

제03장

Version Control

산기대학교
컴퓨터공학부
뇌리공학 연구소

학습 목표

소스 코드 관리와 git

원본 참조: 조대협 (bwcho75@지메일)

소스 코드 관리와 git

- 소프트웨어 개발에서, 소스코드의 관리는 매우 중요
- 다양한 버전과 변경 관리, 팀단위 개발을 위해 소스코드를 저장 및 관리할 수 있는 시스템이 필요
 - VCS (Version Control System)
 - 또는 SCM (Source Code Management) System
- 버전 컨트롤 (version control)
 - RCS
 - SCCS
 - CVS
 - subversion
 - git

소스코드 관리 시스템의 주요 기능

- **팀단위 개발을 위한 코드 공유**

- 여러 사람들이 협업을 할 경우, 코드를 각 개발자와, 팀간에 공유

- **접근 제한**

- 사용자의 권한 등급에 따라 접근을 제한

- **다양한 버전(형상) 관리**

- 소프트웨어 개발 버전 또는 릴리즈 (브랜치) 마다, 다른 코드를 저장
 - 예, 릴리즈된 버전이나 마이너 버전에 대한 코드 관리, 패치 코드 관리 등
- 다양한 브랜치 중 두 개 이상의 브랜치를 하나의 코드로 병합(merge) 가능

- **특정 시점 추적**

- 태깅 설명

- **변경 추적**

- 각 코드에 대한 변경 추적 가능
- 누가? 언제? 어떤 이유로 코드를 어떻게 변경을 했는지를 추적하여 문제 발생시 원인 분석

Commit, check Out & merge

- 소스코드 저장소 생성 후 코드를 저장하거나 내려 받기 가능
- Check Out
 - 저장소에서 코드를 내려 받는 행위
- Commit
 - 작성된 코드를 저장소에 업로드 하는 행위
- Check Out된 코드의 경우,
 1. 다른 사람이 편집을 할 수 없고 read만 할 수 있도록 lock 사용
 - 다른 사람이 내가 편집하고 있는 코드를 편집할 수 없기 때문에, 코드 작성에는 문제가 없지만 반대로 다른 개발자는 lock이 풀리기 까지 기다려야 하기 때문에, 협업에 문제가 발생 가능
 2. lock을 걸지 않고 코드를 동시에 2인 이상이 코드를 다운받아 편집
 - 코드를 Commit을 할 경우, 한 파일을 다운 받아서 각자 편집을 하고 저장을 시도할 경우, 코드상 Conflict(충돌)이 발생 가능
 - 즉 개발자 A, B가 소스 코드 버전 1을 Check Out 받았다고 하자, 개발자 A가 소스코드 2번 라인에 "printf('hello world')" 라는 코드를 추가하고 Commit을 해서 코드 버전이 2가 되었다고 했을 때, B도 역시 2번 라인에 "printf('hello developer')" 라는 코드를 추가하고 Commit을 하려고 하면, A와 B가 편집한 내용이 충돌이 발생
 - 둘 다 1번 버전의 코드를 가지고 수정을 하였으나, 공교롭게 개발자 A, B가 같은 파일을 수정하였기 때문에 VCS 입장에서는 B 개발자가 수정한 내용을 반영하게 되면 개발자 A가 반영한 내용이 없어져 버리기 때문에 문제 대두 가능
 - 이런 경우 VCS에서 B가 Commit하려는 코드가 현재 버전인 2 버전에 의해서 편집 된 것이 아니라는 것을 알려주고, 개발자 B에게 선택을 하도록 하는데, 2 버전 코드와 다른 내용을 병합 (merge)하게 하거나, 3버전의 코드를 Overwrite하여 2버전의 코드 변경 내용을 무시할 수 있도록 한다.
 - 대부분의 경우, 다른 개발자의 변경 내용을 무시할 수 없기 때문에, merge를 선택하게 되고, merge는 개발자 B가 2번 버전과의 로직의 차이를 일일이 확인하면서 수동으로 merge

Update

- VCS와 Local의 코드를 동기화 시키기 위해서 사용
 - 개발자가 소스 코드를 Check Out 받아 봤을 때,
 - 현재 작업 버전이 오래되거나, 또는 다른 개발자가 Commit을 먼저 해서, VCS의 소스코드 버전이 올라갔을 때,
- 내가 변경한 코드와 다른 개발자에 의해서 변경된 코드가 Conflict이 발생할 수 있으며
- 수동으로 코드의 변경 부분(차이 나는 부분)을 병합하도록 제공

Tagging

- 코드를 개발 중에, 특정 시점의 이미지에 표시를 해놓는 것
 - 예를 들어, 매일 소스코드에 대한 Tagging을 달아놓으면, 개발 중에 문제가 생겼을 경우에, 특정 날짜의 소스 코드로 다시 돌아가기 가능
- Tagging은 주로, Build 시마다 사용 - 빌드 태그
 - 통상 적으로 빌드 시 에러가 날 경우에 다른 개발자들이 빌드 에러로 인하여 개발을 못하는 경우가 생길 수 있기 때문에, 이런 경우에는 이전 Build시 태그해 놓은 버전으로 소스코드를 돌려서 다시 개발을 수행하고, 문제가 해결 되면, 새로운 코드를 다시 Commit 하는 방식으로 개발을 진행

Branch

- 개발을 진행하다가 특정 목적에 의해서 별도의 작업에 의해 새롭게 만들어진 소스코드의 복사본을 지칭
- 예를 들어 영문으로 된 버전을 중국어나 한국어로 제공하기 위해서는 기존 개발 소스코드의 복사본을 만들어서, 중국어 개발용으로 하나 사용하고, 한국어 개발용으로 하나 사용 한다. 새롭게 만들어진 중국어 버전, 한국어 버전을 브랜치라 호칭
- 메인 브랜치
 - 원래 소프트웨어를 개발하던 소스코드 저장소
 - 또는 trunk version, head version 이라고 호칭
- 코드의 원본에서 용도와 목적에 따라서 새로운 복사본을 만들어 나가기 때문에, 메인 코드에서 복사본을 나뉘어가지 (Branch)라고 하고, 이 모양이 마치 나무와 같은 구조가 되기에, Source Code Tree 또는 Branch Tree라고 한다.

Release Branch

- 새로운 상위 버전 출시 후 하위 버전 사용자로부터 버그 수정 요청 시 개발 진행 중인 소스코드를 보호하기 위해 각 릴리즈 마다 release branch를 발급
- 예를 들어,
 1. 패키지 소프트웨어 개발 프로세스에서, 서버 제품을 개발하여, 출시를 했다고 하자. 현재 개발 중인 메인 Branch 에서 해당 시점에 릴리즈를 했다. 릴리즈한 서버의 버전은 6.1이다. 메인 branch 로는 계속해서 신제품 개발을 이어나가고 7.0 개발을 진행 중이었다.
 2. 이때, 6.1을 사용하던 고객의 버그 수정 요청은 어떻게 해야 할까?
 3. 6.1 코드에 일부 코드만 수정하여 패치를 발급하면 되지만, 메인 branch의 경우 이미 7.0 버전 개발을 위해서 코드 개발이 많이 진행되었기 때문에 6.1 에 대한 코드는 바뀌어서 찾을 수 가 없고 7.0 버전 역시 개발이 완료되지 않아서 수정이 불가능
- 웹서비스 (face book 등)의 release branch
 - 하나의 코드가 하나의 서비스에만 배포가 되기 때문에, 별도의 release branch를 유지하는 것 보다는 release tagging을 한 후에, 이슈가 있을 경우 이슈를 fix한 최신 버전을 다음 release 때 배포 하는 방식등을 주로 사용하기 때문에, release branch의 효용성이 패키지 소프트웨어에 비해 감소

QA branch

- 상품화 하기 위해 개발팀에서 source code를 freezing 하고 QA 팀에 해당 제품을 넘겨서 테스트를 의뢰할 경우, QA팀에서 계속해서 버그 수정 요청
- 이 때, 버그를 main branch에서 계속해서 수정을 하게 되면, main 개발에 많은 수정이 가해지기 때문에 개발에 어려움 발생 (merge 발생과 번거로움), 그리고, main branch에서는 개발이 계속 진행되기 때문에, feature 변경이나 기타 수정이 있을 경우 QA에 의해서 reporting된 버그가 제대로 재현이 되지 않는 경우가 발생 가능
- 그래서 QA에 넘기기 전에 QA branch를 생성하고, 버그에 대한 fix를 이QA branch에서 수행 및 반영
- QA가 모두 완료되고 나면, 이 QA branch에 있는 변경 내용을 다시 main branch로 merge하여, bug 수정 내용을 반영

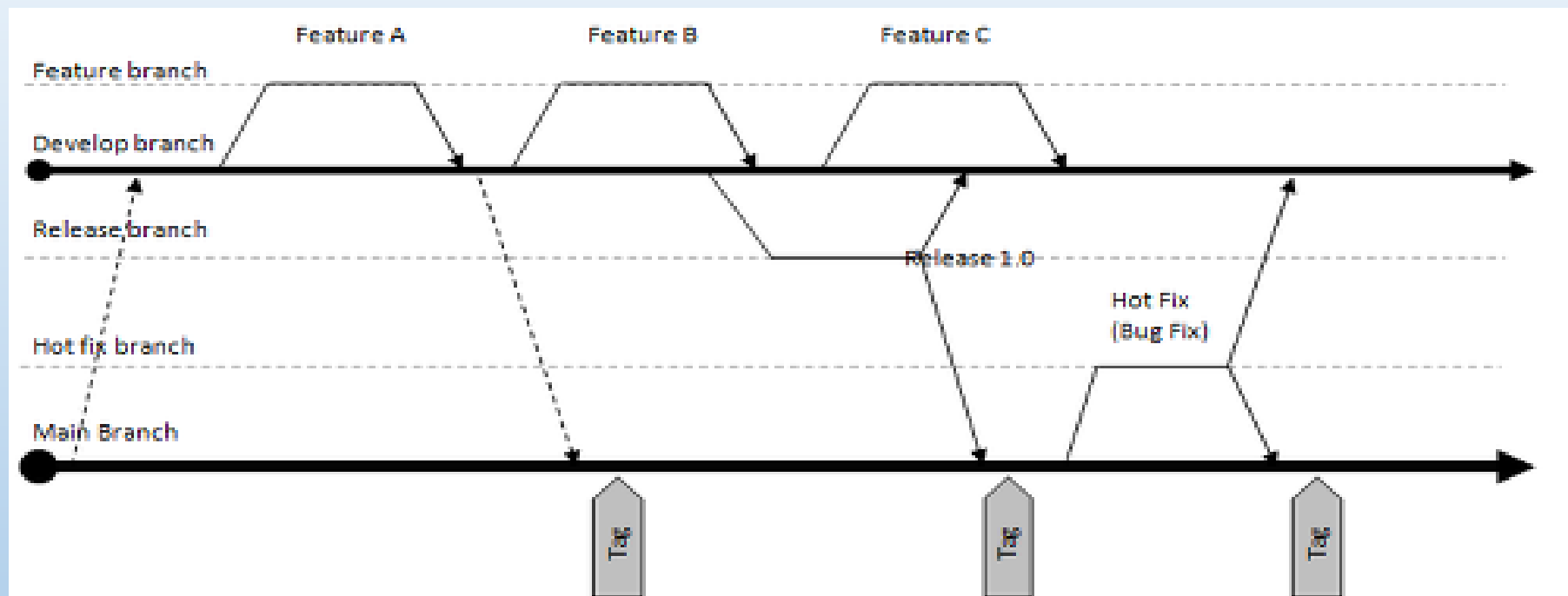
소스 코드 브랜치 관리 전략

- branch는 main code (branch)의 복사본
 - 용도에 따라 만들 수 있고, 필요에 따라 merge 가능
- 패키지나, 웹서비스나와 같은 소프트웨어의 종류와, 팀의 크기와 구조, 릴리즈 정책 등에 따라서 어떠한 branch를 언제 사용할 지가 다르기 때문에, 개발하는 소프트웨어의 버전에 따라서 알맞은 branch 전략 결정
- VCS 제품에 따라,
 - 자사 제품 특성에 맞는 브랜치 구조에 대한 레퍼런스를 제공하거나,
 - 또는 성격에 따라 맞는 레퍼런스 브랜치 모델이 많이 공개적으로 제공
- 오픈소스에서 사용하고 있는 소스 코드의 tree 구조를 레퍼런스

오픈소스 소스 코드 브랜치 관리 전략 - git flow 브랜치 모델

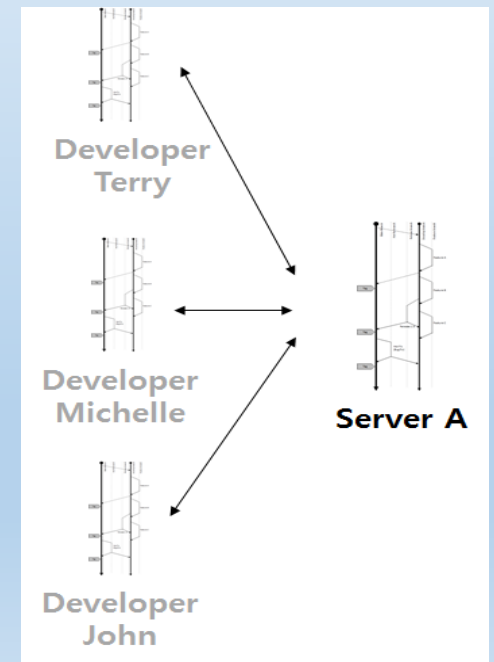
- 오픈소스와 같이 대규모의 분산된 개발 조직이 서비스나 오픈소스 제품을 만드는 구조에 매우 적합
- main branch에서는 개발을 진행하지 않고, 별도의 develop branch를 만들어 개발을 진행
- 기능 (feature) 별로 feature branch를 별도로 만들어 개발을 진행
- feature 개발이 완료되면, develop branch로 merge
- code review
 - 개발자의 코드 수정을 한 경우, Jenkins 등의 빌드 시스템에 통합돼서 빌드되고, 테스트되어야 동작 여부를 확인할 수 있기 때문에, 어딘가 코드를 공유할 수 있는 장소가 필요하다. 즉 개발이 완료된 부분은 먼저 develop branch에 저장되고 컴파일 및 테스트를 끝내고, code review를 위해서 다른 개발자와 reivew를 하고 승인이 되면 그 때 master branch로 반영이 되는 것
- release 시기가 되면, 별도로 release branch를 만든 후에, release에 필요한 각종 configuration file 정리, 기타 매뉴얼이나 문서 등을 합쳐서 release하고, release가 된 버전은 main branch에 반영한 후 tagging
- 오픈소스 프로젝트에 보면, 소스 코드가 날짜나 버전 별로 open 되어 있는 형태
 - main branch는 외부 개발자 공개를 위해서 유지하고, 특별한 release가 있을 때에만 반영하여, 항상 main branch에는 문제 없는 버전이 존재하도록 유지
- main branch를 통해서 공개된 버전이 문제가 있을 경우 별도로 Hot fix (버그 수정용) branch를 만들어 버그를 수정하고, 테스트를 끝낸 후에, 버그 수정 내용을 main branch에 반영하고, develop branch에도 반영

git flow 소스 코드 브랜치 관리 전략



분산 버전 관리 시스템 Git

- Git는 기본적으로 분산 소스코드 관리 시스템
- 모든 개발자가 중앙 저장소에서 작업을 하는 것의 어려움
- 소스 코드가 여러 개의 서버에 다른 복사본으로 존재 가능
- **중앙 집중형 저장소 (Centralized Version Control System)**
 - 중앙 집중형 저장소는, 코드가 저장 서버 단 한군데만 저장
 - 개발자가, 코드를 받아서 수정하고 저장하면, 그 내용이 바로 중앙 저장소에 반영
 - 서버에는 항상 마스터 버전(최신 버전)의 소스코드가 저장
 - 서버가 다운 되거나, 네트워크에 접속할 수 없다면 코드를 commit하거나 최신 코드를 내려 받기 불가



분산 형상 관리 시스템 Git

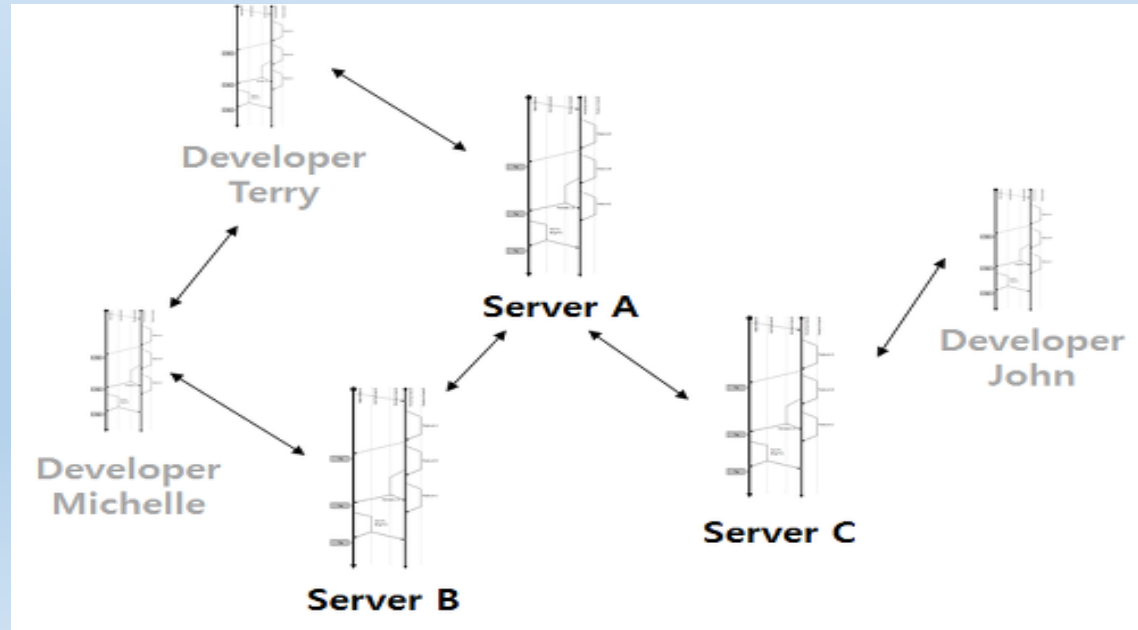
■ 분산형 저장소 (Distributed Version Control System)

- 소스코드가 하나의 중앙 서버가 아니라, 여러 개의 서버나 여러 개의 개발자 PC에 저장될 수 있으며, 각각이 소스 코드 저장소 (source repository)가 된다.
- 각 저장소에 저장되는 소스 코드는 같은 버전의 코드가 아니라 제각기 다른 브랜치 코드가 저장
 - ✓ 즉 서버 A에는 branch A,B,C 버전이, 서버 B에는 branch A,C,D 버전과 같이 다른 브랜치 버전을 저장할 수 있다. 각 저장소에 브랜치 버전이 모두 틀리고, 소스 코드를 access해서 가지고 오는 장소도 모두 다르기 때문에, 시스템 자체에서는 마스터 버전 (최신 버전이 항상 어느 곳에 저장되어 있는가)의 개념이 없다.
 - ✓ 예, 아래 그림과 같이 Developer Terry는 Server A에서 코드를 내려 받아서, 데이터 베이스 관련 모듈을 개발하고 있고, Michelle은 Server B에서 UI 관련 모듈을 개발하고 있다. 각자는 개발을 진행하면서, 수시로 각자 Server A와 Server B에 Commit을 하고 있다고 가정하자. Server A는 전체 시스템에서 데이터 베이스 모듈 부분은 가장 최신이고, Server B에는 UI 모듈의 가장 최신 버전의 코드가 들어가 있을 것이다.
- 각 모듈의 개발이 끝나면, Server A와 Server B의 코드를 merge하여, 개발 내용을 병합 가능
- 전체 시스템의 최신 소스 코드가 명시적으로 어느 한곳에 저장되어 있지 않는 구조

분산 형상 관리 시스템 Git

- 장점

- 팀 단위나 기능 단위로 저장소를 분리해서 개발하거나 릴리즈 버전 단위로 저장소를 분리해서 개발할 수 있는 등 소스코드 버전관리에 많은 유연성
- 중앙 저장소의 개념이 없기 때문에, 특정 VCS 시스템이 장애가 나더라도, 내가 사용하고 있는 VCS만 문제가 없다면 개발을 지속 가능
- 개발자의 Local PC에 VCS를 설치하여 네트워크 연결이 없는 상태에도 개발 가능
- 소스코드가 중앙 서버만이 아니라 여러 서버와 PC에 분산되어 저장되기에 서버 장애로 저장소가 손상된다 해도, 다른 서버나 다른PC에서 소스코드와 History들을 모두 저장하고 있기 때문에, 중앙 서버 방식에 비해서 복구 간편



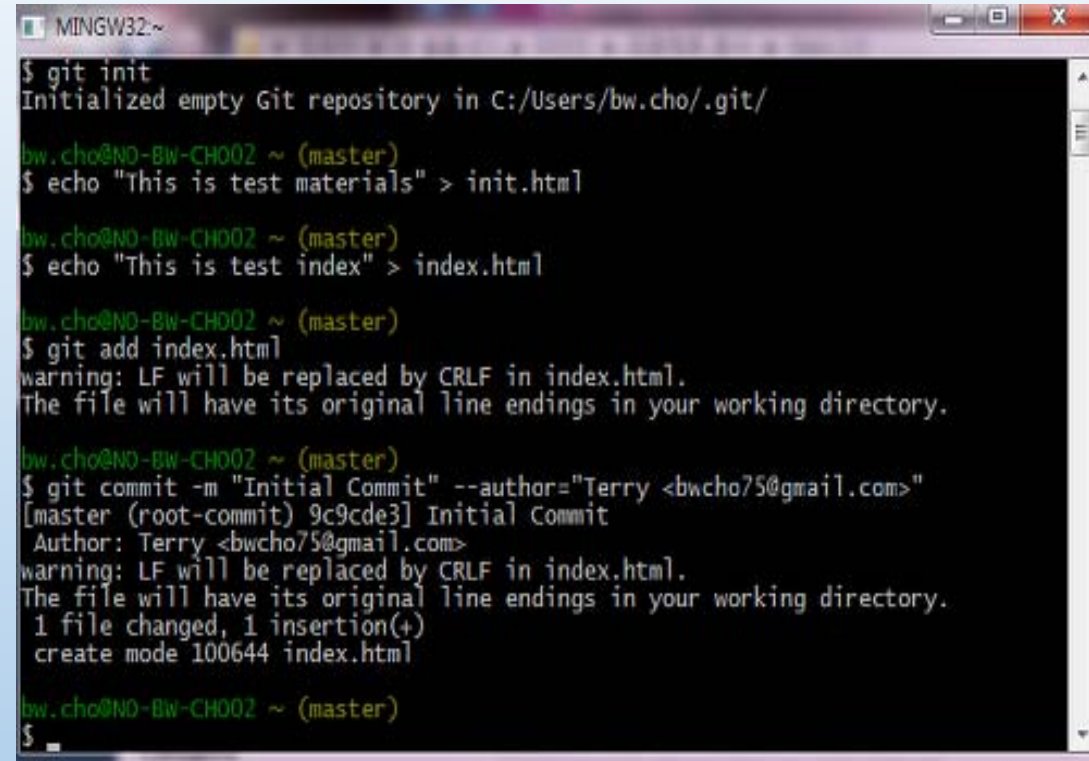
Git

- DVCS: Mercurial, Bazaar, git
- Git를 사용하는 대표적인 프로젝트
 - Linux Kernel 프로젝트나,
 - 안드로이드,
 - Gnome,
 - Ruby on Rails 등
- Git는 기존의 VCS에 비해
 - 설치가 쉽고, 속도가 매우 빠름
 - branch와 merge가 매우 빠르고 편리
 - 특히 merge의 경우 누가? 언제? 무엇을 어떤 부분을 merge했는지 까지 상세 추적이 가능하여, 오픈 소스와 같이 대규모 개발자가 동시에 개발을 진행하는 환경에서 매우 유용
 - 오픈소스 개발자들이 network가 연결되어 있지 않은 상황에서도 언제 어디서나 개발 가능
 - 자기만의 개발branch로 개발을 하다가 main branch로 쉽게 merge

git 기본 사용법

1) Repository 생성 (init과 clone)

- init 명령은 local에 새롭게 저장소를 만드는 명령
- 원격에 있는 서버의 저장소를 복제해서 로컬에 만들려면
git clone 사용자명@서버주소:"저장소 경로"
명령어를 사용



```
$ git init
Initialized empty Git repository in C:/Users/bw.cho/.git/

bw.cho@N0-BW-CH002 ~ (master)
$ echo "This is test materials" > init.html

bw.cho@N0-BW-CH002 ~ (master)
$ echo "This is test index" > index.html

bw.cho@N0-BW-CH002 ~ (master)
$ git add index.html
warning: LF will be replaced by CRLF in index.html.
The file will have its original line endings in your working directory.

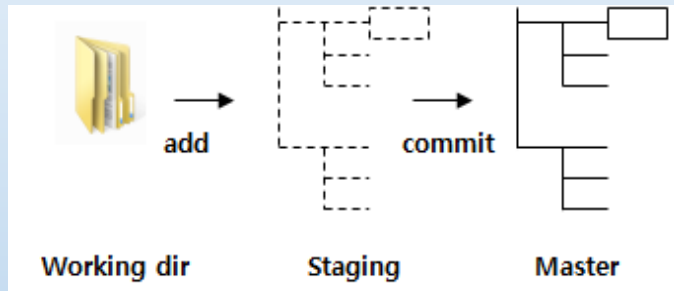
bw.cho@N0-BW-CH002 ~ (master)
$ git commit -m "Initial Commit" --author="Terry <bwcho75@gmail.com>"
[master (root-commit) 9c9cde3] Initial Commit
Author: Terry <bwcho75@gmail.com>
warning: LF will be replaced by CRLF in index.html.
The file will have its original line endings in your working directory.
1 file changed, 1 insertion(+)
create mode 100644 index.html

bw.cho@N0-BW-CH002 ~ (master)
$
```

git 기본 사용법

2) 파일 추가

- 앞의 그림에서 echo 명령을 사용하여, init.html과, index.html 파일을 생성하고 git에 이 파일이 git에 commit이 될 예정이라고 mark 설정



- 작업 디렉토리 (working dir)에서 작업을 한 것은 내 local pc에만 반영된 내용
 - 이 내용을 저장소로 올리기 전에, git는staging (git에서는 index라는 이름으로 사용) 이라는 개념을 제공
 - 소스 코드를 저장소에 최종 반영하기 전에 두 단계를 거치는 two-phase commit을 사용
- staging이라는 개념은 작업 디렉토리에서 작업한 내용을 반영하기 전에, 최종으로 확인하는 중간 단계
 - 작업 디렉토리에서 작업한 내용 중 commit할 내용을 미리 "add" 명령어를 통해서 stage에 반영한 후에, stage에 있는 내용을 commit전에, 저장소내의 코드와 비교(diff)하면서, commit
- 작업 영역과 (working directory)와 stage영역간의 비교는 "*git diff*" 명령어로 가능
- stage와 master 버전에 저장된 코드의 변경 사항은"*git diff --cached*" (또는 *git diff --staged*) 명령으로 비교가 가능
- 작업 디렉토리에서 수정 내용을 전부 한번에 commit하는 것이 아니라, featur별이나 특정 그룹 (기능이나 FIX별 또는 모듈별) staging에 이동 시킨 후, 하나하나 검증하면서 그룹별로 commit이 가능

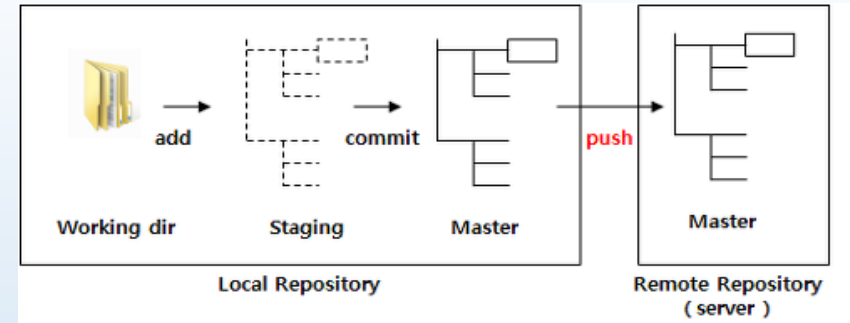
git 기본 사용법

3) 저장소에 반영 (commit)

- add를 통해서, 변경본 반영 리스트를 작성하고, diff등을 통해서 확인이 끝나면, 이를 저장소에 반영
- 반영은commit 명령어를 사용하며, commit에 대한 comment (변경 내용)을 인자로 제공
git commit -m "변경 내용 description"
- *git status*
 - 파일의 상태 확인 명령

git 기본 사용법

4) 변경 내용을 원격 저장소에 반영 (push)



- 앞 단계 까지로, 소스코드의 변경 내용은 내 local pc에 있는 git 로컬 저장소에 반영 완료
- git는 앞서도 설명했지만 분산저장소이기 때문에, commit을 한다고 해서, 서버에 코드가 저장 불가
- 서버에 반영은 push 명령어를 사용

git push origin "브랜치명"

예) git push origin master (master branch로 push하는 명령)

※단 처음 저장소를 만들 때, git clone을 통해서 원격 저장소로부터 코드를 읽어서 로컬 저장소를 만들었을 경우임

- 원격 저장소로부터 clone을 해서 만든 경우가 아닐 때,
 - 원격 저장소로 code를 저장하고자 한다면, 원격 저장소 정의 필요
 - 원격 저장소 정의 방법

git remote add "원격저장소명" "원격저장소주소"

예) git remote add zipkin <https://github.com/twitter/zipkin.git>

zipkin이라는 이름으로 <https://github.com/twitter/zipkin.git> URL에 있는 원격 저장소 등록

여기에 push를 하려면, git push zipkin master (zipkin 원격 저장소 master branch에 push)

- 원격 저장소는 하나가 아닌 여러 개를 git remote add 명령으로 추가 가능
- 등록되어 있는 원격 저장소는 git remote -v 명령으로 조회 가능

git 기본 사용법

5) 브랜치 관리

- 현재 코드에서 브랜치를 생성 - *git branch "브랜치명"*
예) bugFix라는 이름으로, 현재 코드에서 branch 만들기 - *git branch bugFix*
- 현재 작업중인 브랜치를 이동 - *git checkout "브랜치명"*
예) master 브랜치로 이동 - *git checkout master*

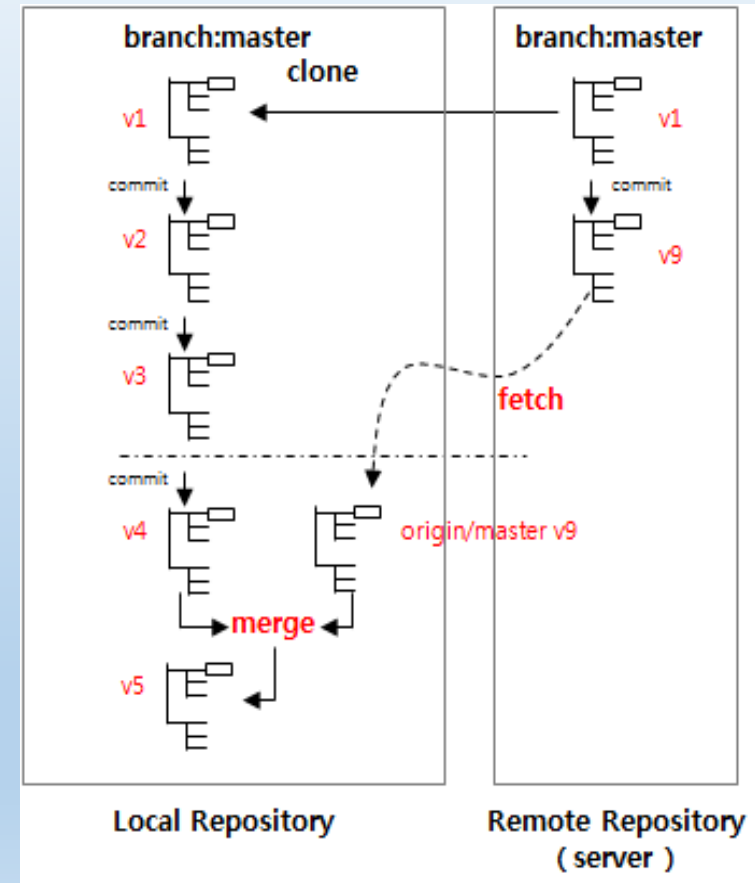
6) merge

- 다른 브랜치의 내용을 현재 작업 중인 브랜치로 합쳐 오는 작업
- 예, 내가 master 브랜치에서 작업을 하고 있을 때, 예전에 버그 수정을 위해 만들었던 bugFix라는 브랜치의 내용을 현재 브랜치에 반영하고 싶을 경우, master 브랜치에서 "*git merge bugFix*" 라는 명령을 사용하면 bugFix 브랜치의 변경 내용을 master 브랜치에 반영
- 서로 변경 내용이 다르거나, 같은 코드 라인을 수정하였을 경우 충돌 (conflict)이 발생
 - 직접 충돌 부분을 수정한 후, *git add*를 통해서 수정한 파일을 넣고, *git commit*을 통해서 최종 반영

git 기본 사용법

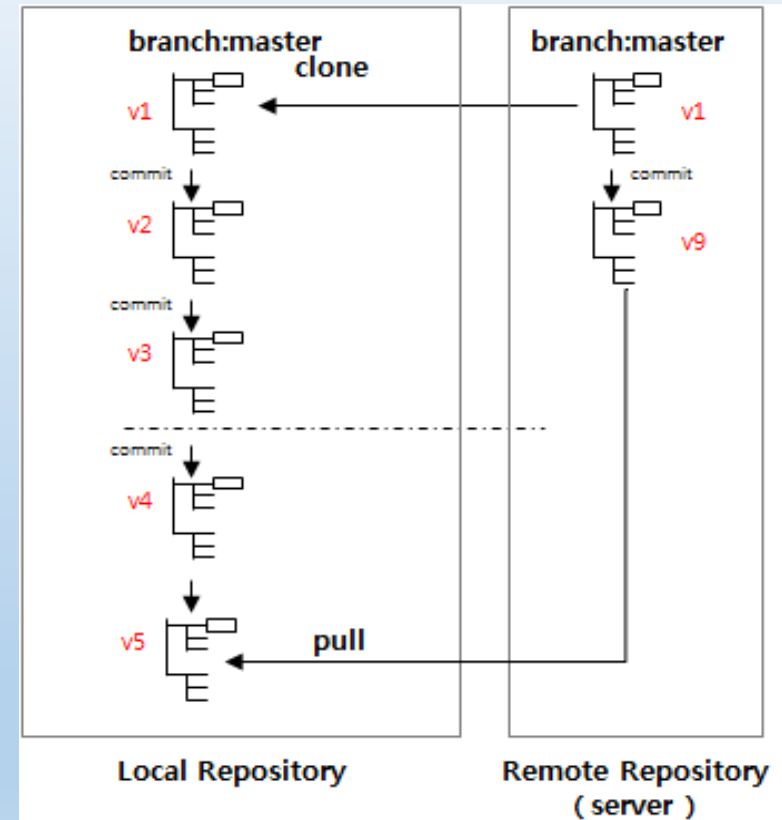
7) 원격 저장소의 변경 내용을 읽어오기 (pull & fetch)

- 원격 저장소에서 다른 사람들이 작업했던 내용을 내 로컬 저장소로 가지고 오려면 pull과 fetch
- fetch ("git fetch")의 경우,
 - 원격 저장소의 업데이트 된 내용을 별도의 브랜치로 읽어 온다. 실제 내가 로컬에서 작업중인 로컬 저장소에는 반영되지 않는다. 반영을 하려면 원격 저장소에서 읽어온 내용을 merge를 통해서 내 작업 영역에 반영
 - 원격 저장소의 v1 버전에서 clone을 받아서 로컬 저장소에서 개발을 시작하고 로컬에서 여러 번의 commit을 통해서, v4버전까지 개발을 진행
 - 원격 저장소의 변경 내용을 업데이트 하기 위해서 fetch를 하면, 원래 clone을 하였던 원격 저장소의 브랜치(master)의 최신 코드를 로컬로 복사해서 origin/master라는 이름의 브랜치에 업데이트를 한다. 내 작업 영역은 여전히 v4이고, 원격 저장소의 변경 내용은 반영되지 않았다.
 - 이를 반영하려면 "git merge origin/master"를 해주면 merge를 통해서 내 작업 영역에 반영된다. (v5 버전상태)



git 기본 사용법

- pull ("git pull")
 - 명령어 하나로 자동으로 fetch와 merge를 한번에 수행
 - 그림에서
 - 처음에 원격 저장소에서 v1 버전을 clone으로 내려 받아서 개발을 하다가, v4 버전에서 원격 저장소의 코드를 pull 해주면, 원격 저장소의 clone을 한 원본저장소와, 로컬의 저장소의 현재 브랜치를 merge하여 새로운 버전으로 작성



git 기본 사용법

8) 태깅

git tag -a "태그명" -m "태그 설명"

예) *git tag -a "build1109" -m "July.15 빌드 태그"*

- 태깅한 버전으로 이동하려면, 브랜치 이동과 마찬가지로 checkout을 사용

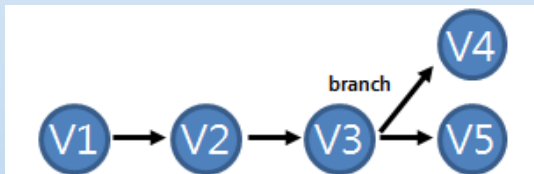
git checkout "태그명"

- 태그 버전으로부터 어떤 작업하려면, 해당 태그로 이동 후에, git branch를 이용하여, 그 버전에서부터 branch를 분리하여 작업을 하거나 clone등을 해서 작업
- 태그를 다른 사람과 공유하려면, 브랜치나 코드와 마찬가지로, 태그를 서버로 push
git push origin "태그명"

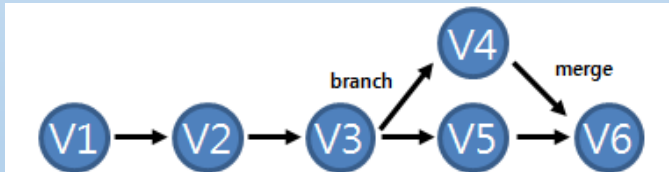
git 기본 사용법

9) Rebase

- rebase는 git만의 고유 기능
- merge와 유사한 기능이지만, 코드 변경 History를 조금 더 정리해주는 기능
- 예,
 - V3에서 브랜치를 생성하여 V4를 만들었고, master에서는 commit을 진행하여 V5로 진행되었다고 하자. 이 상태에서 V4의 변경 내용을 V5로 합치고자 한다



- 일반적인 경우 - merge를 하는 경우에는 V6 버전이 새로 생기고, V4 branch와 내용과 history가 모두 유지



- rebase 명령을 수행할 경우 - merge와 마찬가지로, 코드를 master branch로 합치지만, 기존의 V4 branch는 없어지고, V4에 작업된 내용 (변경 History)는 마치 master branch의 변경 history인 것 처럼 하나로 병합

