# 1. Introduction to Git

## 1.1 Version Control

팀 단위로 수행하는 프로젝트에 대해서 여러 명이 동시에 작업한 파일은 특별한 장치가 없으면 덧씌워진다. **버전 관리(Version Control), 소스 관리(Source Control)**은 동일한 장비에 대해서 여러 버전을 관리함으로써 소스코드의 변경사항을 추적하고, 대규모 수정 작업을 더욱 안전하게 관리할 수 있다. 대표적인 버전 관리 시스템은 Git이 있다.

## 1.2 Git

커맨드 라인에서 Git은 git 명령어를 통해서 셸에서 실행가능하다. Git을 사용하기 위해 필요한 첫번째 방법은 **저장소(repository)를** 초기화 하는 것이다. git init 명령어는 파일 시스템 내부의 디렉토리를 깃 저장소로 초기화 시킨다.

```
user@host: /home$ git init
```

## 1.3 States in Git

파일은 Git 내부에서 3가지의 상태를 가질 수 있다.

1. committed : 현재 버전의 파일이 커밋되었으며, Git에 저장되어 있음
2. staged : 파일이 다음 커밋에 포함되어 있으며 아직 커밋되지는 않았음, Staging Area 내부에 존재
3. modified : 파일이 지난 커밋 이후로 변경되었으며 아직 staged 되지 않았음

Git repository를 변경하고 나서 git status 명령어를 통해 repository 내부의 파일의 상태를 확인할 수 있다. 파일을 변경하고 커밋할 준비가 되었다면 git add를 통해 Staging Area로 파일을 전송하고 커밋한다.

```
user@host: /home$ git status
user@host: /home$ git add *
user@host: /home$ git commit -m "what is changed/when is changed"
```

## 1.4 Commits

Git repository를 초기화 하면 해당 폴더 내부에 .git 폴더를 생성한다. 전형적인 Git의 워크플로우는 파일을 변경한 체크포인트를 저장하게 된다. 이러한 체크포인트를 **커밋(Commit)**이라고 한다. 모든 커밋은 모든 파일을 새로 저장하는 방식이 아닌 각각의 커밋 별 차이(**diff**)를 추가해서 저장하는 방식을 사용한다. 따라서 커밋은 유저들이 생성한 변경점을 통합하는 강력한 방법을 제공하게 된다.

커밋은 특정 시간에 저장한 파일의 스냅샷을 저장한다. 이러한 스냅샷을 기록함으로써 이전의 스냅샷으로 돌아가서 다른 사람들의 변경 내용과 합칠 수 있다. 커밋 명령어의 -m 옵션은 텍스트 메시지를 커밋에 붙일 수 있다. 주로 변경 사항과 시간을 첨부한다.

```
user@host: /home$ git commit -m "what is changed/when is changed"
```

## 1.5 git config

처음 커밋을 실행 무리는 Git에게 커밋의 주체가 누구인지에 대한 정보를 제공해야 한다. 이 단계는 프로젝트 팀원들에게 특정 커밋의 소유주가 누구인지 알려주게 된다. git config –global 명령어는 email과 user name의 정보를 입력한다.

```
user@host: /home$ git config --global user.email "your.email@domain.com"
user@host: /home$ git config --global user.name "Your name"
```

## 1.6 git diff

diff 명령어는 현재 커밋과 이전 커밋의 차이를 추적하여 출력한다. 파일을 stage 한 이후의 차이를 확인하고 싶으면 --staged 옵션을 사용한다.

```
user@host: /home$ git diff
```

## 1.7 git log

git log 명령어는 repository에 커밋된 모든 기록을 가져온다.

```
user@host: /home$ git log --stat
```

# 2. Git Remotes

## 2.1 Github

**Github**은 Git기반으로 설계된 GUI로, Git과 함께 사용할때 매 매우 유용하다. Github는 **로컬 저장소(Loacal Repository)**와 **원격 저장소(Remote Repository)**에 저장하게 된다. Github를 사용하게 되면 본인의 코드를 다른 사람들과 공유할 수 있고, 프로젝트 내부에서 협업할 수 있으며, 다른 사람의 코드를 다운로드 할 수 있다.

## 2.2 git clone

원격 저장소에 있는 데이터를 다운로드 받기 위해선 git clone 명령어를 통해 가능하다. 원격 저장소의 데이터를 다운받고 변경한 후에 커밋하게 되면 로컬 저장소는 원격 저장소보다 한 커밋 앞서 있게 된다.

```
user@host: /home$ git clone https://github.com/username/repository_name
```

## 2.3 branch/main/origin

모든 Git 저장소는 하나 이상의 **브랜치(branch)를** 가지고 있다. 각각의 브랜치는 서로 다른 버전의 코드를 포함하고 있으며 원격 저장소의 브랜치 중 가장 주요 브랜치를 main(master) 라고한다. 개발자는 서로 다른 브랜치를 생성하고 커밋한 후 다시 main(master)로 돌아오게 된다. 즉 main(master) 브랜치는 어떤 프로젝트의 가장 최근에 공유한 버전이다.

git branch는 저장소 내부에 있는 모든 브랜치를 리스트 업 해준다. 또한 현재 활성화 중인 가지에 대해서 * 를 붙어 강조한다.

```
user@host: /home$ git branch
```

## 2.4 git push

로컬 저장소에서 변경을 변경했다면 다시 원격 저장소에 업데이트하여 다른 사람과 공유할 필요가 있다. 로컬에서 작업한 내용은 로컬 저장소에만 반영이 되어 있기 때문이다. 해당 작업은 git push 명령어를 통해 가능하다.

git push를 사용할때 저장소의 이름과 브랜치의 이름을 명시해줘야 한다. 원격 저장소의 코드를 clone 하면 Git은 자동적으로 원격 저장소의 이름을 origin 으로 저장한다. 최신 버전에 모드를 업데이트 할 경우, main 브랜치에 커밋한다.

```
user@host: /home$ git push origin [branch]
```

## 2.5 git show

git log 명령어는 커밋 별로 가지고 있는 해시 데이터를 출력한다. 해당 해시를 사용해 이전의 커밋으로 돌아갈 수 도 있고 특정 커밋이 어떻게 변했는지 확인할 수 있다. git show 명령어는 이전 커밋과 특정 커밋의 변경 사항을 출력한다.

```
user@host: /home$ git show [commit hash]
```

## 2.6 git diff

git diff에 서로 다른 커밋의 hash 값을 입력하면, 두 해시간의 변경 사항을 출력한다. git diff 명령어는 모든 해시값을 입력할 필요없이 5~6글자만 입력하면 자동으로 해시값을 찾는다.

```
user@host: /home$ git diff [hash1] [hash2]
```

## 2.7 git reset

커밋의 해쉬는 영구적인 값이며, Git은 로컬 저장소와 원격 저장소의 해쉬를 보관하고 있다. 원격 저장소에서 로컬 저장소로 c12 커밋을 불러올 경우 로컬 저장소에는 c12 커밋이 저장되게 된다. 파일의 변경을 마친후 커밋하고 푸시하면 로컬 저장소에는 c12, c13 커밋이 존재하게 된다. 즉, git clone을 통해 저장한 데이터는 원격 저장소의 기존 커밋을 불러온다.

git reset 명령어는 입력받은 해시값으로 작업 디렉토리에 있는 모든 사항을 변경한다. --hard 옵션은 작업 디렉토리와 깃 히스토리를 특정 상태로 초기화 시킨다. --soft 옵션은 깃 히스토리만 리셋하게 된다.

```
user@host: /home$ git reset --hard [hash]
```

## 2.8 git pull

원격 저장소가 변경되었을 경우 원격 저장소의 변경 사항을 로컬 저장소에 반영해야 한다. git pull 명령어는 원격 저장소의 업데이트 내용을 로컬 저장소로 업데이트한다.

```
user@host: /home$ git pull
```

## 2.9 HEAD refernce

git log 명령어를 통해 커밋의 해시를 확인하고 커밋을 변경하는 작업은 귀찮다. HEAD reference는 해시값을 최근 커밋 부터 HEAD~n 형태로 추상화 시켜 사용할 수 있도록 한다. 해시 값을 알고 싶다면 git rev-parse HEAD~n을 통해 정확한 해시값을 가져올 수 있다.

```
user@host: /home$ git rev-parse HEAD~5
user@host: /home$ git reset --soft HEAD~3
```

# 3. Git Branches

## 3.1 Merge Conflicts

**병합 충돌(Merge Conflicts)**은 원격 저장소의 이전 커밋에 대해 서로 다른 커밋을 동시에 푸시하려고 시도라 때 발생하는 충돌이다. Git은 서로 다른 커밋들이 어떤 것이 "진실"인지 판단할 수 없기 때문에 병합 충돌이 발생하게 된다. Git은 병합 충돌을 예방하기 위해 **브랜치(Branch)** 개념을 사용하게 된다.

## 3.2 Create branches

브랜치는 동일한 저장소에 대해서 서로 다른 작업을 수행할 수 있도록 한다. 따라서 프로젝트 내부에서 변경 사항을 만들 때 새로운 가지를 생성하고 변경 작업이 끝났을 때 main(master) 브랜치로 병합하는 작업을 수행한다. 서로 다른 가지를 생성하고 원격 저장소에 푸시하게 되면 Git은 병합 충돌이 발생하지 않게 분산하여 저장하게 된다.

브랜치를 생성하는 방법은 두가지가 있다.

- git branch [new_branch] + git checkout [new_barnch]
- git checkout -b[new_branch]

git checkout 명령어는 뒤에 입력된 브랜치로 이동한다. git checkout -b 옵션은 브랜치를 생성하고 자동으로 이동하도록 지시한다.

```
user@host: /home$ git branch [new_branch]
user@host: /home$ git checkout [new_branch]

user@host: /home$ git checkout -b [new_branch]
```

## 3.3 Push branches

로컬 저장소의 브랜치의 변경 사항을 원격 저장소에 반영하기 위해 git push 명령어를 브랜치 명을 바꿔 푸시해야 한다.

```
user@host: /home$ git push origin [new_branch]
```

## 3.4 Merge branches

Github에 제공되어 있는 소스코드는 수많은 가지가 존재하지만 사용자들이 다운로드 하는 곳은 main branch의 커밋이다. 즉, 다른 가지에서 개발자들이 작업한 사항은 반드시 main(master) 가지로 합쳐져야 한다. Merging은 다른 가지의 커밋을 다른 가지로 병합하는 작업을 수행한다. git merge 명령어는 한 가지를 다른 가지로 합치는 명령이다. 합쳐지는 부모 가지로 반드시 이동해서 병합을 수행해야 한다.

```
user@host: /home$ git checkout main
user@host: /home$ git merge [new_branch]
```

## 3.5 Delete branches

병합이 끝난 가지나 사용할 일이 없는 가지는 삭제해야 한다. git branch -d 명령어는 뒤에 브랜치 명을 입력하면 해당 브랜치를 삭제한다. 단약 병합하지 않는 가지를 삭제하려면 -D 옵션을 사용한다.

```
user@host: /home$ git branch -d [branch]
```

## 3.6 Fetch branches

다른 사람이 원격 저장소에서 작업하는 것을 확인하기 위해 푸시한 가지를 로컬 저장소에서 다운로드 받을 때는 git clone이나 git pull이 아닌 git fetch 명령어를 사용하면 된다. git clone은 최초에 원격 저장소에 존재하는 커밋을 다운로드 할때, git pull은 main(master)의 커밋을 로컬 저장소로 불러올때, git fetch는 모든 브랜치에 대해 로컬 저장소에 반영할때 사용한다.

```
user@host: /home$ git fetch
```

## 3.7 Check difference between branches

git diff 명령어에 서로 다른 두 브랜치를 입력하면 두 브랜치 사이의 변경 사항을 출력한다.

```
user@host: /home$ git diff [branch1] [branch2]
```

## 3.8 Git workflow

1. 메인 브랜치에서 특정의 이름을 딴 브랜치를 생성한다
2. 브랜치에서 변경 사항을 추가하고 커밋을 생성한다
3. 원격 저장소에 브랜치를 푸시한다
4. 다른 사람들과 브랜치를 리뷰하고 평가한다
5. 메인 브랜치로 병합한다
6. 분리된 브랜치를 삭제한다

```
# Create and Change branches
user@host: /home$ git checkout -b new_branch

# Check current branches (Check every steps)
user@host: /home$ git branch -[rla]

# Send changes to staging area and commit to local repository
user@host: /home$ git add .
user@host: /home$ git commit -m "There is some changes."

# Push commits to remote repository
user@host: /home$ git push origin new_branch

# Return to main branch and merge branches
user@host: /home$ git checkout main
user@host: /home$ git merge new_branch

# Delet branches
user@host: /home$ git branch -d new_branch

# Get other branches from remote repo
user@host: /home$ git fetch
user@host: /home$ git checkout other_branch
user@host: /home$ git add .
user@host: /home$ git commit -m "This is change for other_branch"
user@host: /home$ git push origin other_branch
```

## 3.9 Type of branches

브랜치의 이름은 파일이 어떤 목적을 위해 사용되고 있는지를 중심으로 결정한다.

- Feature(개발 사항) : feature/feature_name
- Fix(수정) : fix/error_name
- Chor(관리) : chore/chore_name

# 4. Merge Conflicts

## 4.1 Merge Conflicts

병합 충돌의 예러문은 다음과 같다.

```
user@host: /home$ git merge feature/queen-bot

Auto-merging bot.py
CONFLIT (content) : Merge conflict in bot.py
Automatic merge failed; fix conflicts and then commit the result
```

## git merge --abort

gut merge --abort 명령어는 Merge conflict가 발생하기 이전의 커밋으로 되돌아간다. git reset 명령어와 비슷하다.

```
user@host: /home$ git merge --abort
```

## 4.2 Fix the merge conflict

Merge conflicts가 발생하고 해당 파일을 수정하려 들어가면 화살표 표식으로 이전에 합쳐졌던 브랜치와 합치려는 브랜치가 어떻게 충돌하고 있는지 표기해 준다. 표식은 저장으로 저자오디가 때문에 표식을 모두 제거하고 원하는 코드를 수동으로 작성함히 원격 저장소에 커밋하는 방법이 있다.

## 4.3 git mergetool

프로젝트의 특성은 수많은 파일로 구성되어 있다는 것이다. 변경 사항은 한 파일에만 국한되지 않고 여러 파일을 변경하게 된다. 즘 더 효율적인 수정 작업은 git mergetool 를 통해 GUI에서 코드를 수정할 수 있다. DiffMerge라는 툴은 좌측에 충돌하는 커밋을, 우측에는 기존에 병합되어 있는 커밋의 코드를 띄워 병합을 좀 더 편하게 수정할 수 있도록 작업할 수 있다.

```
user@host: /home$ git mergetool --tool=[tool_name]
```

## 4.4 ours/theirs

두 브랜치에 대해서 병합 충돌이 발생했을 경우 git check 명령어의 --ours, --theirs를 통해 어떤 브랜치가 진짜인지를 Git 엔진에 알려줄 수 있다.

- ours : 현재 병합이 되어있는 브랜치
- theirs : 병합을 하고자 하는 브랜치

```
# file1.txt에 대해서 현재 병합 중인 브랜치의 버전을 사용하고 병합을 하고자 하는 브랜치의 변경 사항은 무시한다.
user@host: /home$ git checkout --ours --file1.txt

# file1.txt에 대해서 현재 병합 중인 브랜치의 변경사항은 무시하고 병합을 하고자 하는 브랜치의 변경사항을 적용한다.
user@host: /home$ git checkout --theirs --file1.txt
```

## 4.5 Remove file from Git

git rm 명령어는 저장소의 파일을 삭제하는 명령어이다. --chached 옵션은 원격 저장소의 파일만 삭제한다.

```
user@host: /home$ git rm --cached .DS_Store
```

## 4.6 git ignore

프로젝트에 유용하지 않고 해쉬 자주 변경되어 병합 충돌을 야기할 수 있는 확장자들이 간혹 존재한다. 대표적으로 .DS_Store과 .pyc가 존재한다. 가장 간단한 방법은 Git 엔진에 해당 확장자를 무시하고 커밋을을 하라고 알려주는 방법이다. .gitignore 파일 내부에 해당 확장자를 입력하면 커밋을 진행할 때 자동으로 무시하고 커밋을 진행한다.