GitScm cmd인터페이스를 사용하여 작업

Lecture1

하드웨어 -> 운영체제 -> 응용소프트웨어 -> 사용자

소프트웨어의 종류

1. 시스템 소프트웨어

사용자와 연결된 것이 아닌 디바이스와 엮여있다.

OS나 디바이스드라이버

2. 응용소프트웨어 (오피스, 게임, 미디어플레이어)

목적에 따른 분류

계층에 따른 분류

사용자 작성 소프트웨어(매크로)

응용 소프트웨어

플랫폼 소프트웨어

여태까지는 소프트웨어 – 소프트웨어 공학으로 넘어가겠다.

소프트웨어 과학이 아닌 공학이 붙은 이유는 무엇일까?

소프트웨어개발을 공학적 접근으로 냉정하게 해결해야 되게 되었다. 머릿속으로만 상상하는 것 이상으로.

폰 노이만(프로그램 내장방식으로 배선조작을 저장해서 자동으로 바뀌도록 만듬)

저장된 프로그램을 읽는 기계를 만들자

연구부와 사업부와의 관계에서 사업부는 양산을 하기 때문에 하루 코드의 사용이 천만번 이상이라고 가정. 연구부에서는 공학적인 접근을 못했다.

문제는 개발 프로세스였다.

소프트웨어 개발에 있어 체계적이며 공학적인 프로세스가 확립되어야 여러 문제점이 냉정하게 해결 될 수 있음

게임소프트웨어 공학은 다른 소프트웨어공학보다 유지보수단계에서의 리액트가 빨랐으나 요즘 다른 소프트웨어도 그런 추세이다.

대표적인 개발모델 – 폭포수모델

순차적 모델, 고전적 생명주기, 이전단계의 완벽한 작업완료를 가정

요구

설계

구현

검증

유지보수

Ex)애플부터 시작된 핀치줌(아직까지도 소송이 되고있는) 을 바다 os에서 하고자 했을 때

실제 개발을 하고나니 요구사항이 달랐다.

차례차례 밀리니 총 2주간 연기가 되었다.

Lecture 2

Version control System – 변화를 시간에 따라 기록한 것 (하나의 time stamp)

각 파일을 특정 시점으로 되돌릴 수 있음

프로젝트 자체를 특정 시점으로 되돌릴 수 있음

시간에 따른 수정 내용 파악 가능

문제를 만들어 낸 원인 파악 가능

3가지의 분류

1. 로컬 버전 관리 시스템 – 간단한 데이터베이스, 파일의 버전정보 관리, 한 사람이 관리

2. 중앙 집중식 버전관리 시스템 – 서버존재, 클라이언트 존재, 서버 데이터를 받아 사용

(여러 사람이 동시에 접근가능한)

클라이언트가 요구한 시간 이전의 버전중 가장 최신을 서버가 전송한다. Natives?

장점 – 누가 무엇을하는지 쉽게판단, 관리가 쉬움

단점 – 서버가 하나밖에 존재하지 않기 때문에 서버 다운 염려, 데이터가 다 날아갔을 때 각자 버전 하나씩만 가지고 있기 때문에 복구가 어려움(로컬데이터의 스냅샷이 명확하지 않음)

3. 분산버전 관리 시스템 – 서버존재, 클라이언트 존재, “서버의 모든 데이터 복제”(clone)

Clone 은 서버의 모든 데이터를 가져오는 것

모든 데이터를 클라이언트에 복제, 서버 다운시 클라이언트 기반 복구 가능

SCM – 형상관리

Git – 리눅스 개발을 위해 만듬

빠른속도, 단순한구조, 비선형적인 개발(수천개의 동시다발적인 브랜치), 완벽한 분산

대형프로젝트에도 유용할 것 (속도나 데이터 크기 면에서)

- 최종 버전만 모아놨을 때 10G 라면 중앙집중형은 개발을 위해 10G를 가져와야 한다.

Git clone은 모든 데이터를 가져오기 때문에 10G보다 많이 가져오고, 불리해질 수 있다.

대신 모든버전이 local에 있어서 버전 왔다갔다 할 때 데이터 트래픽이 적다. 다 로컬에 있으니까

Add, commit 까지는 로컬 리포지토리 안에서 일어나는 일

리모트 리포지토리에는 push 할때 반영이 된다.

이때부터는 네트워크를 탄다.

Git scm은 .git폴더가 존재하는 폴더의 하위폴더를 모두 다 git가 관리를 한다.

변화량만 저장을 하기 때문에 효율적이다

Git0

Central version control system – 중앙 버전관리 시스템

각각 파일의 최신버전을 가져오는 것

버전5에서 버전4로 가면 현재폴더의 데이터는 사라지고 버전4의 파일들로 채워진다

GitVCS – 버전별의 파일들을 다 저장해놓는게 아니라, 파일의 링크만 저장해놓는 형식

그래서 동일한 파일을 여러번 저장은 안한다. 이런 링크의 리스트를 스냅샷이라 한다

Branch – 여러가지 파일들의 버전. Ex. Version3-1, 3-2, 3-3

Git는 데이터를 체크섬형태 및 해쉬코드로 관리 같은파일은 해쉬코드가 같다

workingDirectory – 작업 디렉토리

StagingArea – add 하면 저장되는 영역

.git directory - commit하면 저장되는 영역

.git/objects폴더에 실제 데이터가 저장됨

스테이징 에어리어에서 추가적인 수정이 필요한것들까지 해서 repository에 불필요한 버전이 올라가는것을 방지한다

Add 하면 해쉬코드가 생성되고 관리가 되기 시작한다

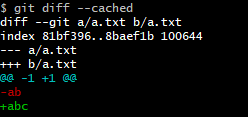
Repository에 저장될 준비를 한다

A와 ab를 저장한 파일 a.txt는 object폴더에 각각 다른 해시로 저장이 된다

Staging area와 working디렉토리의 내용이 같으면 초록색, 다르면 빨간색

파일 이름과 상관없이 내용이 a 라면 폴더는 따로 생성되지 않는다

Commit의 해쉬는 commit의 아이디를 의미한다. 무결성과는 상관없다. 컴퓨터마다 다르다.



a/a.txt 와 b/a.txt를 비교한다

ab와 내용 abc가 다르다

깃의 히스토리는 스냅샷, 해당하는 해쉬코드(하나의 해쉬코드는 하나의 파일)의 블록

Git 브랜치

Tree – 실제 데이터가 있는곳의 포인터

Blob이 있는곳이 스냅샷?

파일이 바뀌면 blob의 포인터가 바뀐다

브랜치는 가지 끝부터 뿌리까지를 의미한다

브랜치 2번으로 다시 돌아가서 작업을 하게되면 commit한게 바뀌는 것이 아니라

새로운 branch가 생성된다

Unmodifiyed 워킹디렉토리와 리포지토리와 다를 때

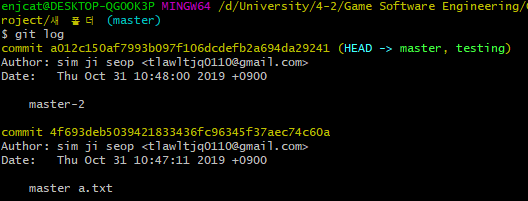
Head 내 워킹디렉토리가 지칭하고있는 commit ID

Branch를 옮겨가면 그곳의 제일 마지막 commit

Tracking이 되고있는 파일에 한하여 git commit -a는 add를 해주는 옵션이다

Head - 현재 워킹디렉토리의 근간을 이루는 commit ID

마지막으로 commit된 id



브랜치 이동할 때 스냅샷에 있는 체크섬형태의 해쉬코드를 비교하여

다르면 워킹디렉토리를 바꾼다

브랜치의 커밋아이디가 서로 같으면 stagingarea에 있는것도 괜찮다

Git Merge testing 현재의 branch에 testing을 merge한다

Threeway merge 공통조상의 스냅샷, 머지할스냅샷(into), 머지를위해가져올스냅샷(in)

Ancestor 와 Into를 비교

-둘이 같을 때 (스냅샷이 같음)- IN이 다르면 fast forward(달라진 것을 우선시)

-셋다같으면 아무일도 안일어남

-셋다다르면 자동merge로 합치다가 conflict를 냄

clone한것도 remote에서 origin이라는 branch를 가지고 있다. Pull 은 이 branch와 remote를 merge하는것이다.

Git6.

HEAD를 옮기는 명령어들

Reset, Revert, Checkout

Checkout할 때 작업하던 내용(working directory)를 필수적으로 고려한다.

내부적으로 동작하는 것이 서로 다르다.

Head가 옮겨갔을 때 워킹,스테이징,리포지토리가 동일할 때 클린하다고 말할 수 있다.

빨간색 unkracked, modified. 이 파일이 스냅샷에 포함이 한곳에라도 되어있어야 tracked

파일이 직접 저장된 해시폴더 – blob

Head를 옮길때는 역순으로 head->index->working으로 바뀐다.

Reset은 같은 branch내에서 정상적으로 작동한다. commitID로 버전이 이동된다.

HEAD – 워킹 디렉토리를 이뤄아햐는 근간이 되는 commitID 브렌치와 연관이 없다.

Branch가 가지는 정보 = commit ID 하나. 결국 commitID하나만 바꾸고 끝나는 것

리셋은 브렌치가 가지는 커밋을 바꾸는거, 체크아웃은 헤드를 옮기는 것

동일한commitID를 제외하고 브랜치를 옮길때는 HEAD와INDEX를 맞춘다

Git reset hard와 비슷하다. 저장되지 않은 워킹디렉토리의 정보는 위험하다.

인덱스와 워킹디렉토리가 같으면 초록색, 다르면 빨간색

Head index working

B B A

A,b a,b a

빨간색 Deleted

Head index working

B A A

A,b a a

Ready to commit 초록색deleted

Reset은 꼭 같은 branch로만 가는 것이 아니다.

상태볼 때 head와 workingdirectory의 다름을 확인하고, 이게 index와 같으면 초록, 다르면 빨강

Reset은 스냅샷을 기준으로 날린다. Untraked file은 지우지 않는다.

Git stash apply –index 하면 다 index로 올리는 것이 아니라, stash할때의 상황에 따라 다름

Commit은 untracked를 반영하지 않음

9 remote repository

접근할 수 있는 주소만 있으면 서버를 만들어 낼 수 있다.

서버에 해당 브랜치가 없으면 push할 때 날아갈 수 있다

클론할 때 싹 긁어오고 서버의 브랜치를 origin/master로 바꾸로, master브랜치를 만들고 checkout 한것이다.

브랜치가 다른브랜치를 트래킹하게 할 수 있다.

Fetch. 연동되어있는 브랜치를 업데이트 하는 것

Pull == fetch + merge

로컬이 다른 작업이 있을경우 그 브랜치와 서버의 fetch된 브랜치와 3way merge

그래서 대부분 conflict가 나는것이다.

Remote가 local과 다를때는 reomote가 날아갈 수 있어서 push못하고, pull받으면 conflic이 날것이다.

그떄는 fetch만 하면 된다.