오픈소스SW개론 기말고사 정리본

리눅스

리눅스: 리누스 토르발스(1991년에 개발), 대표적 오픈소스, 안정성높음

CLI: GUI에서 하지 못하는 세세한 작업 가능, 작업속도 빠름

- 리눅스 설치 유형
  - 노트북에서 기존 OS를 제거하고 리눅스 설치(빠른 속도, 기존 OS 삭제)
  - VM를 사용하여 가상환경으로 리눅스 설치(기존 OS 유지 GUI 가능, 느린 속도)
  - 듀얼부팅 사용하여 기존 OS에 추가로 리눅스 OS를 설치(기존 OS 유지 GUI 가능, OS전환 시 재부팅)
  - Docker를 설치하여 가상환경으로 리눅스 설치(기존 OS 유지 빠른 속도, GUI 불가능)

바이오스: 윈도우 등 운영체제 설치 시 부팅장치를 전환하는 용도

Legacy

- 바이오스는 약 20년전 운영체제 가동 위해 설계된 오래된 유틸리티
- 16비트 시스템이며 최대 1MB 메모리 사용 가능

UEFI

- 오래된 바이오스 펌웨어는 최신 PC와 점점 호환이 어려워짐
- 따라서 Legacy 바이오스를 대신하여 2010년쯤 UEFI 모드로 전환이 시작됨
- UEFI 모드는 최신 PC용 펌웨어 인터페이스
- 온도와 전력 모니터링, 원격 보안 관리, 가상화 등 지원
- Lagacy와 다르게 GUI이므로 쉽게 조작 가능함
- Docker
  - 프로그램을 감싸는 일종의 컨테이너
  - 가상화 기술을 -> 프로그램을 독립적으로 실행
  - 컴퓨터 환경에 따라 달라지는 프로그램
  - 실행 문제를 해결, 프로그램 실행 환경을 일관성 있게 유지하면서 여러 대의 컴퓨터에서 실행
  - 호스트 운영체제를 공유하여 필요한 최소한의 리소스만 할당받아 동작하므로 가벼움
  - 가상머신보다 훨씬 빠른 실행속도
  - 개발 언어에 종속되지 않으며 보안성이 좋음

컨테이너: 여러 가지 프로그램을 담아 실행할 수 있는 환경을 제공함

이미지란 도커 컨테이너를 만들기 위한 지시사항들을 정리해놓은 것

- 패키지

- 리눅스 패키지란, 리눅스 시스템에서 소프트웨어를 실행하는데 필요한 파일들이 담겨 있는 설치 파일 묶음
- 패키지는 소스 패키지와 바이너리 패키지로 구분
- 소스패키지
  - 말 그대로 소스 코드가 들어 있는 패키지로 컴파일 과정을 통해 바이너라 파일로 만들어야 실행 가능
- 바이너리 패키지
  - 성공적으로 컴파일된 바이너리 파일이 들어있는 패키지
  - 이미 컴파일이 되어 있으니 소스 패키지에 비해 설치 시간도 짧고 오류가 발생할 가능성도 적음
  - 따라서 리눅스의 기본 설치패키지들이 대부분 바이너리 패키지
- 패키지 형식
  - 리눅스 배포판에 따라서 서로 다른 패키지 형식을 지원한다
  - Debian 계열(Debian, Ubuntu): .deb
  - Red Hat 계열(RedHat, Fedora, CentOs): .rpm
  - OpenSUSE 계열: .rpm
- Debian
  - 메이저 GNU/리눅스 배포판 중 거의 비자유 소프트웨어를 적극적으로 배제하는 배포판
  - 자유 소프트웨어가 지침
  - 크롬, 우분투, 라즈베리, Neptune
- 패키지 관리 툴

구분	저수준 툴	고수준 툴
데비안계열	dpkg	apt-get / apt
레드헷계열	rpm	yum
openSUSE	rpm	zypper

-
  - 저수준 툴
    - 실제 패키지의 설치, 업데이트, 삭제 등을 수행
  - 고수준 툴
    - 의존성의 해결, 패키지 검색 등의 기능을 제공
  - apt
    - apt는 advanced packaging tool의 약자로
    - apt-get은 데비안 계열의 리눅스에서 쓰이는 패키지 관리 명령어 도구
    - 좀 더 가독성이 좋고 메세지 안내가 나옴
    - 더 새롭고 사용자 친화적으로 설계 더 많은 정보를 제공
  - apt-get
    - 좀 더 세부적인 옵션과 스크립트 작성을 할때 유리함
    - 더 오래되고 최소한의 디자인을 가지며 스크립팅에 더 안정적
  - apt vs dpkg
    - dpkg는 window환경에서는 exe 파일과 비슷한 설치파일이며, deb 확장명으로 불린다.

        - 즉, A패키지가 설치되기 위해서 B패키지가 필요한경우, 해결이 까다로움
        - 이를 apt-get을 사용
        - apt-get을 사용시 인터넷에 있는 저장소에서 파일을 다운로드해서 설치
        - 이 방식은 dpkg와 달리 종속된 프로그램이 만약 작업환경에 미설치되어있다면 추가 수동설치 필요없이 자동으로 설치
        - 내부적인 동작 차이는 거의 없음
    - 패키지 설치
        - apt-get install 패키지명
    - 패키지 삭제
        - apt-get remove 패키지명(apt-get autoremove 사용하지 않는 패키지 제거)
    - 패키지 목록 갱신
        - apt-get update
    - 패키지 업그레이드
        - apt-get upgrade
    - 패키지 검색
        - dpkg -l | grep 패키지명: 이름 검색
        - apt-cache show 패키지명: 정보 보기
        - apt-cache depends 패키지명: 의존성 확인
        - apt-cache rdepends 패키지명: 역의존성 확인

- 압축

    - tar: 여러 개의 파일을 하나의 파일로 묶거나, 하나의 파일을 여러 개의 파일로 풀 때 사용하는 명령어
        - tar -cvf: 파일을 묶을 때 사용
        - tar -xvf: 파일을 풀 때 사용
    - zip
        - zip -r: 파일을 묶을 때 사용
        - unzip: 파일을 풀 때 사용

- $PATH

    - 리눅스에서 $PATH란 실행파일들의 디렉토리 위치를 저장해 놓은 변수
    - 현재 위치와 상관없이 특정 명령어를 입력하면 PATH변수에 저장되어 있는 경로로 명령어 실행
    - .bashrc파일에 추가하여 경로 추가

- gcc + g++

    - GCC는 GNU 프로젝트에서 다양한 프로그래밍 언어용으로 제작한 컴파일러 세트
    - gcc: C컴파일
    - g++: c++ 컴파일
    - 본질적으로 같은 컴파일러, 코드를 다르게 취급하고 다른 라이브러리에 연결
    - gcc는 수동으로 인수를 포함하여 라이브러리를 호출하지만 g++은 자동
    - gcc는 .c, .h파일만 취급하지만 g++은 .cc, .cpp, .c등을 취급
    - g++은 gcc가 정의하지 않은 일부 매크로를 정의
    - g++는 이름 맹글링, 예외 및 C++ 고유의 기타 기능을 허용
    - 실제로 C++ 프로그램을 컴파일하는 경우 잠재적인 문제를 피하기 위해 g++를 사용해야 한다.

- 컴파일

- .c과 같은 인간이 이해하는 소스코드를 CPU가 이해하는 이진수로 번역하는 과정
- 어셈블리 코드와 1 대 1로 대응
  - g++ -c test.cpp : 오브젝트 파일 생성
- 컴파일 명령어
  - -L
    - 라이브러리를 찾을 디렉토리를 지정
  - -I
    - 헤터파일 탐색하는 기본 디렉토리를 추가할 때 사용
  - -l<라이브러리명>
    - 실제 사용할 라이브러리를 씀
    - 이 때, 그 라이브러리에 있는 함수를 사용하려면 해당 함수가 정의되어있는 헤더파일도 포함시켜야 한다.
  - ex) gcc -W -Wall -O2 -o like love.c

리눅스 CLI 명령어

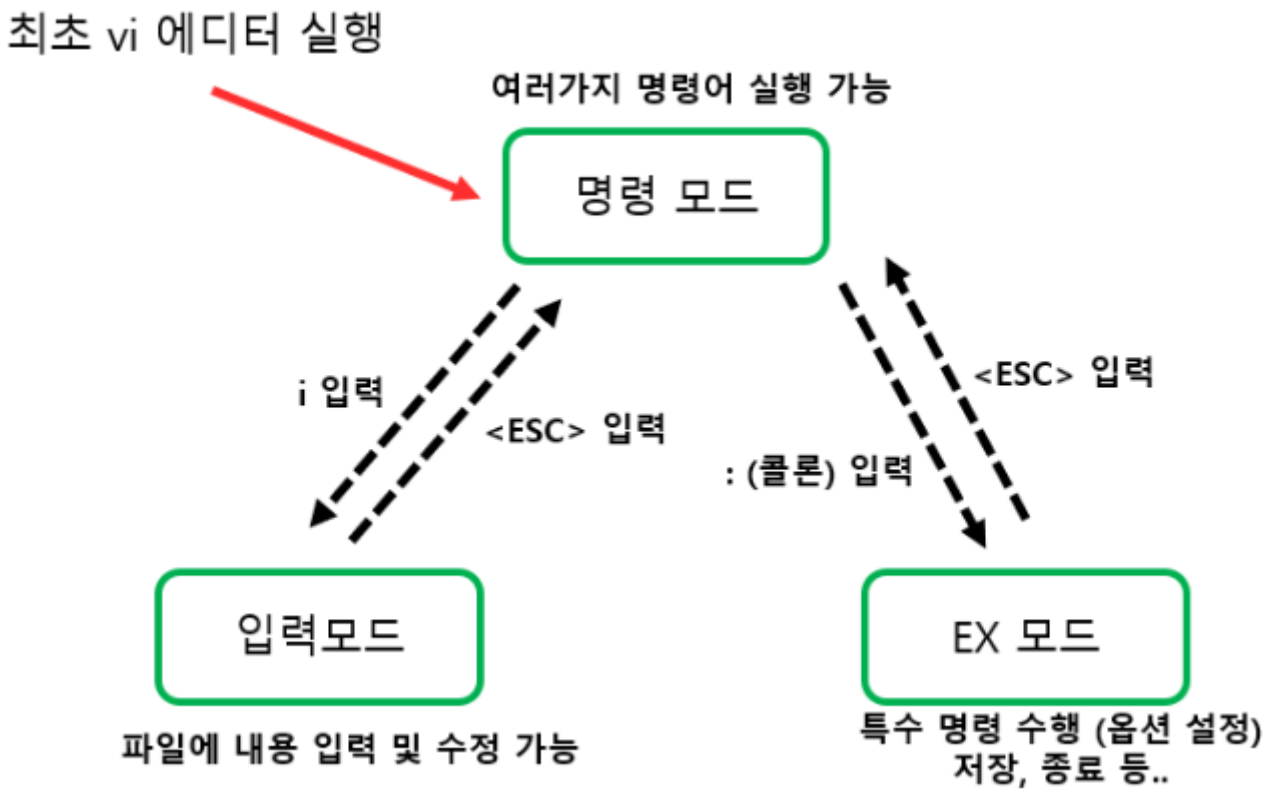
VI: 리눅스에서 사용되는 텍스트 편집기, visual editor의 약자, 터미널 환경에서 사용

Vim

- Vi improved의 약자
- vi가 개선된 버전
- 사용자 인터페이스, 강력한 문법 강조 기능, 다중 창 및 다중 버퍼 기능 등을 제공
- 다양한 플러그인과 확장 기능을 제공

VI 모드

- 명령 모드: vi 편집기를 실행하면 처음으로 진입하는 모드, vi 편집기의 명령을 실행하는 모드
- 입력 모드: 명령 모드에서 i, a, o 등의 키를 누르면 입력 모드로 진입, 텍스트를 입력하는 모드



i	현재 커서의 위치부터 입력	I	현재 커서 줄의 맨 앞에서부터 입력
a	현재 커서의 위치 다음 칸부터 입력	A	현재 커서 줄의 맨 마지막부터 입력
o	현재 커서의 다음 줄에 입력	O	현재 커서의 이전 줄에 입력
s	현재 커서 위치의 한 글자를 지우고 입력	S	현재 커서의 한 줄을 지우고 입력

방향키

- h : 왼쪽
- j : 아래
- k : 위
- l : 오른쪽

단어 단위 이동

- w : 다음 단어의 첫 글자로 이동
- b : 이전 단어의 첫 글자로 이동

행의 첫/마지막

- 같은 행에서 제일 처음 글자 및 마지막 글자로 이동 가능
- ^ : 행의 첫 글자로 이동
- $: 행의 마지막 글자로 이동

이전/다음 행 이동

- + : 다음 행의 첫 글자로 이동
- - : 이전 행의 첫 글자로 이동

문서의 시작/끝

- gg : 문서의 첫 행으로 이동
- G : 문서의 마지막 행으로 이동

임의 행으로 이동

- :행번호 : 해당 행으로 이동
- {,} : 이전 문단, 다음 문단

화면 이동

- Ctrl + f : 다음 페이지로 이동
- Ctrl + b : 이전 페이지로 이동

기본 규칙

- 명령 앞에 숫자를 넣으면 그 명령 앞에 누른 숫자만큼 반복한다
- ex) + 명령어가 다음 행으로 이동하는 것이라면 3+ 누르면 3행 아래로 이동

삭제

- x: 커서 위치의 글자 삭제
- nx: n개의 글자 삭제
- dw: 한 단어를 삭제
- ndw: n개의 단어를 삭제
- dd: 한 행을 삭제
- ndd: n개의 행을 삭제
- D: 커서 위치부터 행의 끝까지 삭제

복사

- :%y: 문서 전체를 복사
- :a,by: a행부터 b행까지 복사
- yw: 현재 커서 위치의 단어를 복사
- nyw: 현재 커서 위치부터 n개의 단어를 복사
- yy: 현재 커서 위치의 행을 복사
- nyy: 현재 커서 위치부터 n개의 행을 복사

붙여넣기

- P: 현재 행 이후에 붙여넣기
- p: 현재 행 이전에 붙여넣기

블록지정

- 원하는 위치에서 v를 누르고 커서를 이동하면 블록이 형성
- 블록을 지우려면 d, 복사는 y

되돌리기

- u: 직전 명령을 취소

저장

- w: 저장
- :w file.txt: 이름으로 파일 저장

종료

- :q: 종료
- :q!: 저장하지 않고 종료
- :wq!: 강제 저장 후 종료

탐색

- /문자열: 문서에서 문자열을 찾음(아래 방향)
- ?문자열: 문서에서 문자열을 찾음(윗 방향)
- n: 다음 문자열로 이동
- N: 이전 문자열로 이동

커널: 운영체제의 핵심, 하드웨어와 소프트웨어의 연결(상호작용), 하드웨어 관리, 프로세스 관리, 메모리 관리, 파일 시스템 관리, 네트워크 관리 등등

Shell: 커널과 사용자 사이를 이어주는 명령어 처리기

Bourne Shell: sh C Shell: csh tee-see-shell: tcsh Z Shell: zsh Bourne Again Shell: bash (가장 많이 쓰이고 기본으로 채택)

터미널: 터미널은 컴퓨터에서 명령어를 입력하고 실행하는 인터페이스, GUI x, 리눅스는 shell을 사용

shell vs 터미널

- shell
  - 레스토랑 쉐프
  - 명령줄 인터프리터
  - 터미널에 입력하면 shell이 대신 해당 명령어 읽고 해석
- 터미널
  - 레스토랑 웨이터
  - 텍스트 명령을 입력하여 리눅스 운영체제와 상호작용하는 인터페이스
  - shell에게 전달
  - 그 후 shell이 명령어를 결과, 출력을 터미널에게 전달

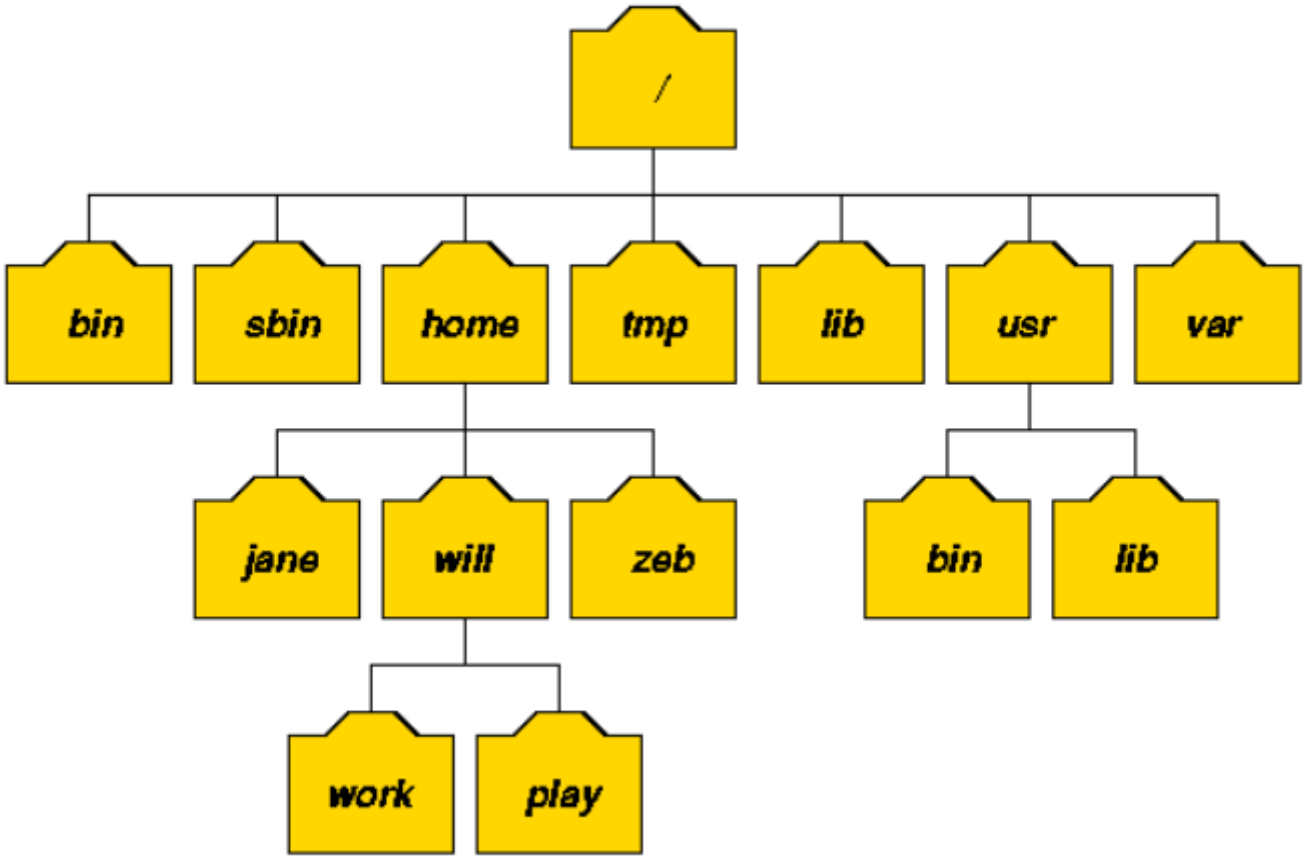
디렉토리 이동 cd

- /
  - 최상위 디렉토리 혹은 루트
- ~
  - 홈 디렉토리
  - 사용자 계정 디렉
- ..
  - 상위 디렉
- .
  - 현재 디렉

- 절대 경로
  - 절대 경로는 /로 시작
- 상대 경로
  - 현재 위치를 기준으로 이동
- pwd
  - 현재 디렉토리 위치를 표시
- echo
  - 문자열 표시
- cat
  - 파일 내용 표시
- history
  - 명령어 히스토리 표시
- mkdir
  - 디렉토리 생성
- rm -r(f)
  - 디렉토리 삭제
- cp -r
  - 디렉토리 복사
  - 파일 복사 cp [file1] [file2]
- mv
  - 디렉토리 이동(이름 변경)
- ls
  - list 현재 파일/디렉토리
  - -a: 숨김 파일 표시
  - -al: 숨김 파일 표시 및 상세 정보 표시
- wget
  - 단일 파일을 다운

디렉토리 구조

- 리눅스의 디렉토리, 파일 시스템은 윈도우와 다름
- 리눅스는 /, 윈도우는 \
- 리눅스는 디렉토리, 윈도우는 폴더



리눅스 디렉토리는 역 트리 구조

명령어의 종류와 성격, 권한에 따라 분리됨

- /
  - 최상위 디렉토리 루트를 의미
  - 모든 디렉토리 시작점
  - 절대 경로표시
- /bin
  - 기본적인 명령어들이 위치
  - cd, ls, mkdir, rm, cp, mv, cat, echo, history, wget
- /boot
  - 리눅스 부트로더가 존재
- /dev
  - 시스템 디바이스 파일을 저장
- /etc
  - 시스템 설정 파일들이 저장
- /home
  - 사용자 계정의 홈 디렉토리
- /lib
  - 커널모듈파일과 라이브러리 파일 즉, 커널이 필요로 하는 파일과 프로그램에 필요한 라이브러리 파일 존재
- /root

    - 시스템 최고 관리자인 root 개인 홈 디렉
- /tmp
    - 공용 디렉토리
- /proc
    - 가상파일시스템
    - 현재 메모리에 존재하는 모든 작업들이 파일형태로 존재(메모리 상에 존재하기 때문에 가상 파일 시스템)
    - 여기에 존재하는 파일들 가운데 실행중인 커널의 옵션 값을 즉시 변경할 수 있는 파라미터 파일들이 있기 때문에 시스템 운용에 중요한 역할
- /usr
    - 시스템이 아닌 일반 사용자들이 주로 사용하는 디렉토리
    - c++, cpp등과 같이 일반 사용자 명령어들 위치

CLI 환경에서의 버전관리

- 로컬 저장소
    - .git에서 생성한 버전정보 주소등이 저장되어 있음
    - 커밋, 커밋을 구성하는 객체
    - 워킹트리
        - 로컬저장소에 있는 현재 디렉토리를 의미
    - 삭제
        - .git 디렉토리 삭제
- config
    - git을 사용하기 위해 옵션을 설정하는 곳
    - 전역, 지역, 시스템 환경이 있으며 우선순위는 지역 > 전역 > 시스템
    - 지역: 해당 저장소에서만 사용
    - 전역: 현재 사용자 계정에서만 사용
    - 시스템: 시스템 전체에서 사용
- .gitignore
    - 프로젝트에서 원하지 않는 백업 파일이나 로그파일, 컴파일된 파일 등을 git 버전관리에서 제외시키도록 하는 설정 파일
    - # : 주석
    - file : 특정 파일
    - /file : 현재 디렉토리의 특정 파일
    - dir/file : 특정 경로 특정 파일
    - *.file : 확장자가 file인 모든 파일
    - dir/** : 특정 디렉토리 하위의 모든 디렉토리
    - !file : 특정 파일 제외

```
// 사용자 등록
git config --global user.name "name"
git config --global user.email "email"
```

```
// 기본 에디터 확인
git config --global core.editor

// 스테이징 및 커밋
git add [file]
git commit -m "message"

// 언스테이징
git reset [file]

// 커밋 취소
git reset --hard HEAD^

// 커밋 이력
git log
git log --oneline
git log --oneline --graph --all

// Push
git push origin master
// if (원격 저장소가 등록되어 있지 않다면)
git remote add origin [url]

//매번 원격저장소의 브랜치명을 적기 싫다면
git push -u origin master (업스트림 등록)

// Pull (git fetch + git merge 합성)
gir pull

// 클론
git clone [url] [dir]
```

- 브랜치
  - 특정 기준에서 줄기를 나누어 작업할 수 있따.
  - 커밋을 가리키는 포인터
  - 새로운 기능을 추가할 때:Feature
  - 버그 수정:Bug
  - 병합과 리베이스 테스트
  - 이전 코드 개선
  - 특정 커밋으로 돌아가고 싶을 때

```
// 브랜치 목록 확인
git branch
git branch -v

// 브랜치 생성
git branch [branch name]

// 브랜치 생성 이동
git checkout -b [branch name]
```

```
// 브랜치 삭제
git branch -d [branch name]
git branch -D [branch name] // 강제 삭제

// 원격 브랜치 삭제
git push origin --delete [branch name]

// 브랜치 병합 (빨리 감기 병합)
git merge [branch name]

// 태크 생성
git tag [tag name]

// 태그 조회
git tag

// 태그 푸시
git push origin [tag name]

// 태그 삭제
git tag -d [tag name]

// rebase
git rebase [branch name]

// amend
git commit --amend -m "message"

// cherry-pick
git cherry-pick [commit hash]

// reset (기본, hard)
git reset [commit hash]
git reset --hard [commit hash]

// revert
git revert [commit hash]

// stash
git stash save "message"
git stash list
git stash apply [stash name]
```

Git 브랜치 전략

Git 브랜치 전략

- 여러 개발자가 하나의 저장소를 사용하는 환경에서 저장소를 효과적으로 활용하기 위한 work-flow
    - 브랜치의 생성, 삭제, 병합등 git의 유연한 구조를 활용해서 각 개발자들의 혼란을 최대한 줄이며 다양한 방식으로 소스를 관리하는 역할

    - 브랜치 생성에 규칙을 만들어 협업을 유연하게 함

- Git 브랜치 전략이 없으면..?
    - 다음과 같은 것이 헷갈릴 수 있다.
        - 어떤 브랜치가 최신 브랜치인지
        - 어떤 브랜치를 끌어와서 개발을 시작해야하는가?
        - 어디에 push를 보내야 하는가
        - 핫픽스를 해야하는데 어떤 브랜치를 기준으로 수정할까?
        - 배포 버전을 어떤 걸 골라야하나?

- 가지 종류
    - 기본적으로 5가지로 구분
    - feature > develop > release > hotfix > master
    - master
        - 라이브 서버에 제품으로 출시되는 브랜치
    - develop
        - 다음 출시 버전을 대비하여 개발하는 브랜치
    - feature
        - 추가 기능 개발 브랜치. develop 브랜치에서 분기
    - release
        - 이번 출시 버전을 준비하는 브랜치. develop 브랜치에서 분기 QA를 통해 테스트 후 master 브랜치에 병합
        - 뻗어나갔던 가지가 다시 합쳐지는 곳: develop, master
        - devlop 브랜치에 버전에 포함된 기능이 merge되었다면 QA를 위해 devlop 브랜치에서 release 브랜치를 생성
        - 배포 가능한 상태가 되면 master 브랜치로 병합시키고, 출시된 master 브랜치에 버전 태그를 추가함
        - release 브랜치에서 기능을 점검하며 발견한 버그 수정 사항은 devlop 브랜치에도 적용해야함 배포 완료 후 develop 브랜치에 대해서도 merge 작업을 수행
    - hotfix
        - 출시 버전에서 발생한 버그를 수정 하는 브랜치. master 브랜치에서 분기
        - develop, master로 병합됨
        - 제품에서 버그가 발생했을 경우에는 처리를 위해 이 가지로 해당 정보를 모아준다.
        - 버그 수정이 완료된 후에는 develop, master에 곧장 반영해주며 tag를 통해 정보를 기록
        - 버그를 잡는 사람이 일하는 동안에도 develop브랜치에서 일할 수 있다.
        - 이 때, 만든 hotfix 브랜치에서의 변경사항은 develop 브랜치에도 merge하여 문제부분을 처리해야함
        - release 가지가 생성되어 관리된다면 해당 가지에 hotfix 정보를 병합시켜 다음번 배포시 반영이 정상적으로 이루어지도록 함
    - 메인 브랜치(master, develop) 항시 유지
        - master 배포 가능한 상태만 관리하는 브랜치
        - develop 다음에 개발될 기능을 개발하는 브랜치, 이 브랜치를 기반으로 개발
    - 보조 브랜치(feature, release, hotfix)는 필요할 때마다 생성하고 삭제
        - feature브랜치 또는 topic 브랜치를 말한다.
        - 가지가 뻗어나오는 것 devlop
        - 뻗어나갔던 가지가 다시 합쳐지는 곳 devlop

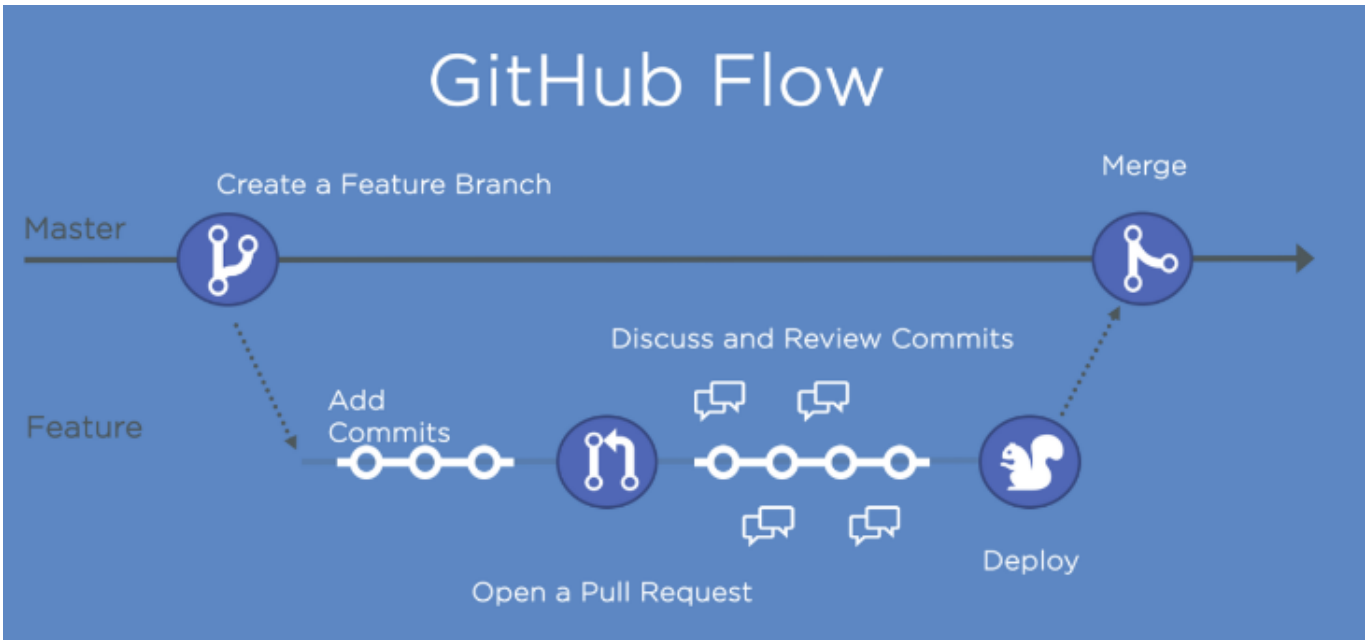
        - 이름 설정: master, develop, release-, *hotfix-*,
        - 새로운 기능을 추가할 때 주로 사용

- Git Flow 전략 흐름

    - master와 develop 브랜치가 가장 많이 사용
    - feature, release, hotfix 브랜치는 필요할 때마다 생성하고 삭제
    - 대부분의 작업은 develop에서 취합한다고 생각하면 되며 테스트를 통해 정말 확실하게 더 이상 변동사항이 없다 싶을 때 master로의 병합을 진행해야 함
    - master가 아닌 가지들은 master의 변동사항을 꾸준히 주시
    - 1. 신규 기능 개발
        - 개발자는 develop 브랜치로부터 본인이 신규 개발할 기능을 위한 feature브랜치를 생성
        - feature 브랜치에서 기능을 완성하면 devlop 브랜치에 merge를 진행
    - 2. 라이브 서버로 배포
        - feature 브랜치들이 모두 develop 브랜치에 merge되면 QA를 위해 release 브랜치를 생성
        - release 브랜치를 통해 오류가 확인된다면 release 브랜치 내에서 수정을 진행
        - QA와 테스트가 모두 통과했다면, 배포를 위해 release브랜치를 master쪽으로 merge함
        - 만일 release 브랜치 내부에서 오류 수정이 진행되었을 경우 동기화를 위해 develop 브랜치 쪽에도 merge를 진행
    - 3. 배포 후 관리
        - 만일 배포된 라이브 서버에서 버그가 발생한다면 hotfix 브랜치를 생성하여 bug fix를 진행함
        - 그리고 종료된 bug fix를 master와 develop 브랜치에 merge함

Github Flow 전략

- github flow
    - git flow가 좋은 방식이지만 github에 적용하기에는 복잡하다는 단점 때문에 만들어진 새로운 git 관리 방식
    - 자동화 개념이 들어있다는 큰 특징이 존재하며 자동화가 적용되지 않은 곳에서만 수동으로 진행하면 됨
    - git flow에 비해 흐름, 규칙이 단순
    - 기본적으로 master 브랜치에 대한 규칙만 정확하게 정립되어있으면 나머지 가지들에 대해서는 특별한 관여를 하지 않으며 PR기능을 사용하도록 권장
    - 특징
        - release branch가 명확하게 구분되지 않은 시스템에서의 사용이 유용
        - github자체의 서비스 특성상 배포의 개념이 없는 시스템으로 되어있기 때문에 이 flow가 유용
        - 웹 서비스들에 배포의개념이 없어지고 있는 추세 따라서 유용
        - hotfix와 가장 작은 기능들을 구분하지 않음



- Github Flow 전략 흐름
    - 1. 브랜치 생성
        - Github-flow전략은 기능 개발, 버그 픽스 등 어떤 이유로든 새로운 브랜치를 생성하는 것으로 시작
        - 단, 이 때, 체계적인 분류 없이 브랜치 하나에 의존하게 되기 때문에 이름을 통해 의도를 명확하게 드러내는 것이 매우 중요하다.
        - master 브랜치는 하상 최신 상태이며, stable 상태로 produc 배포되는 브랜치 이 브랜치에 대해서는 엄격한 role과 함께 사용
        - 새로운 브랜치는 항상 master에서 생성
        - git-flow와 다르게 feature 브랜치나 develop 브랜치가 존재하지 않음
        - 그렇지만, 새로운 기능을 추가하거나 버그를 해결하기 위한 브랜치 이름은 자세하게 어떤 일을 하고 있는지에 대해서 작성함
    - 2. 개발 & 커밋 & 푸쉬
        - 개발을 진행하면서 커밋을 남김
        - 이 때도 브랜치와 같이 커밋 메세지에 의존해야 하기에 자세하게 적어주는 것이 중요(컨벤션)
        - 원격 브랜치로 수시로 push함
        - git flow와 상반되는 방식
        - 항상 원격지에 자신이 하고 있는 일들을 올려 다른 사람들도 확인할 수 있도록 해야함
        - 이는 하드웨어 문제가 발생해 작업하던 부분이 없어지라도, 원격지에 있는 소스를 받아서 작업할 수 있도록 해준다.
    - 3. PR 생성
        - 피드백이나 도움이 필요할 때, merge 준비가 되었을 때는 PR을 생성
        - 이것을 이용해 자신의 코드를 공유하고, 리뷰 받음
        - merge 준비가 완료되었다면 master 브랜치로 반영을 요구함
    - 4. 리뷰 & 토의
        - PR가 master 브랜치 쪽에 합쳐진다면 곧장 라이브 서버에 뱊포되는 것과 다름 없으므로, 상세한 리뷰와 토의가 이루어져야 함
    - 5. 테스트
        - 리뷰와 토의가 끝났으면 해당 내용을 라이브 서버에 배포

        - 배포시 문제가 발생한다면 곧장 master 브랜치의 내용을 다시 배포하여 초기화
    - 6. 최종 Merge
        - 라이브 서버에 배포했음에도 문제가 발견되지 않는다면 그대로 master 브랜치에 푸시를 하고, 즉시 배포를 진행
        - 대부분의 github-flow에선 master 브랜치를 최신 브랜치라고 가정하기 때문에 배포 자동화 도구를 이용해 merge 시킴
        - master로 merge되고 push되었을 때는, 즉시 배포되어야 한다. 이는 Git-flow의 핵심으로 master 브랜치로 merge가 일어나면 자동으로 배포가 되도록 설정

- git flow vs github flow
    - 1개월 이상의 긴 호흡으로 개발하여 주기적으로 배포, QA 및 테스트, hotfix등 수행할 여력이 있는 팀은 git flow가 적합
    - 수시로 릴리즈 되어야 할 필요가 있는 서비스를 지속적으로 테스트하고 배포하는 팀이라면 github-flow와 같은 간단한 work-flow가 적합

github

최근 GitHub에서는 ID/PW 기반의 basic authentication 인증을 금지하고, ID/Personal Access Token 방식의 Token Authentication 인증을 요구함

따라서 소스코드를 push/clone 시 오류가 뜨면서 작동하지 않는 상황 발생 가능 (ex. Linux)