1.

여러분 깃에 대해 들어보셨나요? 아마도 깃허브에 대해서는 들어보셨을 것입니다.
깃은 프로그래머로서 어떤 언어를 사용하건 필수적으로 익혀야 할 어떻게 보면 기본 소양이라고 할 수 있습니다.

깃은 일종의 버전 관리 시스템으로서 버전 관리 시스템(version control system)이라는 말을 처음 들어보신 분들도 계실텐데 버전 관리 시스템이란 게임에서의 세이브 로드와 비슷한 기능입니다. 여러분들이 코드를 짤 때 짠 코드에 대해 세이브와 로드 기능을 제공한다는 것은 필수적입니다.

예를들어 c로 계산기를 만들어 제출하는 과제가 부여되었다고 가정해봅시다 일단 기본적으로 더하기 빼기 곱하기 나누기 등등의 기능을 만들어서 제출하면 되겠죠? 그래서 제출했는데 교수님께서 더하기 빼기는 필요없고 루트연산이나 제곱 연산을 계산하는 코드를 추가해라 < 라는 추가적인 코멘트가 왔다면 우리는 일반적으로 원래 만들었던 코드를 수정해 즉 원본 파일을 수정해 다시 제출하게 됩니다. 근데 또 교수님께서 다시 생각해보니 더하기 빼기만 있으면 괜찮을 것 같다 라고 더하기 빼기 코드를 추가하라고 요청이 다시 오게 되면 우리는 그 더하기 빼기 하는 코드를 재입력해서 제출해야 합니다 .. 지금은 이렇게 학생과 교수의 관계라고 설정했지만 앞으로 회사나 고객을 상대하는 프로그래머로서 고객의 요구사항의 변동이나 코드 작성 시 발생할 수 있는 각종 버그들에 대해 적절하게 대응하기 위해 이런 버전 관리 시스템이 필요한 것입니다.

앞서 보셨듯이 버전 관리 프로그렘에는 다양한 종류가 있지만, 그 중에서도 git은 많은 회사에서 사용하고 있고, 지속적으로 사용자가 증가하는 추세에 있어 git을 익히게 된다면 앞으로 그룹 프로젝트나 그 이후에 대해서 적응하기 쉬울 것입니다.

이번 시간에는 그중에서 가장 일반적으로 사용되는 git에 대해서 알아보고 github 가입 및 git을 사용하기 위한 기본적인 환경을 만들어 보겠습니다.

깃은 분산 버전 관리 시스템입니다 분산 버전 관리 시스템이란 사용자 컴퓨터 내에 서버의 백업본을 가지는 구조입니다.

분산 버전 관리 시스템에 대해 알기 위해서는 로컬 버전 관리 시스템과 중앙집중식 버전 관리 시스템에 대한 이해가 필요합니다.

깃과 같은 vcs과 없었던 시절 깃과 같이 버전 관리를 하려면 앞에서도 확인해 봤듯이 해당 파일을 복사해 시간을 기록하는 방식을 많이 사용하게 됩니다. 혹은 버전을 직접 작성할 수도 있겠죠? 이 방법은 용량을 많이 필요로 할 뿐더러 파일을 잘못 수정하거나 삭제할 수도 있습니다, 이를 해결하기 위해 로컬 vcs가 등장하게 됩니다.

로컬 컴퓨터 내에서 데이터베이스를 사용해 버전들을 관리하게 됩니다.

다만 이런 로컬 파일 관리 시스템에도 문제가 있죠 대형 프로젝트의 경우 즉 많은 개발자가 일련의 파일을 다루게 될 때 파일의 공유가 불가능하다는 점이 문제라고 할 수 있겠습니다.

그래서 중앙집중 버전 관리 시스템이 등장했습니다.

중앙집중 버전관리란 데이터베이스를 서버에 올려서 파일을 관리하고 클라이언트가 서버에서 파일을 받아 사용하는 방식입니다.

하지만 이런 중앙 집중 버전 관리 시스템에도 문제가 있습니다. 서버가 다운되는 상황이 발생하게 되면 시간적인 제약이 생기고 사람들이 로컬에서 하는 일을 백업할 수 없게 됩니다.

그래서 분산 버전 관리 시스템이 생기게 되었습니다.

분산 버전 관리 시스템은 앞서 봤던 로컬 vcs와 중앙 집중 vcs가 합쳐진 구조라고 볼 수 있습니다.

git이 바로 이 구조로 각각의 장점을 합쳐서 사용할 수 있습니다.
사용자 컴퓨터에서 로컬로 개발을 하며 이를 중앙 서버로 올릴수도 있고 중앙 서버의 진행상황을 사용자의 컴퓨터로 갱신할수도 있습니다.
따라서 중앙서버에 문제가 생겨도 로컬로 복구하기가 쉽고 서버나 로컬의 데이터가 날라가더라도 복구가 가능합니다.

이제 좀 더 자세하게 살펴보겠습니다. 다음 그림을 보시면 로컬 저장소 영역에 working directory, staging area, repository로 나뉘어 있는데 굳이 따지자면 저 모든 작업들 모두는 working directory 내부에서 이루어집니다. working directory 내부에서 git에 의해 버전관리를 받기 위해서는 repository에 저장해야 합니다. 그러기 위해서 stage area에 올려야 하는데 stage area는 잠깐 담아 놓을 수 있는 바구니라고 생각하시면 됩니다. working dir 내부의 파일들을 선택해 stage area에 add해서 올리고 그것을 commit 해서 local repository에 해당 파일들의 버전을 저장한다고 생각하시면 되겠습니다.

이번엔 조금 다른 그림으로 살펴보겠습니다. git의 파일들을 관리하는 공간으로는 작업 폴더, index, head로 나누어 볼 수 있습니다 head는 commit을 가르키는 포인터로 그냥 commit이라고 생각하시면 됩니다. commit에 대해 설명하자면 앞으로도 commit이라는 말을 많이 듣게 될 것입니다. commit했다는 것은 commit했을 당시 stage area에 add된 파일들을 git 내부 저장소에 저장했다는 것을 말합니다 다르게 말하자면 commit했을 당시의 staged된 파일들의 해당 버전을 저장한 것을 의미합니다. 그냥 버전이라고 생각하셔도 됩니다. 다음으로 git의 파일들의 상태로는 크게 tracked 파일 untracked 파일로 나누어 볼 수 있습니다 untracked 파일은 git의 의해 관리되지 않는 파일로 add가 되기 전의 상태라고 생각하시면 되겠습니다. add가 되게 되면 staged 상태로 넘어가서 commit으로 해당 버전으로 저장이 되고 이후 변경이 일어나느냐 아니냐에 따라 modified와 unmodified 로 나뉘게 됩니다.

그렇다면 git에서 파일이 어떻게 add되고 commit되어서 버전관리가 일어나는지 그림을 통해 알아보겠습니다.

첫 번째 왼쪽 최상단에 위치한 그림을 보시면 working dir 내부에 file.txt v1이 존재하는 것을 확인해 볼 수 있습니다 이 파일은 지금 git에 의해 관리받지 않는 즉 untraced 파일입니다 이후 두 번째 그림에서 add가 되게 되면 index에 올라가게 되고 staged 상태가 되게 됩니다. 다음으로 commit을 하게 되면 head에 즉 하나의 버전으로 저장되게 되고 해당 버전을 head가 가리키게 됩니다.

이제 아래로 내려가보겠습니다. 이제는 v1을 v2로 수add commit을 하게 되면 새로운 commit 즉 버전이 생기고 head가 그 버전을 가리키게 되는 구조로 이해하시면 되겠습니다.

정리해보자면 commit 은 하나의 버전이고 버전이 새로 추가될 때마다 head라는 포인터가 해당 commit을 가리키는 구조입니다. 일반적으로 local에서 혼자 개발하는 상황에서는 대체로 여기서 잘 벗어나지 않습니다.

하지만 다른 사람과 같이 작업하게 된다면 분업화를 하는 것이 효율적이고 합리적입니다. 그렇다면 어떻게 동일한 코드를 각자 분업화해서 개발할 수 있을까요?

여기서 branch라는 개념이 등장합니다. branch는 영어로는 가지, 분기라는 뜻이죠 하나의 프로젝트를 진행할 때 main이라는 중심줄기가 협업하는 프로젝트라고 할 수 있고 각각의 branch가 이제 팀원이라고 생각할 수 있습니다 공동의 repo를 master branch로 하여 공유하며 프로젝트가 진행됩니다.

이제 로컬과 원격을 합쳐서 정리해보겠습니다.

여러분이 팀프로젝트를 시작하려고 합니다. 일단 팀장이 기본적인 구조를 설계하고 팀 멤버들에게 특정 부분의 코드를 작성하라고 임무를 부여했다고 가정하겠습니다.

그림을 보시면 팀장이 로컬에서 add하고 commit 해서 push해 remote repository인 github에 업로드한다면 팀 멤버는 그 코드를 fork 해와서 fork는 타인의 원격 repo를 내 원격 repo로 가져오는 행위라고 생각하시면 됩니다. 코드를 fork해와서 팀 멤버 자신의 remote repository에 가져오고 팀 멤버 본인의 github에서 clone해 와 로컬에 저장하게 됩니다. 이후 팀 멤버가 request 받은 코드를 추가해 pull request를 하면 팀장은 그 코드를 받아서 확인 후 merge하면 팀 멤버의 코드가 팀의 작업물에 추가되게 됩니다.

이 구조가 지금 당장은 익숙하지도 않고 복잡하기 때문에 이해가 가지 않을 수 있습니다.
이런 구조는 다음 시간에 실습을 하며 그림을 보며 이해하면 git을 잘 사용할 수 있게 될 것입니다.

이제 깃을 사용하기 위해 설치하며 기본적인 설정들을 해 보도록 하겠습니다.

깃 홈페이지에 가 깃을 설치해주세요 그냥 다음 다음 다음 해서 설치하면 됩니다. 설치가 잘 되었는지 확인하는 방법은 git bash를 실행한 후 git —version을 입력해 결과로 git의 버전이 출력된다면 올바르게 된 것입니다.

다음은 sourcetree입니다 sourcetree는 cli 커맨드 라인 인터페이스가 아닌 gui 그래픽 유저 인터페이스로 git을 관리하는 소프트웨어로 파일들의 전체적인 구조를 확인하는데 유용합니다. 일반적으로 터미널에서 명령어와 sourcetree를 병행해서 사용하게 됩니다.

다음으로는 vscode를 설치해 보도록 하겠습니다 vscode의 기본적인 사용방법은 생략하겠습니다. vscode의 기본 터미널을 git bash로 바꿔 보겠습니다. 상단에 보시는 검색창에 f1을 누르고 se만 입력해도 select default profile을 클릭한 후 git bash를 클릭해 기본 터미널을 설정해 줍니다. 이 터미널에서 git 명령어들을 실행해 볼 수 있다.

이후 원격 repo인 github을 가입해 보도록 하겠습니다.
일반적인 웹사이트와 가입하는 법은 동일하지만 중요하게 다른 점이 있습니다 회원가입을 진행하다 보면 github pro 라고 있는데 github pro 로 가입하시는 편이 좋습니다. 왜냐하면 github에서는 학생들을 위한 무료 혜택을 주고 있는데 students로 가입하면 github copilot을 사용할 수 있고 각종 혜택들을 받을 수 있습니다. 학생 인증을 받는 방법은 간단합니다. 여러분 입학하실 때 학교 email이 생성되었을 것입니다. 학교 email을 사용해 가입하거나 본인 email을 이용해 가입 후 학교 email을 추가할 수도 있습니다.

이후 블로그에 들어가 결재 방식, 2차 인증 등등의 설정을 하고 copilot 까지 설치

2.

이번 시간에는 git의 간단한 명령어들을 실행해보며 git의 구조를 이해해하고, git으로 파일들을 관리해보겠습니다.

일단 git의 기본 설정을 하겠습니다 git bash를 실행하고 git config — global user.name 다음에 본인의 이름을 user.email으로 본인의 email을 설정해 git을 사용할 때 본인의 연락처와 이름을 지정해 누가 git을 설정했는지 확인해 볼 수 있습니다. 이렇게 하는 이유는 commit 당시 변경사항을 누가 작성했고 연락처는 무엇인지 알리기 위해 설정하게 됩니다.

다음으로 vscode를 실행해주시길 바랍니다. vscode에서 여러분들이 주로 작업 디렉토리를 열어 git으로 관리할 폴더를 지정해줍니다.

vscode를 실행하게 되면 아래에 프레젠테이션의 아래에 있는 부분과 동일한 창이 있을 것입니다. 그 부분에 git status를 입력해보시면 이 메시지는 지금 git에 의해 관리되지 않는 폴더라는 메시지가 출력되는 것을 확인해 볼 수 있습니다. git status는 git의 상태를 알려주는 명

령어입니다.

git init을 하게 되면 숨김 파일인 .git 폴더가 생기면서 해당 디렉토리 내에서 파일들을 관리하게 됩니다.

다음으로 git에서 파일을 제외하는 방법에 대해서 알아보겠습니다. 원격 저장소 예를 들어 깃허브 같은 공용 공간에 코드를 올리게 될 때 자동으로 생성되거나 다운로드 되는 파일들 (라이브러리, 빌드 결과물) 보안상 민감한 파일들이 함께 올라가게 되면 괜히 용량만 차지하고 보안상 위험성을 방지하기 위해 필요 없는 파일들을 제외시키려고 할 때 사용합니다.

깃은 이런 기능을 ,gitignore 파일을 이용해 제어합니다. 그럼 일단 무시하고픈 파일을 하나 생성해 봅시다. 제 경우에는 개인정보가 들어있는 ignore.py 파일을 만들었습니다. 또 이름이 .gitignore이라는 파일을 만들어보겠습니다. gitignore 안에 무시하고자 하는 파일의 이름을 넣게 되면 git status로 확인해 봤을 때 감지가 되지 않는 것을 확인해 볼 수 있습니다.

gitignore에 대한 다양한 사용방법들이 있지만 간단한 것만 알아보면 파일이름이나 디렉토리 이름, 경로 등등으로 파일을 ignore 할 수 있습니다.

다음으로 add 명령어에 대해 알아보겠습니다. add 명령어는 파일을 commit으로 저장하기 전에 index에 등록하는 단계입니다. git add를 사용해 add 다음에 파일 이름을 넣거나 .을 통해 해당 디렉토리 내 모든 파일을 index에 올릴 수 있습니다.

이렇게 staged 된 파일들을 commit 명령어를 사용해 저장할 수 있습니다. commit 은 하나의 버전이라고 생각하시면 됩니다. 왼쪽에 그림을 보시면 커밋에 대해 설명해주는 그림입니다 첫 commit을 하게 되면 main 또는 master이라는 branch가 생기게 되는데 이 중심 줄기에서 commit이 마치 굴비 엮이듯 연결되어 있는 구조입니다.

git commit을 하게 되면 vi 창이 열리게 되고 저같은 경우는 첫 번째 커밋이라고 입력해 저장했습니다.

:wq 로 저장하고 나오면 commit이 완료되었다는 메시지를 확인해 볼 수 있고 git log 명령어를 통해 git의 로그를 확인해 볼 수 있습니다. commit이 정상적으로 되었고 누가 작성했는지 언제 작성했는지 나오는 것을 확인해 볼 수 있습니다. 여기서 노란색으로 commit 오른쪽에 나오는 일련의 문자열은 hash로 각각의 버전을 특정짓는 식별 문자열이라고 생각하시면 되겠습니다.

이렇게 첫 번째 커밋을 하게 된 상태는 왼쪽 아래에 있는 그림과 같습니다. 첫 번째 커밋이라고 하는 버전이 생기게 되고 이 버전을 master가 가리키고 있고 그 master 을 head가 가리키고 있는 구조입니다.

여기서 변경사항을 줘서 커밋을 더 해보도록 하겠습니다. 다음과 같이 하나의 커밋을 추가했

습니다. 이렇게 두 번째 커밋을 추가하면 git의 마스터는 오른쪽 아래처럼 두 번째 커밋을 가리키게 됩니다. 이렇게 add, commit 해서 파일들을 버전별로 관리해 볼 수 있습니다.

이제 버전을 돌아가는 방법을 알아보겠습니다. 버전을 돌아가는 방법으로는 두가지 방법이 있습니다.
첫째로 revert입니다. 저와 같은 경우는 다음과 같이 4개의 커밋을 해 놓고 revert를 진행하였습니다. 각각의 파일들은 first second third fourth 로 새로운 파일을 만들어 add하고 commit했습니다.

revert 명령어는 git revert HEAD^(캐럿) 혹은 git revert hash로 사용할 수 있습니다. revert는 해당 commit을 취소한다고 생각하시면 됩니다. git revert HEAD^을 하게 되면 이전에 커밋했던 것과 같이 vi 창이 열리는데 자동으로 revert "해당 커밋 내용" 이 입력된 채로 있습니다. 콜론 wq를 해 저장하고 나가주면 해당 커밋 내용이 삭제된 새로운 커밋이 생기게 됩니다.

구조에 대한 그림을 살펴보면 첫 번째 커밋에서는 1st라는 파일 하나가 존재했고 이후 4번째 커밋에서는 4개의 파일이 존재했는데 git revert HEAD^을 하면 현재 커밋 즉 4번째 커밋의 바로 이전의 커밋을 삭제한다고 지정할 수 있습니다. git revert HEAD^^(캐럿)을 하게 되면 현재 커밋의 2번 앞의 커밋을 삭제할수도 있습니다.

다음으로 reset입니다. git reset은 git reset 옵션 HEAD^ 혹은 git reset 옵션 hash로 사용할 수 있습니다. reset을 사용하게 되면 해당 커밋 위치로 이동하는 것을 확인해 볼 수 있습니다.

구조에 대한 그림을 살펴보면 이전과 동일한 구조에서 head와 master가 이동한 것을 확인해 볼 수 있습니다. 이렇게 head가 가리키는 위치가 변함으로서 그 이후의 히스토리는 없어지게 됩니다. 하지만 히스토리가 사라졌다고 해당 커밋이 사라진 것은 아닙니다. hash 값으로 해당 커밋에 접근해 이어져있는 히스토리를 되살릴 수도 있습니다.

이제 revert와 reset을 비교하여 정리해보겠습니다 reset은 head를 조정해 과거의 특정 커밋으로 되돌아가는 것이고 revert는 과거의 커밋을 삭제한 새로운 커밋을 생성하는 행동입니다. 각각 장단점이 있습니다. 일단 reset은 커밋 히스토리를 깔끔하게 만들 수 있고 revert에서 발생할 수 있는 충돌 문제에 비교적 자유롭습니다. 하지만 동일한 branch에서 작업하는 파트너가 있다면 커밋 히스토리를 지운다는 점은 굉장히 치명적입니다. 일단 원격 저장소에 저장되어 있는 git 히스토리와 로컬에서 변경한 git 히스토리가 달라지게 되고 충돌이 일어나게 됩니다.

그림을 통해 알아보겠습니다. 다음과 같이 원격 저장소에 저장되어 있는 git에 a와 b가 공동된 작업을 하고 있는 상황을 가정하겠습니다. a유저가 로컬에서 4번 커밋을 reset해서 3번 커밋까지만을 원격 저장소에 올리고 난 후 b 유저가 해당 커밋을 삭제한 줄 모르고 원격 저장소에 다시 4번째 커밋까지 저장되어 있는 로컬의 git을 올리게 되는 상황이 발생할 수 있고

삭제한 커밋의 코드와 이후에 작성한 코드의 충돌이 발생할 수 있습니다. 따라서 타 유저와 협업을 하는 상황이라면 revert를 사용하여 해당 변화에 대한 히스토리를 남겨야만 합니다.

이제 github와 연동해 github에 코드를 올려보도록 하겠습니다. 일단 가입한 github 계정에 저장소를 만들어야 합니다 왼쪽 상단에 new를 눌려 새로운 repo를 만들면 되겠습니다.

create repository를 하게 되면 다음과 같은 화면이 나오게 될 텐데 기존에 있던 git repo를 적용시킬 것이기 때문에 두 번째 블록의 코드를 유의하여 보시면 되겠습니다.

두 번째 코드를 보며 하나하나 vscode에 터미널에 작성해 보겠습니다 일단 git remote add origin 으로 원격 저장소를 등록합니다. origin이라는 이름으로 원격 저장소를 지정한다는 뜻입니다. 이후 git branch -M main으로 현재 branch의 이름을 main으로 하겠다는 뜻입니다. 이후 git push -u origin main으로 브랜치명 main의 내용을 origin 즉 원격 저장소로 보내겠다는 뜻입니다 화면과 같이 잘 보냈다는 메시지를 확인하시면 잘 보낸 것입니다. 방금 창을 새로고침하면 다 업로드되었음을 확인해 볼 수 있습니다.

이제는 gui로 sourchtree를 사용하여 진행해보도록 하겠습니다. sourcetree를 사용하면 이전에 했던 것을 더욱 쉽게 진행할 수 있습니다. 일단 create a repo를 선택하고 create를 해 git 파일을 만들 폴더를 지정합니다.

로컬 저장소에 북마크로 지정하고 열게 되면 아래와 같은 화면을 확인해 볼 수 있습니다.

add 와 commit 또한 간단하게 클릭해 진행하시면 되겠습니다.

reset과 revert로 간단한데 revert를 하고 싶은 해당 커밋을 선택하고 커밋 되돌리기를 하면 revert를 진행할 수 있고 reset 하고 싶은 해당 커밋을 선택하고 이 커밋까지 현재 브랜치를 초기화를 누르면 reset을 진행할 수 있습니다.

github를 연동하는 방법은 저장소에서 원격 저장소를 추가하고 여러분의 github 주소를 복사해 저장하고 이름을 지정하고 나면 완료됩니다.

이후 좌측에 원격 내에 있는 여러분이 지정한 원격 저장소 이름을 우클릭하면 푸시 버튼이 있는데 해당 버튼을 누르고 push 버튼을 누르면 원격 저장소에 해당 디렉토리와 git 파일이 올라가게 됩니다.

여기까지 로컬에서 git을 사용해 버전 관리를 하고 원격 저장소에 올리는 방법까지 알아보았습니다. 지금 설명한 방법들은 아주 일부분입니다. 아직 파트너 또는 팀 단위로 바로 투입되기에는 충분한 내용이 아닙니다. 다만 git에 대한 구조에 대해 이해하고 이후의 내용들을 배우게 된다면 더 다음의 내용을 더 빠르고 쉽게 이해할 수 있을 것입니다. 여러분들이 앞으로 git에 대해 더 익혀서 앞으로 하는 팀프로젝트에 잘 활용할 수 있기를 바랍니다 감사합니다.