Git & Github 실무활용

(보충교재)

목차

1. INTRODUCE

GIT이란?

- Git은 소프트웨어 개발에서 버전 관리를 위한 도구이다.
- Git은 분산 버전 관리 시스템이다.

개별 작업자가 중앙화된 코드 호스팅 시스템에서 독립적으로 일하고 프로젝트 저장소의 개별 로컬 사본에서 작업을 수행한다.

GIT 특징

- Git의 데이터는 파일 시스템의 스냅샷이라 할 수 있다.
- Git은 시간순으로 파일의 스냅샷을 저장한다.
- Git은 대부분의 명령을 로컬에서 실행한다.
- Git은 오프 라인 상태에서도 작업을 할 수 있다.
- Git은 데이터의 무결성을 위해 SHA-1 해시를 사용하여 체크섬을 만든다.
- 대규모 프로젝트를 위한 완전한 분산 환경 지원
- 체크섬은 해시(SHA-A)로 40 자 길이의 16 진수 문자열

저장소(repository)

이력을 관리하는 저장소

로컬 저장소(Local Repository)

개인 전용 저장소

원격 저장소(Remote Repository)

공유 저장소

변경 이력 관리(history)

커밋 단위, 40자리 해시키로 관리

날짜, 시간, 커밋 메시지, 작업자

브랜치 형태로 관리

GIT 상태

Git 은 파일을 세가지(Staged, Modified, Committed) 상태로 관리한다.

GIT 의 3 가지 상태

Staged: 현재 수정한 파일을 곧 커밋할 것이라고 표시한 상태
Modified: 수정한 파일을 로컬 데이터베이스에 커밋하지 않은 상태
Committed: 데이터가 로컬 데이터베이스에 안전하게 저장된 상태

GIT 의 영역

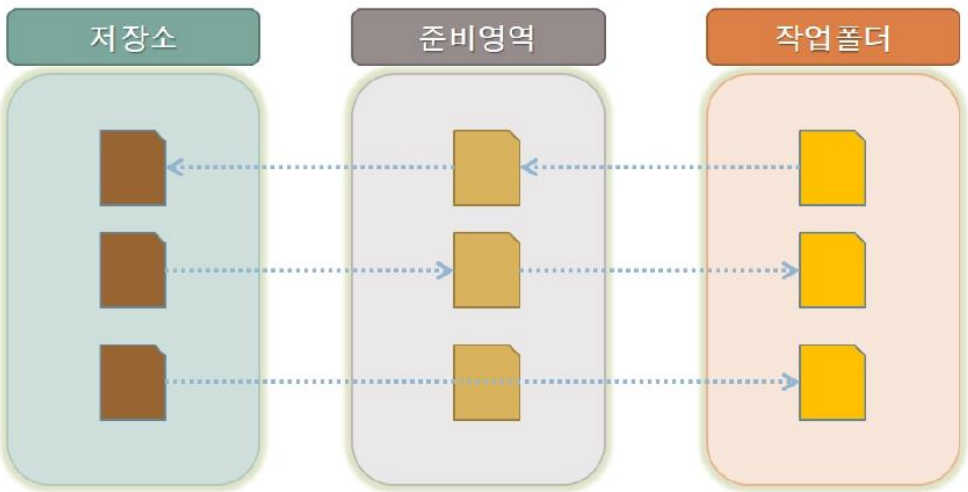
(Working Directory)
프로젝트의 특정 버전을 체크아웃(checkout) 한 것
파일을 수정

(Staging Area)
Git 디렉토리에 있으며 단순한 파일이고 곧 커밋할 파일에 대한 정보를 저장
Index 라고 불림
커밋할 스냅샷을 만듦

(git Directory)
Git 이 프로젝트의 메타데이터와 객체 데이터베이스를 저장하는 곳
파일들을 커밋해서 영구적인 스냅샷으로 저장

깃 작업영역

깃 작업영역



GIT 작업흐름

1. 저장소 만들기

1. 로컬 저장소 만들기(git init)
2. 원격 저장소 만들기
3. 로컬 저장소와 원격 저장소 연결하기(git remote)

2. 파일을 생성하고 커밋하기

1. 로컬 작업 영역에 파일을 생성
2. 파일을 인덱스 영역으로 보내기(git add)
3. 인덱스 영역의 파일을 로컬 저장소에 커밋하기(git commit)

3. 원격 저장소에 저장하기

1. 원격 저장소의 내용과 로컬 저장소의 내용을 병합하기(git pull)
2. 로컬 저장소의 내용을 원격 저장소에 저장하기(git push)

4. 분기(branch)

1. 새로운 브랜치를 생성(git branch)
2. 새로운 브랜치에서 로컬 저장소에서 파일을 수정하거나 생성하기
3. 새로운 브랜치를 병합(git merge)

EXAMPLES

로컬 저장소를 만들고 파일을 생성하여 로컬 저장소에 저장

```
git init
echo "hello world!" > hello.txt
git add hello.txt
git commit "hello.txt 을 새로 생성했습니다"
```

원격의 내용을 복제

```
git clone https://github.com/username/repository.git
```

로컬 저장소를 만들고 원격의 내용을 내려받기

```
git init
git remote add origin https://github.com/username/repository.git
git pull origin master
```

2. COMMIT

Add

작업영역을 준비영역(INDEX)에 반영

Stated 상태로 변경

git add filename

git add filename1 filename2

git add *.cpp

git folder/*

모든 파일(새로운 파일, 수정된 파일)을 Staged 상태로 변경

git add --all

모든 수정된 파일을 Staged 상태로 변경

git add --update

Commit

준비영역(INDEX)을 로컬 저장소에 반영

메시지 EDITOR 를 사용

해당 메시지 첫 줄은 --oneline 출력용(git log --online)

표시로 시작한느 모든 줄은 최종 메시지에서 삭제

```
git commit
```

Stated 상태의 파일을 Commited 상태로 만듦

```
git commit -m "message"
```

```
git add filename
git commit -m "[add] filename"
```

Staging Area 생략하고 commit

준비영역(INDEX)에 등록된 적이 있는 파일만 가능

```
git commit -a -m "message"
```

커밋 메시지 변경

새로운 커밋을 생성하지 않고 가장 최근에 작성한 커밋을 수정

```
git commit --amend
```

☞ git commit 과 같이 지정된 EDITOR 를 실행하여 커밋 메시지 입력

```
git commit --amend -m "message"
```

3. HISTORY

LOG

저장소의 수정 사항을 검토
로컬 저장소에서 현재 체크아웃한 브랜치의 모든 커밋 메시지와 정보를 확인

저장소 히스토리 보기

저장소 커밋 메시지의 모든 히스토리를 시간 역순으로 보기

git log
git reflog

지정된 수만큼 로그를 출력

git log -3
git log -3 --oneline

한 줄로 메시지 보기

git log --oneline
git log --pretty=oneline

커밋의 diff 결과 보여 주기

git log -p
git log -p -2

히스토리 통계(수정된 파일, 얼마나 많은 파일이 변경, 얼마나 많은 라인)

git log --stat

로그 메시지와 하나의 커밋에 포함된 수정사항을 확인

git show 커밋해시
git show ba3784e

TAG

특정한 커밋을 찾아내기 위해 사용(책갈피)

모든 태그 목록 보기

git tag

태그가 적용된 커밋의 상세 정보 보기

git show <tag-name>

Lightweight Tag

Lightweight Tag는 브랜치와 비슷하며 브랜치처럼 가리키는 포인터를 최신 컷밋으로 이동시키지 않음
단순히 특정 커밋에 대한 포인터이다.
Lightweight Tag는 기본적으로 파일에 커밋 체크섬을 저장하는 것뿐이다.

태그 등록

git tag patch-12345

태그 정보 확인

git show pactch-12345

나중에 태그 하기

git tag <태그이름> <**커밋해시**>

git tag pactch-54321 abcdefg

Annotated Tag

Annotated Tag는 태그를 만든 사람의 이름, 이메일, 태그를 만든 날짜, 태그메시지 저장
GPG(GNU Privacy Guard)로 서명할 수도 있다.

태그 등록

git tag **-a** <태그이름> -m "message"

git tag **-a** version-1.0 -m "version 1.0"

나중에 태그 하기

git tag **-a** <태그이름> <**커밋해시**> -m "message"

git tag **-a** version-0.1 a860054 -m "version 0.1"

태그 정보 확인

git show
git show version-1.0

태그 공유하기

태그는 자동으로 리모트 서버에 전송하지 않는다.
별도로 서버에 PUSH 해야 한다.

태그를 원격 서버에 전송
git push origin [태그이름]
git push origin Version-1.0

모든 태그를 원격 서버에 전송
git push origin --tags

태그를 체크아웃

태그가 가리키는 특정 커밋 기반의 새로운 브랜치를 만들어 작업

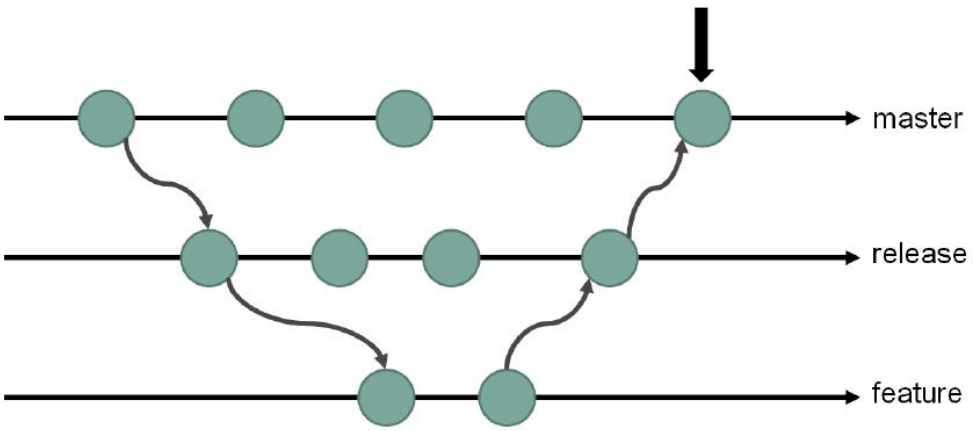
태그를 원격 서버에
git checkout -b [브랜치이름] [태그이름]
git checkout -b release-1.0 version-1.0

4. BRANCH

브랜치

브랜치(Branch)

- 멀티 브랜치
- 다수의 브랜치가 생성



브랜치 확인

로컬 브랜치 목록 출력

```
git branch --list
```

모든 브랜치 목록

```
git branch --all
```

새로운 브랜치 생성

```
git branch <branch-name>
git checkout <branch-name>
git branch release-1
git checkout release-1
```

새로운 브랜치 생성 및 체크 아웃

```
git checkout -b <branch-name>
```

git checkout -b hotfix-1

브랜치 삭제

git branch -d, --delete <branch-name>

git branch -d hotfix-1

브랜치 이름 변경

git branch -m, --move <old-branch> <new-branch>

git branch -m hotfix-1 feature-1

원격 브랜치 가져 오기

원격 브랜치를 가져와 새로운 이름으로 체크아웃

git checkout -b master2 origin/master

5. MERGE

Diff(비교)

변경 내용을 병합하기 전에 비교

git diff <원본브랜치> <대상브랜치>

git diff master hotfix

git diff master origin/master

Merge(병합)

마스터(master)에 브랜치(hotfix)를 병합

git checkout master

git merge hotfix

로컬에 원격 브랜치 병합

git diff master origin/master

git merge origin/master

Conflict(충돌)

병합 취소

git merge --abort

출동 해결

충돌하는 파일을 수정 후 add, commit

git add <conflict-filename>

git commit -m "[merge] message"

Rebase

Reaply commits on top of another base tip

다른 브랜치를 병합할 때 rebase 를 먼저 실행한 후 병합을 하면 이력을 하나의 줄기로 만듦

issue 에 master 를 rebase 한 후

git checkout issue

git rebase master

충돌이 발생하면 충돌 파일을 변경

충돌 부분을 수정 한 후에는 commit 이 아니라 rebase --continue 옵션으로 rebase 수행

git add hello.txt

git rebase --continue 또는 git rebase --abort

master 에 issue 브랜치의 변경 사항을 모두 병합

master 와 issue 는 동일한 HEAD 를 가리키고 있으며 이력이 하나의 줄기로 만들어 짐

git checkout master

git merge issue

병합취소

잘못된 병합 이후 현재 브랜치에 커밋 된 것이 없는 경우

ORIG_HEAD : HEAD 의 이전 포인터(HEAD@{1}, HEAD^, HEAD~)

HEAD^[n] : n 번째 부모, 생략시 1

HEAD~[n] : n 번째 부모의 부모

git reset --merge ORIG_HEAD

git reset --hard ORIG_HEAD

6. CANCEL

Reset

돌아 가려는 커밋으로 리파지토리는 재설정되고, 해당 커밋 이후의 이력은 사라진다.

git reset <option> <commit-id>

1. hard

되돌린 이력 이후의 내용은 지워짐
INDEX 취소, ADD 하지 전 상태(UNSTAGED 상태)
작업영역의 파일 삭제(모두 취소)

git reset --hard <commit-id>

커밋을 취소하고 Unstaged 상태로 되돌리기
git reset --hard HEAD^

2. soft

되돌린 이력 이후의 내용은 보존
해당 내용의 인덱스(스테이지)도 그대로 존재
바로 다시 커밋할 수 있는 상태로 남아 있음
INDEX 보존, ADD 한 상태(STAGED 상태)
작업영역의 파일 보존(모두보존)

git reset --soft <commit-id>

커밋이 된 경우 커밋 이전으로 되돌리기
1. 커밋하기 이전의 Staged(Added) 상태로 돌아 감
git reset --soft HEAD^

2. add 하기 이전의 상태로 되돌리기(Staged -> Unstaged)
git reset filename
git reset HEAD filename

3. mixed

옵션을 적지 않으면 mixed 옵션 적용(디폴드 옵션)
이력은 되돌리며 이후에 변경된 내용에 대해서는 남아있지만 인덱스는 초기화 됨
커밋을 하려면 다시 변경된 내용은 추가해야 하는 상태
INDEX 취소, ADD 하기 전 상태(UNSTAGED 상태)
작업영역의 파일 보존(**기본옵션**)

```
git reset --mixed <commit-id>
```

Unstaged 상태로 변경

```
git reset --mixed HEAD^
git reset HEAD^
```

Unstaged 상태로 변경(마지막 두 개의 커밋 취소)

```
git reset HEAD~2
```

add 취소

```
git reset HEAD
```

Revert

돌아 가려는 커밋으로 리파지토리는 재설정되고, 해당 커밋 이후의 이력은 유지되며 새로운 커밋이 만들어 진다.

한 단계 이전으로 롤백

git revert HEAD

수정하기 이전 상태로 되돌리기

checkout 은 브랜치를 변경하는데 사용되나 변경된 파일을 되돌리는데도 사용

수정하기 이전 초기 버전으로 되돌리기

git checkout filename

파일 삭제

파일 삭제

git rm filename

파일 삭제 취소

git reset HEAD filename

파일을 삭제하여 Unstaged 상태로 만듬(작업영역에 남겨 놈)

git rm --cached filename

파일 이름 변경

파일 이름변경

git mv filename filename2

mv filename filename2

git rm filename

git add filename2

로컬 저장소의 삭제된 파일 복구

삭제된 파일 리스트

git ls-files -d

삭제된 파일 복구

git checkout [files]

삭제된 모든 파일 복구

git ls-files -d | xargs git checkout --

Unstaged 파일 삭제

추적 중이지 않은 파일만 삭제(.gitignore 에 명시된 파일은 지우지 않음)

git clean

파일만 삭제

git clean -f

파일과 폴더 삭제

git clean -f -d

파일과 폴더 및 무시된 파일까지 삭제

git clean -f -d -x

가상으로 실행해 보고 어떤 파일이 지워지는지 알려줌

git clean -n -f

7. REMOTE

git-remote - Manage set of tracked repository

Adds a remote named <name> for the repository at <url>

git remote add <name> <url>

git remote add origin https://github.com/username/repo.git

원격 저장소

원격 저장소 목록 보기

git remote -v

git remote --verbose

원격 저장소 갱신

git remote update

git remote prune origin

git fetch --prune

원격에서 삭제된 브랜치 업데이트

git remote prune origin

git remote update --prune

원격 저장소 브랜치 확인

git branch -r

gir branch --remotes

원격과 로컬의 브랜치 확인

git branch -a

git branch --all

원격 저장소의 브랜치 가져오기

(가져 온 후에 동일한 이름으로 브랜치를 생성하고 체크아웃)

git checkout -t origin/master

원격 저장소의 브랜치 가져오기

(가져 온 후에 이름을 변경하요 브랜치를 생성하고 체크아웃)

git checkout -b master2 origin/master

원격 저장소의 브랜치 확인하기

(로컬에 받아서 확인해 보고 테스트 해 볼 수 있지만 커밋, 푸시할 수 없으며 체크아웃하면 사라짐)

git checkout origin/master

pull

원격 저장소의 내용을 로컬 저장소에 갱신

원격 저장소의 변경 내용을 로컬 작업 영역으로 내려(fetch) 받은 후 병합(merge)

git pull

git pull origin master

git pull --rebase origin master

충돌이 발생하면 수정 후 (git add 또는 git rm)

git rebase --continue

또는

git rebase --abort

원격 브랜치 가져 오기

원격 브랜치를 가져와 새로운 이름으로 체크아웃

git checkout -b master2 origin/master

리모토로부터 가져온 커밋 뒤에 새로운 커밋을 추가

git pull --rebase=preserve origin master

push

로컬 저장소의 변경 내용을 원격 저장소로 보냄

원격 저정소의 업스트림 브랜치 지정하기(한 번 설정하면 그 다음부터는 할 필요가 없음)

git push --set-upstream origin master

원격 브랜치로 전송

git push origin master

새로운 브랜치를 원격에 전송(원격저장소에 새로운 브랜치가 만들어 지고 반영 됨)

git checkout -b master2

git push origin master2

git push origin HEAD:master

origin : 원격 저장소의 별칭

HEAD : 전송할 최종 커밋

master : 원격 저장소의 branch 이름

강제푸시(Force Push)

git push --force origin HEAD:master

git push origin +HEAD:master

원격 브랜치 삭제

git push --delete origin feature

git push [remotename] [:branch]

git push origin :master2

git push [remotename] [localbranch] [:remotebranch]

새로운 브랜치 만들고 원격에 전송

git checkout -b master2

git push origin master2

충돌이 나면 push 하지 않음

git push --force-with-lease origin master

fetch

원격 저장소의 내용을 로컬 저장소에 내려받음

git fetch origin

원격 저장소의 데이터를 가져온 후 병합

git fetch origin master

git merge orgin/master

원격 저장소의 데이터를 가져온 후 로컬의 브랜치가 원격의 이력을 가지도록 변경

로컬에 있는 모든 변경 내용과 확정본을 포기

git fetch origin master

git reset --hard orgin/master

원격 저장소 관리

원격 저장소 살펴 보기

git remote show [remotename]

git remote show origin

원격 저장소 이름 변경

git remote rename [대상이름] [새로운이름]

git remote rename origin origin2

원격 저장소 삭제

git remote rm [remotename]

git remote rm origin

8. STASH

Stashing

커밋하지 않고 나중에 다시 돌아와서 작업을 하고 싶을 때 현재 상태를 저장
Stash 명령은 워킹 디렉토리에서 수정한 파일만 저장
Stash 는 Modified 이면서 Tracked 상태인 파일과 Staging Area 이 있는 파일들을 보관해 두는 장소
아직 끝나지 않은 수정사항을 스택(stack)에 잠식 저장했다가 나중에 다시 작업 할 수 있다.

목록 보기

git stash list

현재 작업 저장

현재 작업을 저장하고 브랜치를 HEAD 로 이동(git reset --hard)
git stash save

저장된 작업 꺼내기

가장 최근에 save 한 stash 가 현재 브랜치에 적용된다.
꺼낸 작업은 list 에서 삭제된다..
git stash pop

가장 최근에 save 한 stash 가 현재 브랜치에 적용된다.
stash pop 과 비슷하지만 list 에서 삭제되지 않고 남아 있는다.
git stash apply
git stash apply stash@{0}

저장된 작업 삭제

stash 가 현재 브랜치에 적용된다.
꺼낸 작업은 list 에서 삭제된다..
git stash drop
git stash drop stash@{0}

전체 리스트를 삭제
git stash clear

9. CONFIG

CONFIG

작업자 이름 설정

git config --global user.name [user name]

작업자 전자메일 설정

git config --global user.email [user email]

글로벌 설정값 확인

git config --global --list

로컬 설정값 확인(저장소별 설정)

git config --list

.gitattributes

Auto detect text files and perform LF normalization

* text=auto

CONFIG

작업자 이름 설정

git config --global user.name [user name]

작업자 전자메일 설정

git config --global user.email [user email]

설정값 확인

git config --global --list

.gitignore

[정규 표현식]

정규표현식이란 애플리케이션과 프로그래밍 언어에 사용할 수 있는 특수한 텍스트 패턴이다.
입력 내용이 텍스트 패턴에 일치하는지 여부를 검사하거나 텍스트를 찾아낼 때 사용한다.

.	개행문자("₩n")를 제외한 임의의 문자 한 개와 매치
*	바로 앞에 있는 패턴이 0 번 혹은 그 이상 반복
[]	대괄호 안에 있는 임의의 문자와 매치하는 문자 클래스 첫번째 문자가 캐럿(^)이면 대괄호 안에 있는 문자를 제외한 어떠한 문자와도 매치 대시(-)가 대괄호 안에 존재하면 대시 앞뒤 문자 사이의 범위를 가리킨다. 예를 들어 [0-9]는 [0123456789]와 동일한 의미이다.
^	행의 시작에서 정규 표현식의 첫번째 문자와 매치
$	행의 끝에서 정규표현식의 마지막 문자
{}	괄호안에 하나 또는 두 개의 숫자를 담고 있을 경우 괄호 안에 있는 패턴이 몇 번 매치해야 있는지를 가리킨다. A{1,3} : A 가 한 번에서 연속 세번까지 나타난 문자열에 매치한다.
₩	매타 문자의 의미를 없애고 C 언어 이스케이프 시퀀스를 표현하는데 사용한다. "₩n"은 개행 문자이지만 "₩*"은 별표 자체를 나타낸다.
+	앞에 있는 정규 표현식이 한 번이나 그 이상 반복하여 매치할 수 있음을 의미한다. [0-9]+ : "1", "111", "123456"에 매치하지만 빈 문자열에는 매치하지 않는다. 플러스(+) 기호 대신 별표(*)를 사용하면, 빈 문자열도 허용된다.
?	앞에 있는 정규 표현식이 0 번이나 한 번 나타날 수 있음을 의미한다. -?[0-9] : 이 패턴은 숫자 앞에 마이너스 기호를 선택에 따라 붙일 수 있는 부호 있는 수를 나타낸다.
|	앞에 있는 정규 표현식 혹은 뒤에 있는 정규 표현식에 매치한다. abc|good|high : 이 패턴은 세 단어 중 하나에 매치한다.
()	정규 표현식 여러 개를 새로운 정규 표현식 한 개로 묶는다.

사이트 : gitignore.io, regexr.com

한글출력

GIT BASH 에서 한글 입력

[.inputrc]

set output -meta on

set convert -meta off

GIT BASH 에서 ls 명령어에서 한글

[.bashrc]

alias ls='ls --show-control-chars'

GIT STATUS, COMMIT 에서 한글

git config --global core.quotepath false

GIT log 에서 한글

git config --global i18n.commitEcoding utf-8

git config --global i18n.logOutputEncoding utf-8

LANG=ko_KR.UTF-8

GIT GLOBAL

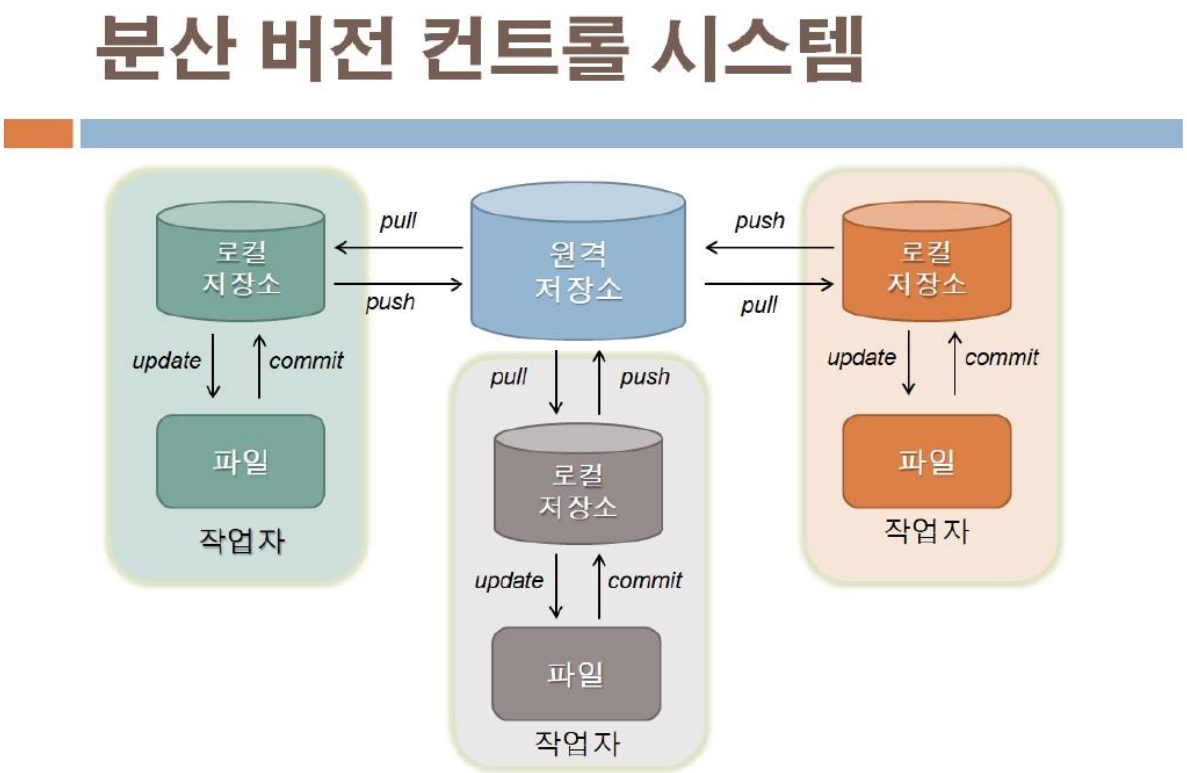
git config --global --list

user.name=username
user.email=username@emailhost.com
core.autocrlf=true
core.filemode=false
core.ui=true
core.editor="C:₩Users₩user₩AppData₩Local₩Programs₩Microsoft VS Code₩Code.exe" --wait
core.quotepath=false
branch.autosetuprebase=always
i18n.commitencoding=utf-8
i18n.logoutputencoding=utf-8

10. GITHUB

GITHUB

분산 버전 컨트롤 시스템



커밋 메시지를 통해 이슈를 완료

이슈 완료 문자

close, closes, closed, fix, fixes, fixed, resolve, resolves, resolved

close #issue

closes #issue

closed #issue