 파일로 되돌리길 원할 때 한번에 처리할 수 있다.  
    
  

### 커밋 작업하기

커밋은 특정 시점의 모든 파일의 상태를 포함하는 저장소의 스냅샷이라고 할 수 있다. reset을 사용해서 특정 커밋을 완전히 삭제할 수 있고, revert를 사용해서 특정 커밋의 효과를 역전시킬 수도 있고, rebase를 사용해서 커밋의 순서를 바꿀 수도 있다.

이번에는 다른 사람과 아직 공유하지 않은 커밋을 가지고 작업한다고 가정한다(예) 브랜치를 push하지 않은 상태).

#### 커밋 수정하기

커밋에 작은 수정 하나가 빠진 경우, 해당 커밋을 수정해서 수정 사항(추가 파일, 파일 수정, 메시지 업데이트)을 업데이트할 수 있다. 자주 간략한 한 줄 커밋 메시지를 잘 짜인 작업 요약문으로 변경하거나 활요할 수 있다.

**공유된 히스토리(작업을 이미 push 했다면)는 절대 변경하지 말라**.

git add --all  
git commit --amend

메시지만 업데이트할 경우, add 명령어는 생략하고 commit만 할 수 있다.  
git commit --amend

파일을 수정하거나 할 경우에는 위의 첫 예와 같이 add 명령어와 commit 명령어를 함께 사용한다.

이때, 새 수정사항이 기존 커밋에 추가되고 변경된 커밋 객체에는 새로운 ID가 할당된다.

#### Reset으로 커밋 합치기

reset 명령어를 사용해서 리베이스의 커밋 합치기(**squash**) 효과를 낼 수 있다. 앞서 말했지만 공유되지 않은 브랜치의 히스토리를 변경하는 데만 reset을 사용하는 것이 가장 좋다(즉, 로컬 브랜치나 서버에 push하지 않은 브랜치를 말하는 것이다).

구슬이 꿰여져 있는 줄을 생각해 보자. 그 줄에 구슬이 20개 달려있는데, 4번째 구슬을 쥐고 앞의 3개의 구슬을 빼내서 하나의 큰 구슬로 교체한다고 가정해 보자.

명령어 reset을 사용해 구슬의 끝 부분(가장 최신 커밋이 끝에 위치)을 기존의 맨앞 구슬에서 4번째 구슬로 지정할 것이다. 그리고는 실 끝에 있는 3개의 구슬을 떼어낸다. 빼낸 3개의 구슬을 합쳐 하나의 커밋으로 만들고 만들어진 커밋을 다시 줄에 추가하는 것이다.

-. log 명령어를 사용해 커밋 히스토리를 확인한다.  


-. 만일 줄의 끝에 놓고 싶은 구슬이1643448(이것이 끝에서 4번째 구슬)이라면, 남길 구슬을 손가락으로 잡고 나머지는 줄에서 빼낼 것이다.  


-. 이제 3개의 떨어진 구슬이 있다. 이 구슬은 저장소에 기록되지 않은 수정사항으로 나타날 것이다. 파일 콘텐츠는 수정되지 않을 것이다.  
  


-. 명령어 diff를 사용해 새 커밋에 어떤 변경사항이 포함되는지 확인할 수 있다.  


-. 앞의 떨어져나간 3개의 구슬(커밋)에 포함된 모든 수정사항을 하나의 커밋으로(하나의 큰 구슬) 합치려면 명령어 add를 사용해 해당 수정사항을 준비영역으로 추가해야 한다.  
  


-. 이제 관련 파일은 준비영역에 추가되었고 명령어 diff를 사용해도 저장소에 커밋할 내용이 보이지 않을 것이다. 준비 영역의 수정사항을 확인하려면 --staged 옵션을 사용한다.  


-. 커밋 과정을 완료한다. 이제 3개의 커밋은 하나의 커밋으로 합쳐질 것이다.  


#### 인터랙티브 리베이싱으로 커밋 수정하기

리베이싱은 논란이 많은 주제 중 하나이다. 리베이싱은 주로 히스토리가 기록된 방식을 바꾸는데 사용된다. 보통 작업 폴더의 파일 내용은 변하지 않는다. 잘못 사용하면 공유 브랜치에 혼란을 초래할 수 있다.

리베이싱을 사용하기 적절한 상황:

1. 브랜치를 업데이트할 때  
2. 커밋을 좀 더 읽기 쉬운 히스토리로 만들 때(이 절에서는 이 부분에 대해 언급)

인터렉티브 리베이싱에서는 여러 개의 작은(자잘한) 커밋을 하나의 온전한 아이디어로 합치는 기본 원칙을 볼 수 있다.

-. 시작점으로 삼을 커밋을 선택: 최근 커밋 3개(1643448, 376efa3, 1c6ab90)를 하나의 커밋으로 만들겠다고 결정했다. 커밋 74dad73을 시작점으로 선택하자.  


Abort!!!

#### 브랜치 병합 취소하기



## 1인 이상의 팀

## 상황별 용도

### 리모트 저장소

* 리모트 저장소 확인하기  
  git remote -v
* 리모트 저장소 추가하기  
  ???
* 리모트 저장소를 Pull 하거나 Fetch하기  
  git fetch [remote-name]
* 리모트 저장소에 push하기  
  git push origin master
* 리모트 저장소 살펴보기  
  git remote show [remote 저장소 이름] 예)git remote show origin
* 리모트 저장소 이름을 바꾸거나 리모트 저장소를 삭제하기  
  git remote rename pb paul  
  git remote remove paul
* 리모트 저장소에 새로운 브랜치 생성하기  
  git checkout -b tm-analyzer  
  git push origin tm-analyzer

### 태그

태그는 특정한 커밋을 정확히 찾아내기 위해 사용된다. 일종의 책갈피라 생각할 수 있다. 공유된 브랜치에서는 보통 배포 목적으로 사용되고 사용 방식도 전체 팀이 정한 규약에 따라야 한다.

* 새로운 태그 추가하기  
  git tag <tag name> 커밋 해쉬  
  git tag import fa04c30
* 태그 조회하기  
  git tag
* 태그 붙이기  
  ?
* 나중에 태그하기  
  ?
* 태그 공유하기  
  ?
* 태그를 checkout하기  
  ?

### 브랜치

* 새 브랜치 생성하기  
  git branch <branch-name>  
  git branch testing
* 새 브랜치 생성하기 - 기존 브랜치 내용에서  
  git checkout -b <branch-name>  
  git checkout -b feature01
* 리모트 브랜치 생성하기  
  git push origin <branch-name>  
  git push origin feature01
* 리모트 브랜치 삭제하기  
  git push origin :<branch-name>  
  
* 브랜치 이동하기  
  git checkout <branch-name>  
  git checkout testing

### 되돌리기 Reset과 checkout

* 파일 상태를 Unstaged로 변경  
  git reset HEAD <file name>
* Modified 파일 되돌리기(수정된 파일 되돌리기)  
  git checkout -- <file name>  
  매우 위험한 명령. 원래 파일로 덮어쓰기 때문에 수정한 내용은 전부 사라짐. 변경한 내용을 쉽게 버릴 수는 없고 하지만 당장은 되돌려야만 하는 상황이라면 Stash와 Branch를 사용한다.

### 원격 브랜치 삭제

더 이상 필요 없는 원격 브랜치 삭제하기

git push origin :<branch-name>  
git push origin :feature01

## 브랜치 워크플로우

### Long-Running 브랜치

개발자가 많이 선호하는 워크플로우. 배포했거나 배포할 코드만 master 브랜치에 merge해서 안정버전의 코드만 master 브랜치에 둔다. 개발을 진행하고 안정화하는 브랜치는 develop이나 next라는 이름으로 추가로 만들어 사용한다. 테스트를 거쳐서 안정적이라고 판단되면 master 브랜치에 merge한다.



### 토픽 브랜치

어떤 한 가지 주제나 작업을 위해 만든 짧은 호흡의 브랜치이다. Git에서는 브랜치를 하나 만드는데 드는 비용이 크지 않기 때문에 매우 일상적으로 브랜치를 만들고 merge하고 삭제한다.

# Definition and Explanation

## HEAD

Git의 축약어. 현재 브랜치의 가장 최신 커밋을 가리킨다.

# Tools

## Beyond Compare

3way merge를 지원한다.

좌측에는 BASE, 중간에는 LOCAL, 오른쪽에는 REMOTE가 있다.

**LOCAL**은 **현재 내가 작업중인 저장소**로 내가 수정한 부분이 있고, **REMOTE**는 merge를 하려고 하는 다른 사람이 수정한 부분이 있다. **BASE**는 LOCAL과 REMOTE의 조상이다. 즉, BASE로부터 파생된 것이라는 뜻이다. 그래서 merge를 할 때 BASE로 돌아갈 수도 있고, LOCAL을 참조할 수도, REMOTE를 참조할 수도 있다.

아래 창의 화살표를 이용해 원하는 부분을 변경하거나, 직접 입력하여 수정한다. 화살표 이용을 권장한다. 그런 다음 느낌표를 눌러 merge했음을 알린다.

# Useful Tip

## Overlay Icon 보이게 하기

레지스트리의 우선순위를 변경해주면 된다. 15개 까지만 표시해 준다고 한다.

-. regedit.exe 실행



-. 아래 위치로 이동 HKEY\_LOCAL\_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\ShellIconOverlayIdentifiers



-. Tortoise 항목들의 이름을 바꾸어 상위로 이동시킨다.(앞에 공백을 추가하면 됨)

-. 재부팅 하면 Overlay Icon이 보인다.

-. 만약 보이지 않는다면, 백업 하기 위해 내보내기를 한 후, reg 파일이 생성되면 더블 클릭하고 재부팅한다.



## Error 발생(clone or push)

인증 강화를 위해 git hub저장소를 이용할 경우, 기존의 password를 입력하라고 나올 경우가 있는데 이때 Token을 입력해야 한다.

토큰 생성은 Git Hub에 로그인한 후 Settings에서 한다. Settings > Developer settings > Personal access tokens 에서 Generate new token 버튼을 클릭하여 생성한다.

<https://curryyou.tistory.com/344>

SSH를 사용하면 매번 token을 입력해야 하는 불편함을 없앨 수 있다고 함.

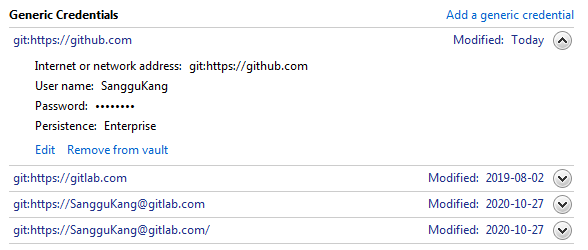
### Windows 자격 증명

매번 토큰을 입력해야 하는 불편함을 해소하고자 설정하였으나 안됨!

제어판>사용자 계정>자격증명 관리자>Windows자격 증명

Control Panel>All Control Panel Items>Credential Manager

Generic Credentials(일반 자격 증명)에서 Add a generic credential을 클릭하여 주소, 사용자이름, PW를 입력한다. PW에는 생성한 Token을 입력한다.



### Tortoise Git 자격증명 저장

<https://medium.com/@zoegroes/easy-quick-setting-up-tortoisegit-credential-bccc8baf802f>

* Open Tortoisegit → settings → Git → Credential
* 1. Change Config types to “Global”, 2. enter your repository URL, 3. change Helper to “Wincred”, 4. Do Enter your Github user name