http://cycorld.com/20161117-git.pdf

2016. 11. 9

최용철

charleschoi87@gmail.com

http://cycorld.com/20161117-git.pdf

목차

- GIT 소개
- GIT의 개념과 명령어
- GIT으로 협업하기
- GIT의 기타 기능들

http://cycorld.com/20161117-git.pdf

수업의 목표

- PM(개발팀장)과 소스코드 관리에 대해 대화가 가능한 수준이 첫번째 목표입니다
- PM으로서 일할 때 git에 관해 알아야 할 것들을 이해하는 것이 두번째 목표입니다

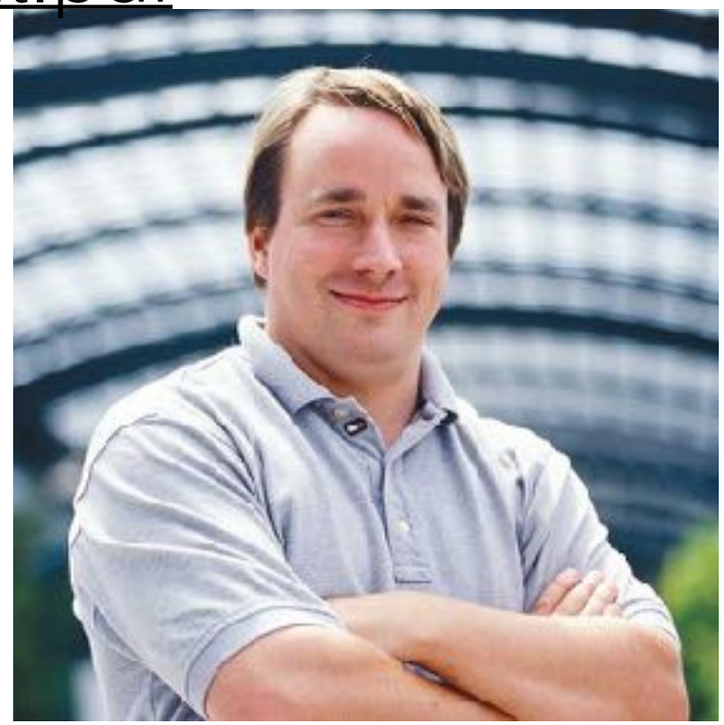
http://cycorld.com/20161117-git.pdf

수업 시작 전 질문

cycorld.com/link/pre-q.html

http://cycorld.com/20161117-git.pdf

git 소개



- GIT의 목표
  - 빠른 속도 / 단순한 구조
  - 비선형적인 개발(수천 개의 동시 다발적인 브랜치)
  - 완벽한 분산
  - Linux 커널 같은 대형 프로젝트에도 유용할 것(속도나 데이터 크기 면에서) : https://github.com/torvalds/linux

Git은 동시다발적인 브랜치에도 끄떡없는 **슈퍼 울트라 브랜칭 시스템**이다

컴공 학부 1학년의 흔한 팀플

- "그 부분 내가 코딩할거니깐, 지금까지 작업한거 보내줘"
- "응? 이거 최신판 아닌데.. 여기 고친거 다 없어졌네?"
- "??? 헐;; 잘못해서 덮어씌운거같다.. 미안;;"

git을 사용하면 가능한 것들

- 소스코드 주고받기가 필요 없고, 같은 파일을 여려 명이 동시에 작업하는 등 병렬 개발이 가능해지며, 버전 관리가 용이해져 생산성이 증가

- 소스코드의 수정 내용이 커밋 단위로 관리되고, 패치 형식으로 배포할 수 있기 때문에 프로그램의 변동 과정을 체계적으로 관리할 수 있고, 언제든지 지난 시점의 소스코드로 점프(Checkout)

- 새로운 기능을 추가하는 Experimental version을 개발하는 경우, 브랜치를 통해 충분히 실험을 한 뒤 본 프로그램에 합치는 방식(Merge)으로 개발을 진행

- '분산' 버전관리이기 때문에, 인터넷이 연결되지 않은 곳에서도 개발을 진행할 수 있으며, 중앙 저장소가 폭파되어도 다시 원상복구 가능

- 비단 팀 프로젝트가 아닌, 개인 프로젝트일지라도 GIT을 통해 버전 관리를 하면 체계적인 개발이 가능

GIT의 개념과 명령어

git을 구성하는 3가지 Object

- Blob —> Git에서는 모든 파일을 Blob이라는 단위로 관리. Blob은 텍스트 파일일 수도 있고 이미지 파일일 수도 있음

- Tree —> Blob들을 모아 놓은 것을 Tree

    - Tree에는 Blob뿐만 아니라 Tree가 있을 수도 있음

- Commit

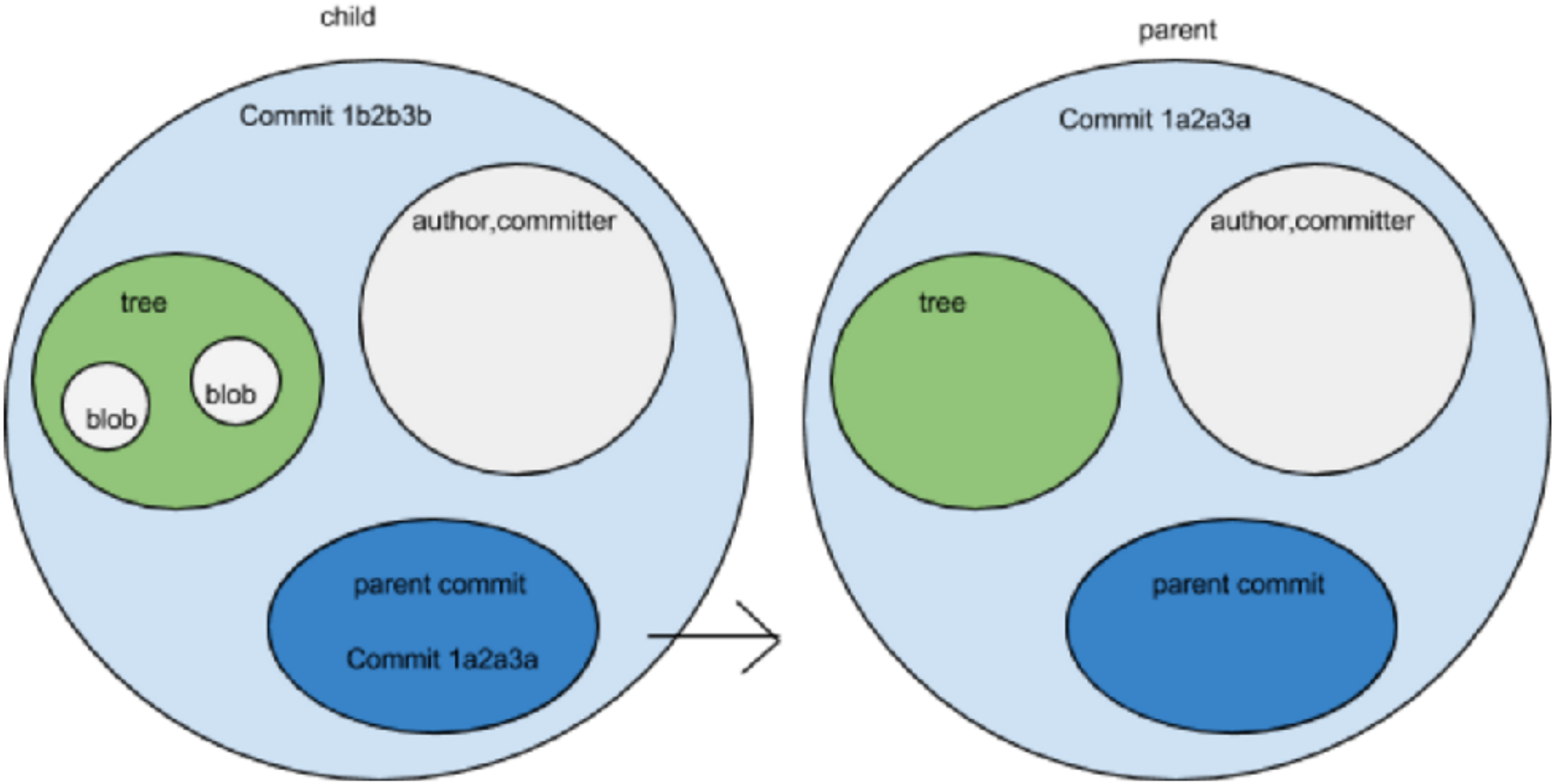
    - author, committer —> commit을 최초 작성한사람, 수정한 사람을 나타냄. 이 값에 시간이 timestamp값으로 들어가 있어 commit의 해시 값이 중복되지 않을 수 있음.

    - repo tree —> Git으로 관리하는 repository의 tree. 즉, 최상위 폴더의 tree이다.

    - parent commit —> 부모 commit의 해시 값을 가지고 있음

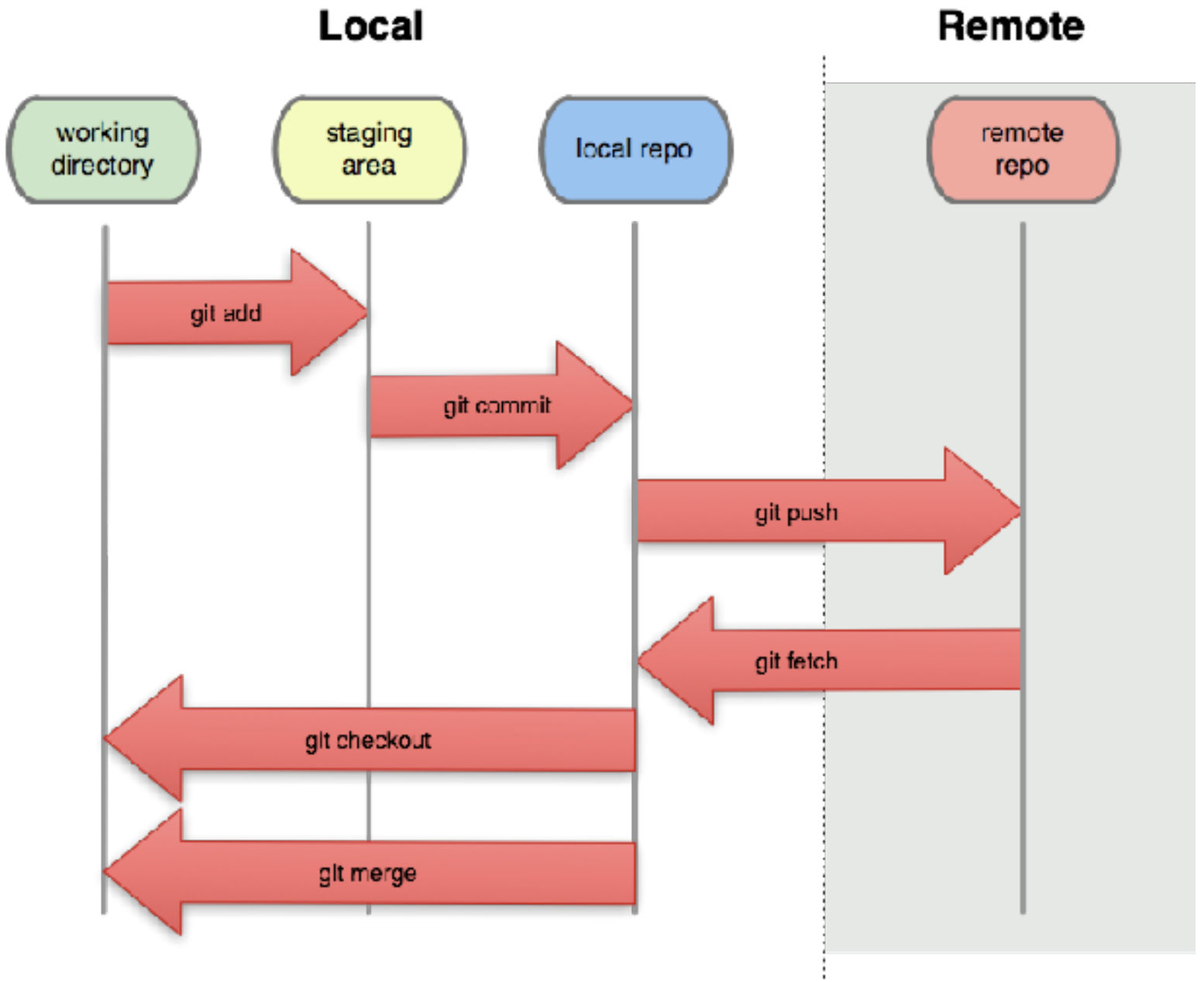
- 이렇게 세가지의 요소가 있는데 모두 고유의 SHA-1해시 값을 ID처럼 사용

git의 세부 구조



git cat-file -p {object-id}

git의 큰 구조



git pull == git fetch + git merge

커밋 commit

- 작업한 내용을 로컬 저장소에 저장하는 과정. 각각의 커밋은 의미 있는 변경 단위이고, 변경에 대한 설명을 커밋 로그로 남김. 대개 하나의 커밋은 '회원 가입 기능 추가', '검색 버그 수정'과 같이 하나의 주제로

- 프로젝트 팀에 따라 커밋을 하는 단위가 서로 다르고, 커밋 로그를 작성하는 형식(Format)도 정해져 있음. 특히, Continuous Build System과 같이 원격 저장소와 연동된 자동화 시스템을 사용하고 있는 경우, 이 자동화 시스템이 인식할 수 있도록 엄격한 형식에 맞춰서 커밋 로그를 작성해야함.

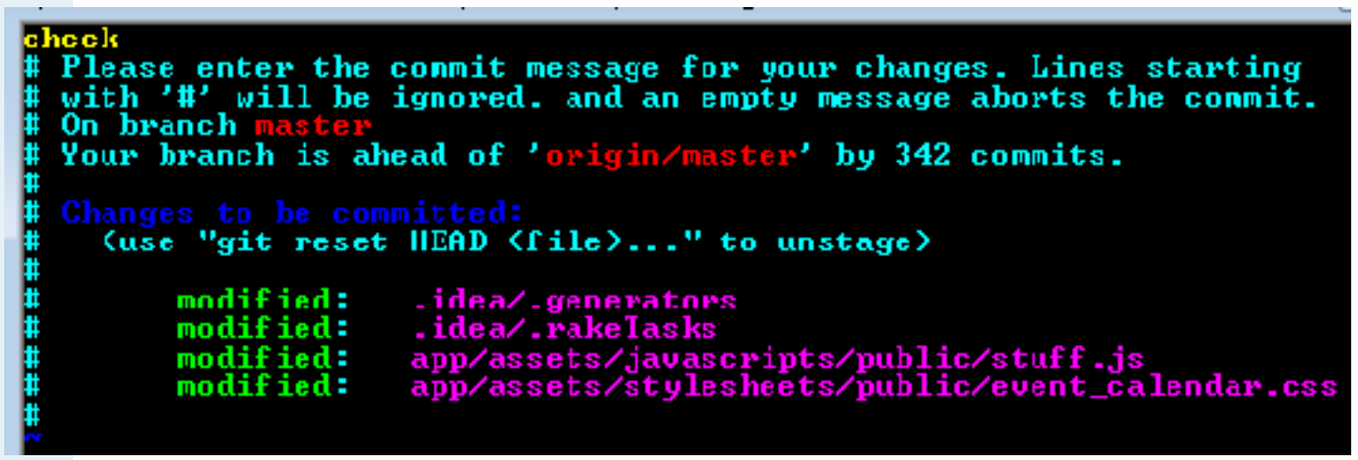
- 첫번째 줄은 한 줄 요약(80자 이하), 두번째는 비우고 세번째 줄 부터 상세 내용 작성

- https://item4.github.io/2016-11-01/How-to-Write-a-Git-Commit-Message/

Username is now the primary way to reference a user

Users where being referenced mostly by email, but on the mobile version they are referenced by username. This makes it easier for them by making the username more important.

* Admin can view and edit user's username
* User can edit his username
* Top nav bar displays username instead of first name
* Username is required upon sign up
* Login is possible with username or email



저장소 repository

- 소스코드가 저장되어 있는 여러 개의 브랜치(branch)들이 모여 있는 디스크 상의 물리적 공간

- 로컬 저장소 vs 원격 저장소 : 본질은 같다 (원격 저장소는 working directory 가 없는 “bare” repository)

- 작업을 시작할 때 원격 저장소에서 로컬 저장소로 소스코드를 복사해서 가져오고(Clone), 이후 소스코드를 변경한 다음 커밋(Commit). 이 때, 커밋한 소스는 로컬 저장소에 저장되며, 푸시를 하기 전에는 원격 저장소에 반영되지 않음

- 위와 같이 임의의 사용자가 마음대로 다운받을 수 있도록 공개된 저장소가 있는 반면, 인증된 사용자만 접근할 수 있는 비공개 저장소도 있음. 단, 공개된 저장소일지라도 저장소의 다운로드만 가능하며, 수정된 코드를 저장소에 반영하기 위해서는 저장소 관리자의 허가를 받고 SSH 키를 등록해야함.

원격 저장소들

- 수많은 오픈소스 프로젝트

- 자체 서버를 세팅하고자 할 때

- 5인 이하 소규모 팀에 적합
  (무제한 private repository)

체크아웃 checkout

- 특정 시점이나 브랜치의 소스코드로 이동하는 것을 의미. 체크아웃 대상은 브랜치, 커밋, 그리고 태그. 체크아웃을 통해 과거 여러 시점의 소스코드로 이동 가능.

- git checkout master : master 브랜치로 이동
- git checkout a1d3f21 : a1d3f21… 커밋으로 이동
- git checkout v1 : v1 태그로 이동
- git checkout -- {filename} : HEAD의 file을 복원

풀 pull

- 푸시와 반대로 원격 저장소에 있는 내용 중 로컬 저장소에 반영되지 않은 내용을 가져와서 로컬 저장소에 저장하는 과정을 의미. 이를 통해 다른 팀원이 변경하고 푸시한 내용을 로컬 저장소로 가져올 수 있음.

- 푸시 과정에서 충돌(Conflict)이 일어나서 푸시가 거절된 경우, 풀을 통해 원격 저장소의 변경 내용을 반영한 뒤 다시 푸시를 시도함.

- pull 은 fetch 후 merge를 한번에 한 것

브랜치 branch

- 브랜치 : 커밋을 단위로 구분된 소스코드 타임라인에서 분기해서 새로운 커밋을 쌓을 수 있는 가지를 만드는 것, 혹은 그 가지

- 브랜치 중에 개발의 주축이 되는 브랜치를 마스터 브랜치(Master Branch)라 하며, 모든 브랜치는 마스터 브랜치에서 분기되어 최종적으로 다시 마스터 브랜치에 병합(Merge)하며 개발 진행

- 로컬 브랜치 확인 : git branch

- 원격 브랜치 확인 : git branch -r

- 전체 브랜치 확인 : git branch -a

브랜치 branch

- 현재 위치에서 새로운 브랜치 생성 : git branch {생성할 Branch 이름}
- 다른 브랜치로 이동하기 : git checkout {이동할 Branch 이름}
- 한번에 하기 : git checkout -b {생성하면서 이동할 Branch 이름}
- '다른 Branch로 이동한다'는 것은 현재 작업 디렉토리의 소스코드 상태를 해당 Branch의 상태로 모두 바꾼다는 것을 의미합니다. 따라서, Branch를 이동할 때마다 디렉토리 내에 있는 (추적중인) 파일들이 시시각각 변함
- (이동할 Branch 이름 대신 Commit ID를 입력하면 해당 Commit이 작성된 직후의 소스코드 상태로도 이동할 수 있음.)

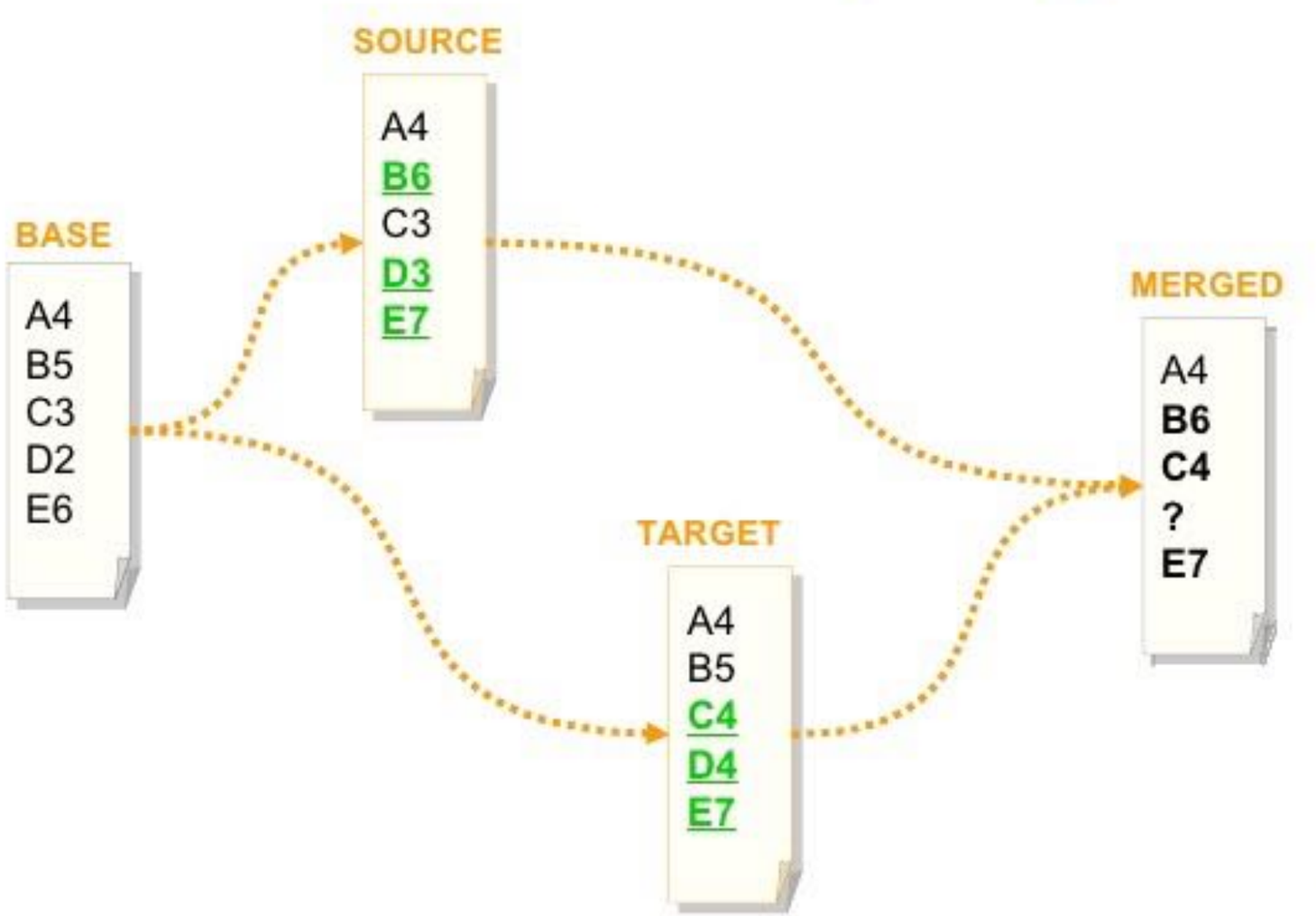
브랜치 branch

- 브랜치 이름 변경
  - git branch -m {기존 Branch 이름} {새로운 Branch 이름}
- 브랜치 삭제 하기
  - git branch -d {삭제할 Branch 이름}
- 원격 저장소에서 브랜치 삭제하기
  - git **push** --delete origin test

병합 merge

- 브랜치와 반대되는 개념으로, 하나의 브랜치에 다른 브랜치를 합치는 과정
- 병합의 기본인 3-Way Merge는 서로 다른 두 커밋으로부터 하나의 새로운 새로운 커밋을 생성하는 작업
- 병합 과정에서 두 개의 브랜치에서 파일의 같은 부분을 서로 다른게 수정한 경우 충돌(Conflict)이 발생하며, 병합이 일시정지. 이 경우, 충돌이 발생한 부분을 직접 수정하거나, Merge Tool 등을 활용하여 충돌을 해결한 뒤 병합을 계속 진행 —> **헬게이트 오픈**
- 보통 Merge 작업은 각각의 팀원이 수행하기 보다는 Project Manager가 일괄적으로 수행하는 것이 일반적. 대개 한 브랜치의 작업이 끝나면 PM에게 Merge Request를 보내고, PM은 병합하기 전 해당 브랜치를 작업한 개발자와 함께 코드 리뷰를 진행한 뒤 이상이 없으면 마스터 브랜치에 해당 브랜치를 병합하는 작업을 수행.

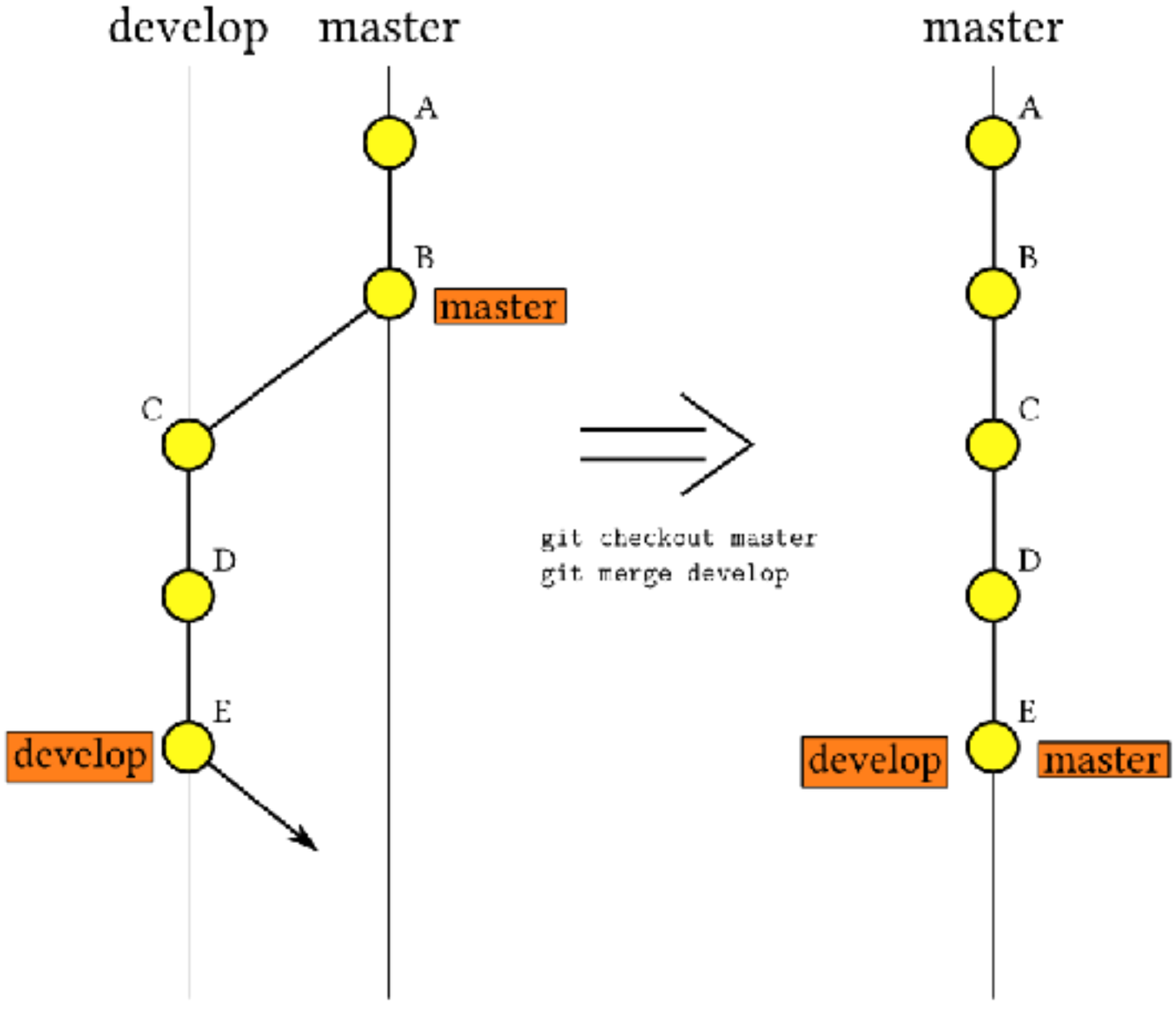
3-way Merge



새로운 merge 커밋이 생성됨

3-way로 merge 하기 : git merge --no-ff develop

Fast Foward Merge



새로운 merge 커밋 없이 1자로 깔끔한 히스토리

git merge develop 명령어 사용시 3-way가 필요 없는 상황일 경우 fast-foward 기본

git 기본 명령어

- git init : 현재 있는 디렉토리를 Working Directory
- git status : 현재 Working Directory 의 상태 확인
- git log --graph --oneline --decorate —all
    - 현재까지 남긴 Commit Log에 대한 정보를 알 수 있는 명령어
    - --graph 옵션은 Commit마다 점선으로 연결되어, 커밋 로그들의 상관 관계들을 그래픽적으로 보여줌
    - --oneline 은 Commit Log 의 내용을 간략적으로 보여주며, 타이틀과 7자리의 해쉬값을 알려줌
    - --decorate 는 HEAD의 위치, refs(브랜치)들의 위치를 알려주는 옵션
    - --all 은 커밋되어 있는 모든 로그를 보여줌
- git show ({commit id}): 해당 Commit의 메시지와 hunks를 보여줌.
    - 현재 (HEAD)와 그 전 커밋(부모)(HEAD^)의 대한 내용을 비교하고 달라진 점을 보여줌. [git show ~] 와 같이 show 뒤에 특정 해쉬값을 붙이게 되면 그 해당 커밋과 그 부모의 커밋의 달라진 내용을 보여줌

git 기본 명령어

- git diff {commitid} {commit2} [—staged]
    - git diff는 두 커밋 사이의 차이를 확인하는 명령어
    - staged 된 상태를 확인하고 싶을 때 그럴 경우에는 --staged 또는 --cached 옵션으로 확인
- git add {파일명/-A/-all/.} [-p]
    - 새로 만들어졌거나 변경된 파일을 staging에 올리는 명령어
    - -A 또는 -all 옵션 : 수정 또는 삭제된 모든 내용을 staging
    - 파일명 또는 . : 특정 파일 또는 모든 파일을 staging
    - p (partial) 옵션을 통해 hunk(코드 덩어리)를 분리할 수 있음
- git commit : 커밋을 남기는 명령어 (-a 옵션을 주면 git add . 효과가 있음)
- git commit -m “커밋 메시지” : 메시지 한번에 남겨서 커밋하기 (엔터로 줄바꿈 가능)

git 기본 명령어

- git fetch : remote의 정보를 가져옴 / git fetch --tags
- git pull
  - remote의 정보를 가져와서 합침(병합)
    - git pull {remote 별칭} {branch 이름} : git pull origin master
- git push {remote 별칭} {branch 이름}
  - remote의 branch로 커밋을 보내기
  - git push -u origin master —> upstream을 설정 : 한번 설정하면 로컬의 브랜치와 리모트의 브랜치가 연결되어 다음부터는 명시 없이 push/pull 가능
  - git push origin master:test —> 로컬의 master 브랜치를 origin 리모트의 test 브랜치로 push

git 알아두면 좋은 명령어

- git reset --soft --hard {대상}
  - 현재 working directory 또는 stage를 reset하는 명령어. staged 혹은 잘못 Commit 한 것을 되돌리고 싶을 때 주로 쓰는 명령어. 대상에는 HEAD, HEAD^, HEAD~N, Commit-id 코드 등 사용 가능
  - --mixed 는 아무런 옵션도 주지 않았을 때, 기본적으로 작동하는 옵션으로서 현재 stage를 해당 커밋과 같게 만듦. --soft 옵션은 현재 stage를 유지시킨 채로 해당 커밋과 같게 만듦
    - --hard 는 stage 뿐만이 아니라, 현재 working directory의 상태까지 해당 커밋과 같게 만듦
- git rm
  - git rm은 “rm”(파일삭제) + git add -A 한 작업과 같다
  - git rm --cached 는 실제 파일은 삭제하지 않지만 git 버전관리에서 삭제
- git mv {수정 전 경로/파일명} {수정 후 경로/파일명}
  - git rm 과 마찬가지로 리눅스 mv를 하면 staged 되지 않은 채로 어색함. git mv 로 파일을 옮기거나 이름을 바꾸면 stage 까지 알아서 됨

git 알아두면 좋은 명령어

- git revert {commit id}
    - 특정 Commit과 같은 내용으로 되돌리되, 새로운 커밋이 추가되는 방식이라 기존의 히스토리는 유지
- git stash
    - stash는 현재 진행되고 있는 상황을 저장하고, 그 이후에 다른 행동을 취하다가 다시 돌아올 수 있음. 작업 중인데 pull할 일이 생기면 보통 커밋을 요구받는데, stash를 유용하게 사용할 수 있음
    - git stash pop 으로 복원
- git reflog : 명령어 히스토리 보기
- git clean -fd : 현재 git status 에서 untracked 된 파일들을 제거하는 명렁어. -d 옵션은 디렉토리를 의미

git 알아두면 좋은 명령어

- git cherry-pick {commit-id}
  - 특정 Commit 하나를 가져오기
- git rebase {branch|commit} [-i]
  - 여러 Commits 를 가져오기
  - 현재 브랜치의 base를 바꾸기
  - git rebase -i —> git **리베이시**라고도 부름 : 히스토리를 마음대로 주무를 수 있음

git tags

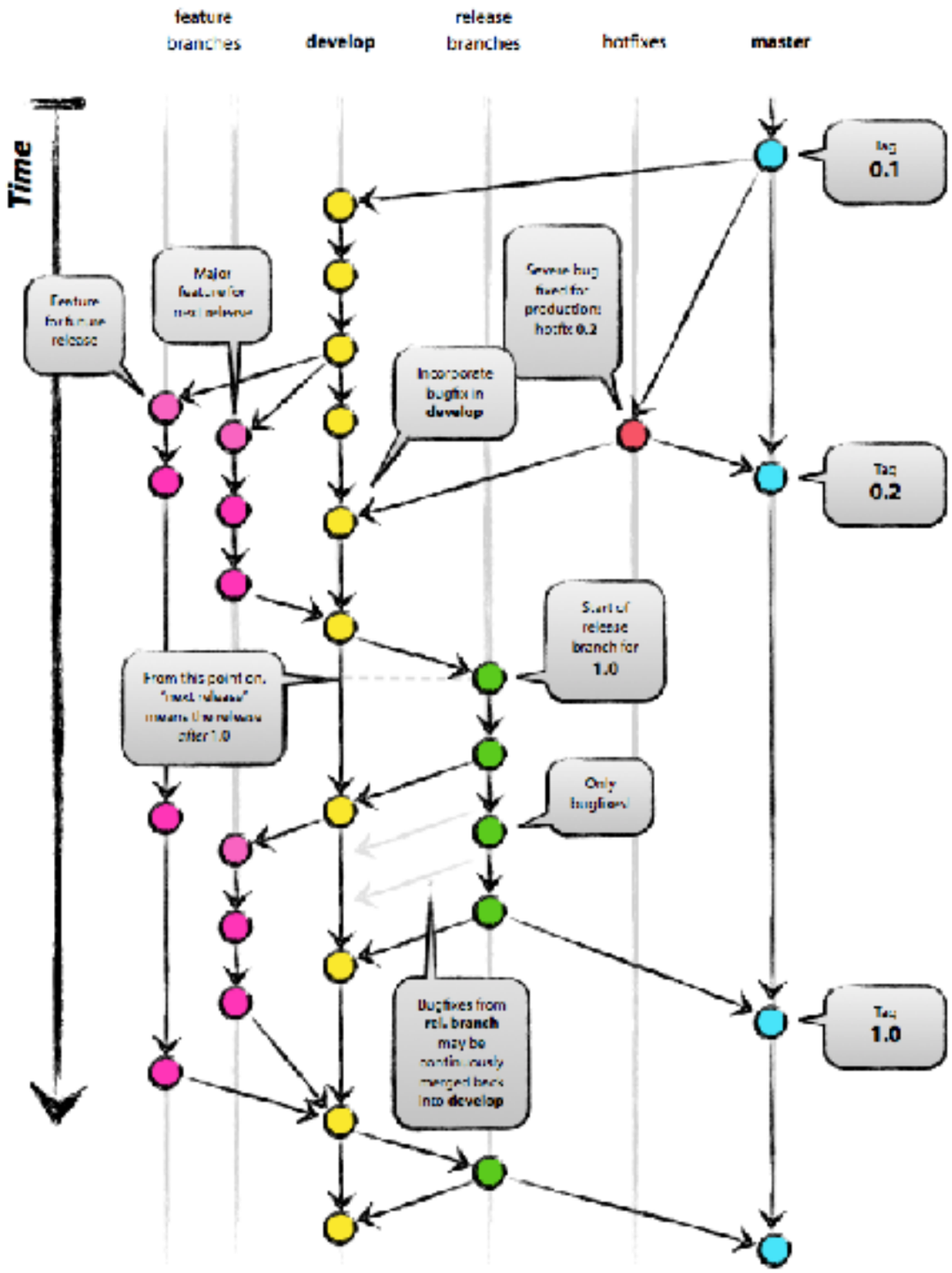
- Lightweight Tag
    - git tag v0.1 —> 현재 HEAD 커밋에 태그 부여
    - git tag v0.0.1 a1b2c3 —> a1b2c3 커밋에 태깅
- Annotated Tag
    - git tag -a v0.2 -m “{태그 메시지}” —> 태그에 메시지 남기기
- tags push 하기
    - git push origin v0.1
    - git push origin --tags
- tags 받아오기 : git pull —tags
- 자세한 내용 : https://git-scm.com/book/ko/v1/Git의-기초-태그

퀴즈

cycorld.com/link/git-quiz.html

GIT으로 협업하기

git flow 전략



- master : staging 용 - tag 달기
- develop : 개발용
- hotfixes : 버그 수정
- features : 기능 개발
- release : 배포용
- 참고 자료 : http://dogfeet.github.io/articles/2011/a-successful-git-branching-model.html

pull request 방식

- 첫번째 : 다른 사람의 원격 저장소(repository)를 fork 뜬 후 개발하고 pull request 보내기 —> 오픈소스 프로젝트에 많이 쓰이는 방식
    - 원본 저장소 주소를 새로운 remote 로 등록해둡니다 : git add remote upstream {github 주소} / git fetch && git merge upstream/master
- 두번째 : 하나의 원격 저장소에서 branch 간에 pull request 보내기 —> 소규모 팀에서 작업하기 적합
- 오픈소스 개발자 용어
    - committer : 직접 커밋할 권한이 있는 자
    - contributor : 풀리퀘스트로 기여한 자
- 참고 자료 : http://dogfeet.github.io/articles/2012/how-to-github.html

Rebase vs Merge

- Rebase와 Merge 모두 새로운 작업 내용을 Master branch와 합치는 데 사용할 수 있는 유용한 명령어. 다만, 이 둘을 각각 언제 사용해야 할 지 잘 모르겠다면 PM에게 물어봐야

- Rebase : 각각의 commit ID 바뀜

    - 로컬 저장소에서 작업을 위해 개인적으로 만든 Branch를 Main stream branch에 반영하여 Push 하기 위해 사용. 로컬에 개인적으로 만든 Branch는 Push 이후 정리를 하면서 삭제. 대개 Hotfix와 같이 단기간에 끝나는 Topic의 경우 이 방법으로 작업. 깔끔한 히스토리 유지

- Merge : 각각의 commit ID 유지

    - 로컬 저장소에서 분기하고 작업한 Branch를 리뷰 등을 위해 원격 저장소에서 Push한 뒤, Review가 끝나고 PM이 Main stream branch에 병합하는 순서로 작업을 진행할 때 사용. Branch가 그대로 유지되기 때문에 나중에 추가 작업을 해야 할 경우나, 큰 규모의 Topic인 경우 이 방법으로 작업.

2인 팀을 구성하여 실습

- github 가입하기

- 선과 후를 정해주세요

시나리오 A : 몸 풀기

- 선 : 레파지토리 생성, README.md 파일에 노트 작성 후 커밋 & 푸시
- 후 : 리모트에서 새 프로젝트 clone 받기
- 후 : 노트에 새로운 내용을 추가하여 커밋 후 푸시

- 선 : 풀 받아 오기
- 선 : 노트에 새로운 내용을 추가하여 커밋 후 푸시
- 후 : 풀 받아 오기

—> 싱크 맞추기

- 반복 연습

시나리오 B : 충돌

- 선 : 첫번째 줄 수정 후 **커밋** & **푸시**
- 후 : 마지막 줄 수정 후 **커밋** & **푸시 (!!!)**

- 선 : **풀** 받고, 마지막 줄 수정 후 **커밋**
- 후 : 마지막 줄 수정 후 **커밋** & **푸시**
- 선 : **푸시**하기 **(!!!)**
- 후 : **풀** 받아 오기

시나리오 C : 커밋 쪼개기

- 선 : README.md 의 첫줄, 마지막 줄 수정
- 선 : **각 line 별로 커밋** 하고 (커밋 2개) **푸시**하기
- 후 : head-1.md / body-2.md 파일 만들고 내용 작성
- 후 : **각 파일별로 커밋**하고(커밋 2개) **푸시**하기 (**!!! rebase 하여 풀** 받기 : history oneline 유지)
- 순서를 바꿔 다시 해보기 (head-2.md / body-2.md)

시나리오 D : stash

- 선 : README.md 의 아무 내용이나 수정 중
- 후 : head-1.md 수정 후 **커밋** & **푸시**
- 선 : **풀** 받기 **(!!!)**, **stash** 한 후 **풀** 받기
- 선 : **stash 팝** 하여 마저 수정하고 **커밋** & **푸시**
- 후 : **풀** 받기
- 순서 바꿔 다시 해보기

시나리오 E : 브랜치

- 선 : dev **브랜치** 만들기
- 선 : 2개의 **커밋** 작성 & **푸시**
- 후 : dev 브랜치 **풀** 받기
- 후 : dev 브랜치에 2개의 **커밋** 작성 & **푸시**
- 선 : dev **풀** 받은 후 master로 **머지** & master **푸시**
- 서로 바꾸서 다시 해보기 (feature 브랜치)

시나리오 F : git-flow

- 후 : hotfix **브랜치** 만들기 & 아무거나 수정 후 **커밋** & **푸시**
- 선 : hotfix 브랜치 **풀** 받고 코드 검토 후 **master로 머지** & **master 푸시**
- 후 : master **풀**
- 순서 바꿔서 다시 해보기 (hotfix-2 브랜치)

시나리오 G : 풀리퀘스트

- 선 : Github 이슈에 오타 발견 **이슈 등록**
- 후 : hotfix **브랜치**를 만들고 오타 수정 후 커밋(메시지에 fixed #1 포함) 후 origin에 **hotfix 브랜치 푸시**
- 후 : **pull-request 생성**
- 선 : 리뷰 후 **머지**하기 (버튼 클릭)
- 순서 바꿔서 다시 해보기
- 이것저것 연습해보기!

GIT의 기타 기능들

git 세팅

- 사용자 정보 설정 (시스템 전체, 각 저장소 별로도 수정 가능)
    - git config --global user.name "John Doe"
    - git config --global user.email johndoe@example.com
- 메시지 편집기 설정
    - git config --global core.editor vim : emacs, atom 등
- 색상 강조 사용하기 : git config --global color.ui true
- 단축어 만들기 : git config --global alias.사용할키워드 '명령어'
- 설정 확인
    - git config --list : 나중 key가 먼저 (global < specific)
    - git config {key}

git notes

- git notes : commit 변경 없이 메시지를 추가하기
- 커밋에 노트 추가하기
    - vim으로 편집 : git notes add {commits}
    - git notes add -m 'I approve - Charles' master~1
- git log 볼 때 노트도 함께 보기
    - git log --show-notes=*
- 원격 저장소 push / fetch 하기 : notes는 tag와 마찬가지로 따로 관리됨
    - git push origin refs/notes/*
    - git fetch origin refs/notes/*:refs/notes/*

git hooks

- hook은 어떤 이벤트가 발생했을 때 실행되는 스크립트. 클라이언트 hook은 커밋이나 Merge 할 때 실행되고 서버 hook은 Push 할 때 서버에서 실행

- .git/hooks 디렉토리에 샘플 있음 (이름만 바꿔주면 바로 사용 가능)

- 사용 예 : 테스트를 통과하지 못한 코드는 커밋조차 못하게 막을 수 있음 / 자동으로 커밋 메시지에 내용 추가 / **push 알림** 등등등

- 자세한 내용 : https://git-scm.com/book/ko/v2/Git맞춤-Git-Hooks

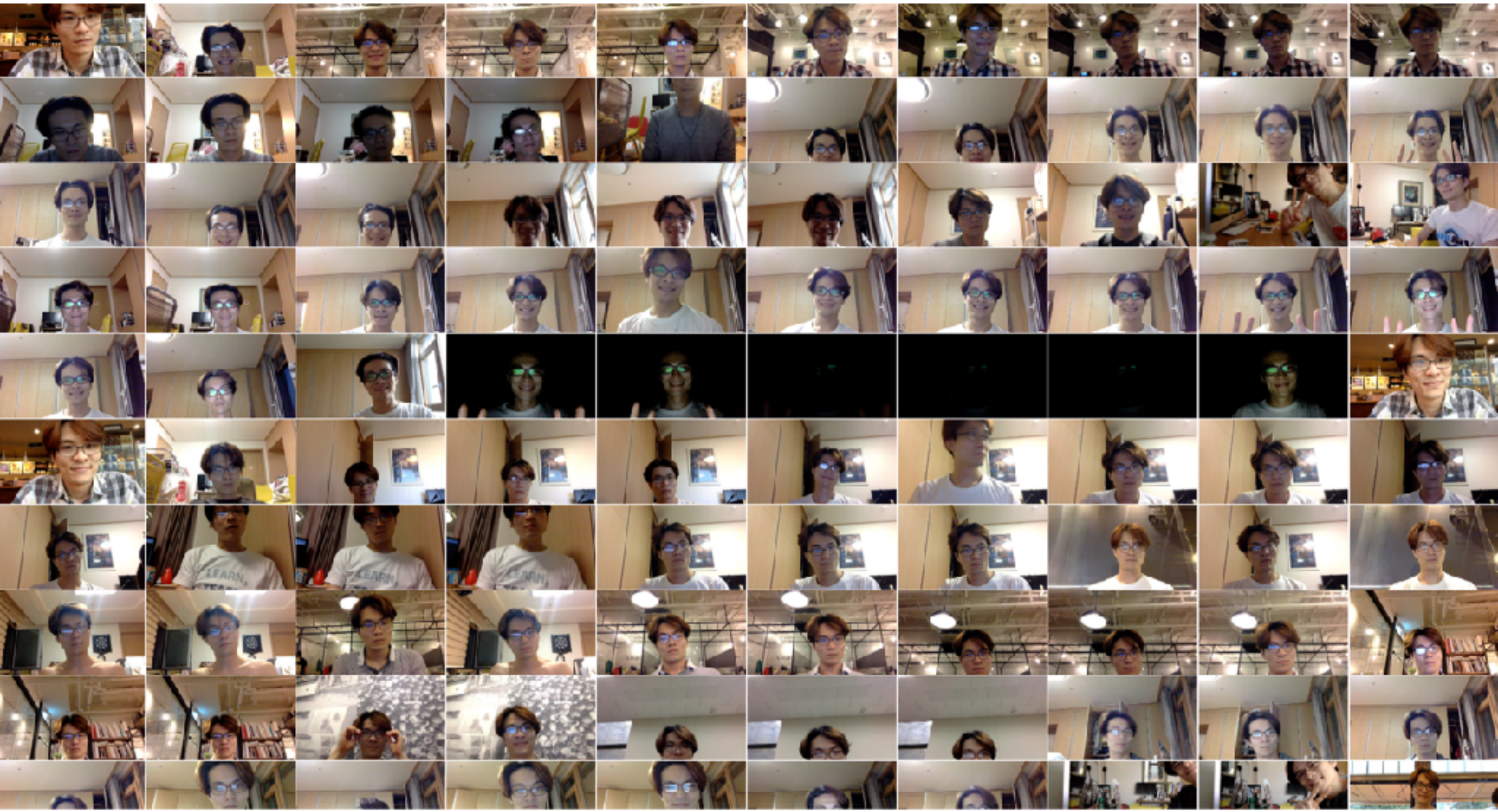
git hooks Example

- 맥북에서 commit 할 때마다 셀카 찍기 ㅋㅋ
- mkdir ~/.gitshots // brew install imagesnap
- ~/.git-templates/hooks/post-commit

```ruby
#!/usr/bin/env ruby
project = `TOP=$(git rev-parse --show-toplevel); echo ${TOP##*/}`.chomp
now = Time.now
hash = `git rev-parse HEAD`.chomp
file="~/.gitshots/#{project}-#{now.strftime("%Y%m%d-%H%M%S")}-#{hash}.jpg"
unless File.directory?(File.expand_path("../../rebase-merge", __FILE__))
  puts "Taking capture into #{file}!"
  system "imagesnap -q -w 3 #{file} &"
end
exit 0
```

- 출처 : https://coderwall.com/p/xlatfq/take-a-photo-of-yourself-every-time-you-commit

커밋의 기록들



git submodule

- git 저장소 안에서 다른 git 저장소를 관리 할 수 있음
- 프로젝트 내에 다른 프로젝트인 라이브러리를 포함할 때 매우 유용

- 자세한 내용 : https://git-scm.com/book/ko/v2/Git-도구-서브모듈

퀴즈 및 설문조사

http://cycorld.com/link/feedback.html

수고하셨습니다!

최고의 레퍼런스 : https://git-scm.com/book/ko/v2