Version Contrl System

파일의 변경사항을 기록하고 싶을때 어떻게 해야할까요?

예를 들어, 중요한 발표를 위해서 ppt 파일을 만든다고 생각을 해보죠 결국은 한가지 결과물이 될 ppt 파일을 위해 우리는 여러번의 검토와 수정을 거쳐야 합니다. 이때, 만약에 수정내용을 되돌리게 될 경우를 생각해서 ppt 파일의 이름을 이렇게

> 발표자료_초안.ppt 발표자료_1차수정.ppt 발표자료_2차수정.ppt ...

이런식으로 계속해서 파일이 만들어 진다면? 만약 이 발표가 너무너무 중요해서 100번의 수정을 거치게 된다면 우리는 무려 100개의 ppt파일을 갖게 될지도 모릅니다.

더군다나 이 발표가 조별과제여서 팀원들끼리 수정 작업을 나눠서 하게 되었다면 수정 과정이 길어질수록 여러가지 혼란이 가중 될 것입니다.

이때 이러한 혼란을 없애기 위해 파일의 이름은 언제나

> 발표자료.ppt

이지만 하나의 파일에 여태까지의 수정사항을 모두 기록 할 수 있도록 해주는것이 바로 **Version Control System** 의 역할인 것입니다.

이러한 **Version Control System** 의 장점으로는 대표적으로 3가지를 들 수 있습니다.

1. Backup: 소스코드를 백업해서 만약의 사태에 대비하게 해줍니다.
2. Recovery: 소스코드를 이전의 상태로 쉽게 되돌릴수 있게 해줍니다.
3. Collaboration: 그룹 내에서의 소스코드에 대한 협업을 용이하게 해줍니다.

이러한 **Version Contrl System** 에는 다양한 소프트웨어들이 존재합니다. 대표적으로는 **CVS, SVN, GIT** 을 예로 들수 있습니다.

그리고 저희가 여기서 다뤄볼 **Version Control System** 은 바로 **GIT** 입니다.

GIT

그렇다면 본격적으로 GIT에 대해서 알아보도록 하겠습니다.

1. 기록의 기준

위에서 설명한 것에 따르면, 이 **Version Control System(이 후 편의상 GIT으로 통일 하겠습니다.)** 이라는 것은 작업물의 수정사항을 기록하게 되어있는데, 그렇다면 이 기록의 기준은 무엇일까요?

이 수정사항을 기록하는 기준은 바로 '의미' 입니다.

하지만 사람이 작성한 작업물의 의미를 소프트웨어가 파악할 수는 없겠죠? 따라서 여기서 기준으로서의 의미는 사람, 즉 GIT의 사용자가 정하는 것으로 합니다.

여기서 우리는 **commit** 이라는 명령어를 사용하게 되는데요, 이 명령어는 사용자가 정한 의미 단위에 따라 이 GIT에 수정사항을 기록할때 사용되어 집니다.

그리고 이때 수정사항에 대한 간략한 설명을 위해 **commit message** 를 함께 기록합니다.

예를 들면, 2018년 07월 10일 09시 35분에 해당 커밋을 수행했다. 라는 식으로 메세지를 함께 기록하면 수정사항에 대한 시간적 정보를 알수 있겠죠?

2. 수정의 범위 소스코드는 분명히 여러개의 파일로 구성이 되어있을 겁니다. 너무너무 간단한 구조의 프로그램이 아니라면 말이죠. 그렇다면 매 **commit** 시 마다 수정되는 파일의 범위가 단일파일의 범위라면 프로젝트가 거대해 졌을 시에는 엄청나게 많은 **commit** 의 수행이 요구되어지겠죠?

따라서 **commit** 의 수정의 범위는 단일파일의 범위가 아닌 다중파일의 범위입니다. 한번의 **commit** 은 여러 파일에 걸친 수정사항을 포함할 수가 있는거죠.

3. branch 이 **branch** 라는 것은 **GIT** 과 다른 **Version Control System** 과의 대표적인 차별점 입니다.

GIT 을 처음 시작할때

```
git init
```

명령어를 통해 현재 위치한 디렉토리에서 **GIT** 을 시작하게 되는데요 이때 **GIT** 은 **commit** 에 따라서 변하는 해당 폴더의 내용을 추적하기 위한 준비를 하게 됩니다.

이때 **init** 명령어를 통해 **GIT 저장소** 가 만들어 지게 되면 **GIT** 은 자동으로 **master** 라는 이름의 최상위 **branch** 를 만들게 되죠.

GIT 은 바로 이 **branch** 단위로 **commit** 에 따라서 수정사항을 기록하게 됩니다.

branch 의 사전적 의미로는 **곁가지, 지사** 등의 의미가 있죠?

GIT 에서의 **branch** 또한 이 사전적 의미와 일맥상통하는 의미를 가집니다.

```
git branch "새로운 브랜치 이름"
```

의 명령어를 사용해서 우리는 현재의 기준 **branch** 에서 갈라져나오는 새로운 **branch** 를 만들 수 있는데요, 이 새로 만들어진 **branch** 는 기준 **branch** 의 상태를 그대로 이어받게 됩니다.

완전히 동일한 상태 **(여기서 상태란 GIT이 관리하는 모든 정보를 포함합니다.)** 를 가진 복사본이 만들어진 것이죠.

만약, **master** 에서 **newlab** 이라는 이름을 가진 새로운 **branch** 를 만든 후 해당 **branch** 에서 **commit** 을 수행하게 된다면 이 **GIT** 에서의 **commit** 은 **branch** 단위로 이루어지기 때문에 **master** 와는 무관하게 새로 생성한 **newlab branch** 에서 독립적으로 이루어지게 됩니다.

GIT 은 매우 강력한 이 **branch** 를 원하는 만큼, 빠르게 생성할 수 있도록 기능을 제공하고 있습니다.

그리고 우리는

```
git checkout "전환하려고 하는 브랜치 이름"
```

를 통해 **GIT** 상에서 현재 작업중인 위치를 가르키는 가상의 커서인 **HEAD** 를 언제든 이동시킬 수 있습니다.

이 **branch** 는 제가 새로운 **branch** 의 이름을 **newlab** 이라고 한 부분에서 짐작 하셨을 수도 있지만 프로그램의 본류에는 영향을 끼치지 않은 상태로 복사본에 대해서 실험적인 작업을 할때 주로 이용되어집니다.

만약 이 실험이 성공적이었다면 우리는 브랜치에 행한 실험정보를 프로그램의 본류에 적용할 필요가 있습니다. 이때 우리는

```
git checkout master
git merge newlab

git branch -d newlab
```

으로 새로운 **branch** 의 내용을 본류에 적용 후 필요성이 사라진 **branch** 를 삭제 할 수 있습니다.

그리고 만약 이 실험이 실패로 끝나게 되었다면, 우리는

```
git branch -D newlab
```

의 명령어를 사용해 본류에 실험내용을 적용하지 않은채로 강제로 **branch** 를 삭제 할 수 있습니다.

4. 협업의 방법

그렇다면 이 **GIT** 의 강력한 기능들을 사용해서 동료들과 함께 어떻게 협업 시스템을 구성할 수 있을까요?

우리는 **리모트 저장소** 를 만들고 네트워크를 통해 해당 **리모트 저장소** 를 동료들과 함께 공유하고, 이를 경유하여 협업 시스템을 구성 할 수 있습니다.

이 **리모트 저장소** 에서 **GIT 저장소** 를 최초로 내려받는 것을

```
git clone "리모트 저장소"
```

위처럼 할 수 있고, 해당 리모트 저장소의 변경사항을 내려받은 clone에 적용 하는 것을

```
git pull
```

을 통해 수행 할 수 있습니다.

그리고 이 **리모트 저장소** 를 쉽게 만들고 이용할 수 있도록 가능하게 해주는 서비스가 바로 어디선가 한번쯤 들어봤던 **Git Hub** 라는 서비스 입니다.

GIT 의 설치

그럼 지금부터는 **GIT** 을 직접 본인의 랩탑 혹은 데스크탑에 설치 해 보도록 할건데요 Winodws 와 Mac 그리고 Linux & Unix 에서의 설치방법을 모두 각각 알아보도록 하겠습니다.

#####1. For Window

1. 웹 브라우저를 통해 https://git-scm.com/ 로 접속합니다.
2. 접속한 사이트의 우측을 보시면 데스크탑 모니터 화면에 **Download 2.18.0 for Windows(2018-07 기준 최신버전)** 이라고 쓰여져 있는 것을 발견 하실 수 있을겁니다.
3. 해당 버튼을 클릭하시면 본인의 운영체제에 알맞는 **Git Installer** 가 자동으로 설치되기 시작합니다.
4. **Installer** 가 설치 완료되면 해당 **Installer** 를 실행하고 특별한 경우가 아니라면 모든 설정값을 **Default** 로 그대로 두고 모두 **Next** 를 누르면 됩니다.
5. 설치가 완료되면 **Finish** 버튼을 누르고 시작메뉴의 검색 창에 **git** 혹은 **git bash** 를 검색하면 나오는 **Git Bash** 를 실행합니다.
6. 실행 후 열린 **Git Bash** 창에서 **git** 을 입력하고 엔터를 쳤을 때 아래와 같은 화면이 나온다오면 설치가 성공적으로 이루어진 것 입니다.

2. For Mac

1. **terminal** 을 실행합니다.
2. **git --version** 을 입력합니다.
3. 만일 git의 버전에 관한 정보가 터미널에 출력된다면 이미 git이 설치되어 있는것입니다. 그렇지 않고 터미널 상에

```
xcode-select: note: no developer tools were found ...
```

처럼 표시되면서 명령어 라인 개발자 도구가 필요하다는 창이 뜬다면 설치를 누릅니다. 4. 설치가 완료된 뒤에 다시 터미널 상에서 **git --version** 을 입력합니다. 5. git version ~~~ 이 출력된다면 정상적으로 설치 된 것입니다. 6. 만약에 위의 방법이 모두 안될 경우에는 https://git-scm.com/ 에 접속해 사이트 우측에 모니터에 있는 Download For Mac 버튼을 클릭해서 pkg 파일을 다운 받고 실행 해 설치합니다. 7. 그 뒤에 터미널에서 다시 **git --version** 을 입력합니다. 8. 버전 정보가 출력되면 성공적으로 설치된 것 입니다.

#####3. For Linux & Unix

1. 기본적으로 CLI 모드를 사용한다는 가정하에 진행합니다.
2. git이 설치가 되있는 경우가 많기 때문에 먼저 git 명령어를 입력합니다.
3. git: command not found 와 같이 출력될 경우에는 apt-get install git 혹은 yum install git 을 입력합니다. (우분투의 경우에는 apt, Cent OS의 경우에는 yum 입니다.)
4. git --version 을 입력해서 버전정보가 출력되면 성공적으로 설치가 된 것입니다.

GIT 맛보기

프로젝트 루트로 설정하고 싶은 원하는 디렉토리로 이동한 뒤,

```
git init
```

명령어를 입력합니다.

이 명령어는 프로젝트 저장소를 만드는 명령어 입니다. 이제 해당 디렉토리는 프로젝트 루트로 설정되었으며, 숨김 디렉토리 .git이 생성되었습니다.

```
ls -a 혹은
ls -al
```

명령어로 .git 디렉토리가 생성된 것을 확인 할 수 있습니다.

GIT을 사용하다 보면 분명히 해당 프로젝트 내에서도 GIT에서 추적하지 않았으면 하는 파일이 분명 생길 것 입니다. 이를 위해서 .gitignore 파일을 생성합니다. 예를 들면,

```
vi .gitignore

------vi 에디터---------

 #temporary files
 *.tmp

 #useless files
 *.bak
------------------------
```

이를 통해 GIT은 bak과 tmp 확장자를 가진 파일을 추적하지 않습니다.

```
git status
```

이 명령어는 GIT 저장소의 현재 상태를 출력합니다. GIT을 사용하다 보면 아주 많이 사용하게 될 명령어입니다.

다음으로는 GIT을 사용하기전에 최초 설정을 해주도록 하겠습니다.

```
git config --global
```

해당 옵션으로 사용자에게만 적용되는 설정을 확인하고 변경 할 수 있습니다.

```
git config --global user.name "username"
git config --global user.email useremail@exmaple.com
```

위와 같이 git config --global 옵션을 통해 사용자의 이름과 이메일을 설정한다. 이 설정은 한번만 해주면 됩니다. 만약에 매 프로젝트마다 다른 이름과 이메일을 사용하고자 한다면 global 옵션을 제외해주면 됩니다.

만일 Windows 이용자라면 이하 설정을 추가적으로 해줍니다.

```
git config core.eol lf
git config core.autocrlf input
```

이는 윈도우는 개행문자로 CRLF를 사용 맥과 리눅스는 LF만 사용하는 것 때문에 추가적으로 해주는 설정입니다. 이 설정을 이용하면 윈도에서는 CRLF를 사용하고 Mac, Linux, 저장소에서는 LF를 사용할 수 있습니다.

이 후에 방금 만들어준 .gitignore 파일을 한번 commit 해보도록 합니다.

```
git add .gitignore //다음 커밋에서 .gitignore 파일을 커밋 하게끔 해당 파일을 대기 시킨다.
git commit -m "Initial commit: added .gitignore."// -m 옵션으로 commit message를 추가 해 줄수있다.
```

여기서 한번 commit한 파일인 .gitignore 파일을 수정 한 뒤에 git status를 명령어를 통해 저장소의 상태를 확인해 보도록 하겠습니다.

```
vi .gitignore

---vi 에디터 ---
 #temporary files
 *.tmp
 *.log // 확장자가 .log인 파일을 모두 추적하지 않도록 합니다.
 #useless files
 *.bak
----------------

git stauts // .gitignore 파일을 수정, 저장 한뒤에 git status 명령을 실행합니다.
```

이렇게 .gitignore 파일을 수정, 저장 한뒤에 git status 명령어를 실행 했을 경우에 터미널에는 이하의 내용이 출력된다.

```
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:   .gitignore

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

내용을 요약하자면 master branch 에서 .gitignore 파일이 modified 되었는데 그 Changes 가 commit 되기 위한 준비 상태에 들어가지 않았다는 의미입니다.

따라서

```
git add .gitignore 혹은
git add -A // 모든 변경사항을 commit 하기 위한 준비상태로 만듬
```

을 입력한 뒤 git status 명령을 실행하면 아래와 같은 출력을 볼 수 있습니다.

```
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   .gitignore
```

그리고 다시 commit을 실행 합니다.

```
git commit –m ".gitignore is modified slightly"
```

그 후 git status 명령어를 다시 실행하면 최종적으로 아래와 같은 출력을 볼 수 있습니다.

```
$ git status
On branch master
nothing to commit, working tree clean
```

commit 할 것이 없으며 작업트리가 깨끗하다는 의미입니다.

여기까지 GIT 입문자를 위한 가이드북이었습니다.