8/8/2011

|  |
| --- |
| 강은경, 한태희 |



|  |  |
| --- | --- |
| sevencore | GIT 사용법 |

**목 차**

**1.** **GIT 소개** 2

***1)*** ***GIT란?*** 2

***2)*** ***GIT의 역사*** 2

***3)*** ***GIT 개발 목표*** 2

***4)*** ***GIT을 버전관리로 채택, 사용하는 Project*** 2

***5)*** ***GIT 용어정리 및 기능*** 2

***6)*** ***CVS, Subversion과의 비교 및 Git의 특징*** 2

***2.*** ***Git 설치하기*** 2

***3.*** ***Git 환경설정*** 2

***4.*** ***Git 저장소 만들기*** 2

***5.*** ***Git 저장소 관리목록 추가 및 등록하기*** 2

***6.*** ***Git 이력확인 및 차이점 살펴보기*** 2

***7.*** ***브랜치 / 태그 사용하기*** 2

***8.*** ***브랜치 간의 변경사항 합치기*** 2

***1)*** ***바로 합치기*** 2

***2)*** ***커밋 합치기*** 2

***3)*** ***선택하여 합치기*** 2

***4)*** ***충돌 다루기*** 2

***9.*** ***이력 고쳐쓰기*** 2

***10.*** ***원격 저장소 이용하기*** 2

***11.*** ***SVN에서 Git로 옮겨가기*** 2

***1)*** ***SVN과 Git 명령어 비교*** 2

***2)*** ***SVN 서버 이용하기*** 2

***12.*** ***github 이용하기*** 2

***부록. Git 명령어 요약*** 2

***A.1 설정과 초기화*** 2

***A.2 기본적인 사용법*** 2

***A.3 브랜치*** 2

***A.4 이력*** 2

***A.5 원격 저장소*** 2

***A.6 Git와 SVN 연결하기*** 2

1. **GIT 소개**
2. ***GIT란?***

Linux소스를 관리하기 위해 Linus Torvalds가 개발한 소스 버전 관리 프로그램이다.

무료이자 오픈 소스이며 분산 버전 관리 시스템(DVCS)이다. 속도와 효율이 우수하고 작은 프로젝트부터 큰 프로젝트까지 다양하게 사용할 수 있도록 디자인 되었다.

Git Clone들은 full-fledged repository. 모든 history, revision추적 기능, 중앙 서버로부터 서로 의존적이지 않다.

Git는 Mercurial, Bazaar, Subversion, CVS, Perforce, Visual SourceSafe 툴과 같이 파일에 대한 버전 관리에 사용된다.

현재 GIT 프로그램은 Windows, Mac OSX, 기타 Linux 시스템을 위하여(Binary, Source) 로 제공이 된다.

Windows, Mac OSX 용 GIT 프로그램은 Google 에서 배포하고 있다.

1. ***GIT의 역사***

1991~2002 : Linux 커널 유지보수는 소프트웨어의 패치와 압축된 형식으로 전달.

2002 : Linux 커널 프로젝트는 독점권이 있는 BitKeeper라는 분산버전관리 시스템(DVCS)을 사용.

2005 : Linux 커널 개발자 커뮤티니와 BitKeeper를 개발한 회사와의 관계가 나빠짐. Linux 개발자 커뮤니티(대표주자 Linus Torvalds)는 BitKeeper를 대체할 무료의 DVCS을 개발하기 시작.

1. ***GIT 개발 목표***

|  |  |  |
| --- | --- | --- |
| 1. 속도 |  |  |
| 1. 디자인의 간결함 | | |
| 1. 완전한 분산 시스템 | | |
| 1. Non-linear하지 않은 개발에 대한 강력한 지원 (수천 개의 branch들을 지원할 수 있도록) | | |
| 1. Linux 커널과 같은 매우 큰 프로젝트를 효율적으로 관리할 수 있는 시스템 | | |

1. ***GIT을 버전관리로 채택, 사용하는 Project***

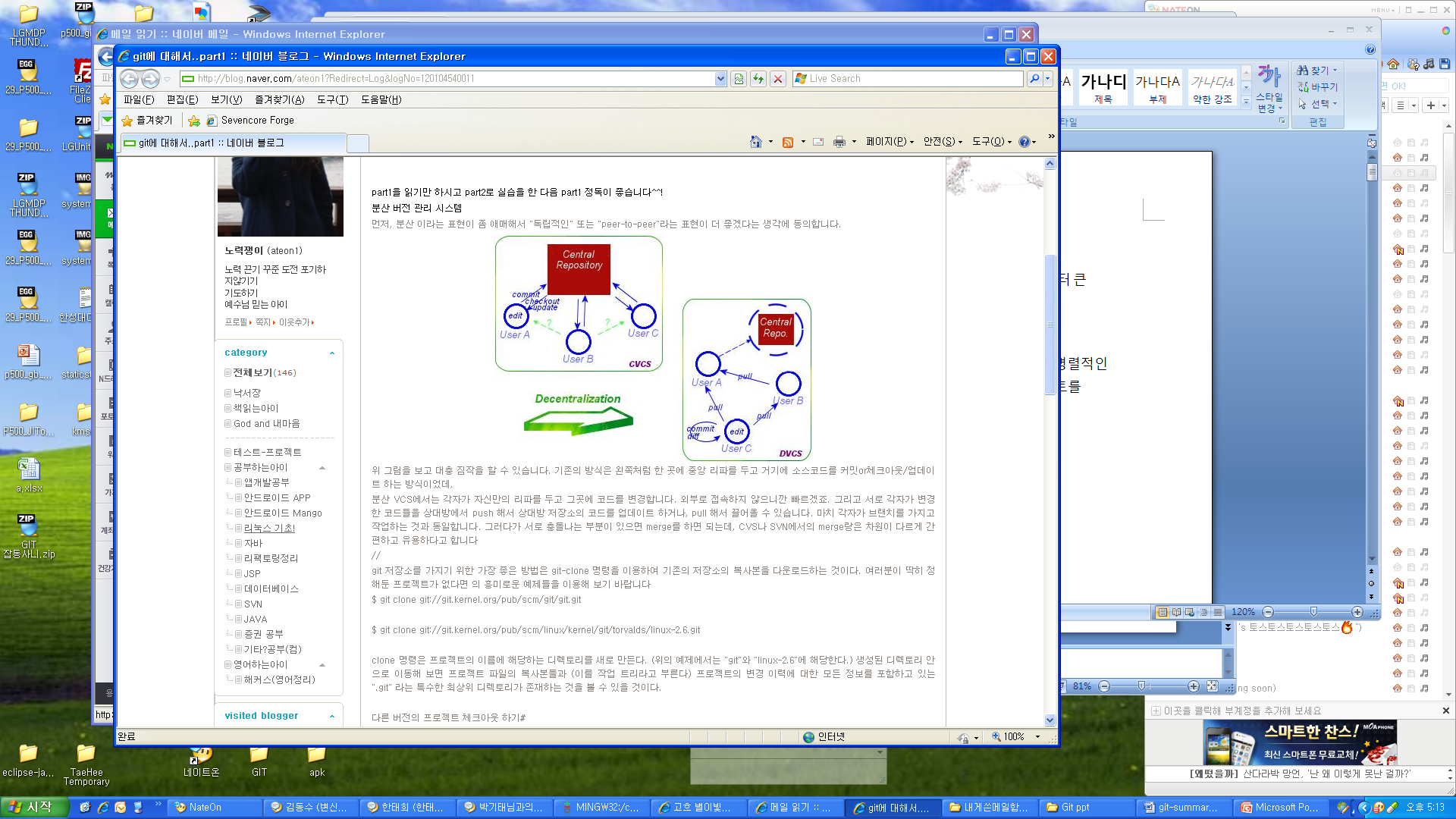
|  |  |  |  |
| --- | --- | --- | --- |
| 1. GIT | 1. Linux Kernel | 1. Perl | 1. Eclipse |
| 1. Gnome | 1. KDE | 1. Qt | 1. Ruby on Rails |
| 1. Android | 1. Debian | 1. X.org |  |

1. ***GIT 용어정리 및 기능***

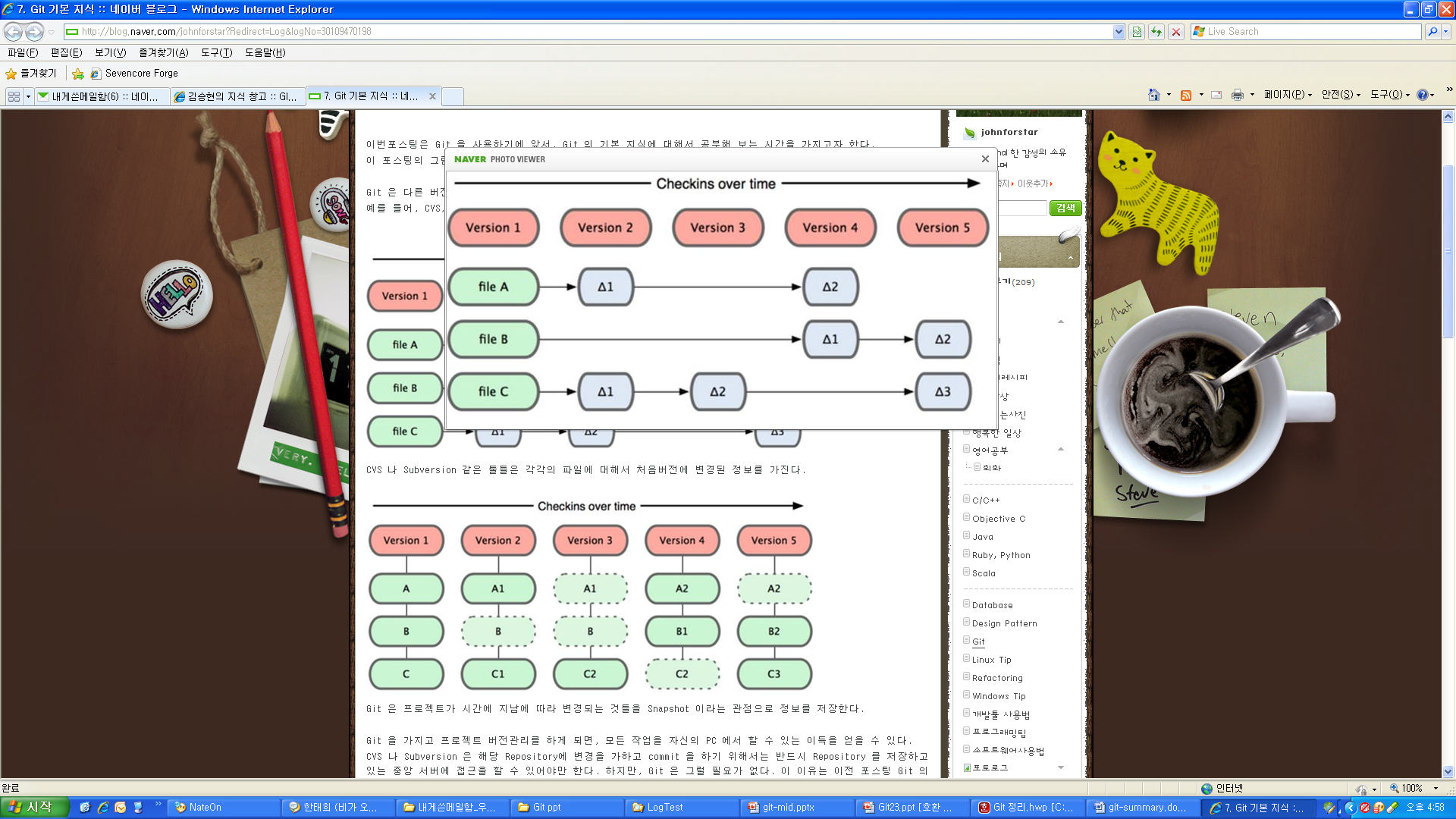
|  |  |
| --- | --- |
| 1. *저장소*   *(Repository)* : | 사용자가 변경한 모든 내용을 추적하는 공간이다.  대부분의 VCS는 코드의 현재 상태는 물론이고 변경이 언제 발생했는지, 누가 변경했는지 변경사항을 설명하는 텍스트 로그메시지까지 저장한다. |
| 1. *작업 트리*   *(Working Tree)* : | 저장소를 어느 한 시점을 바라보는 작업자의 현재 시점이다. |
| 1. *체크아웃*   *(Checkout)* : | 작업자의 작업 트리를 저장소의 특정 시점과 일치하도록 변경하는 작업이다. |
| 1. *스테이징 영역*   *(Staging area)* : | 저장소에 commit하기 전에 commit을 준비하는 위치. index라고도 불린다.  변경사항을 적용하기 전에 한번 더 변경사항을 정리하고 다듬을 수 있는 기회를 제공한다. 변경사항을 추가하기 위해서는 git add 를 사용한다. commit예정인 변경사항이 있다고 보면 된다. |
|  | |
| 1. *master* : | master branch는 복사해온 저장소 내의 HEAD의 복사본이다. |
| 1. *브랜치*   *(Branch)* : | 하나의 개발라인을 의미한다. 가장 기본이 되는 master branch에서 버그 수정이나 특정 기능을 추가하기 위해서 개발라인을 따로 두고 작업할 수 있다. 한 개의 프로젝트에서도 여러 개의 개발라인이 존재할 수 있다.  하나의 git 저장소는 여러 branch의 개발과정을 관리할 수 있다. 이는 한 개의 branch내에서 가장 최근에 commit된 reference인 HEAD(branch head)의 목록을 관리하는 방식으로 수행된다.  (Ex. branch apple에 3개의 commit이 있는데 이 중 가장 최근에 추가된 commit이 HEAD.)  Branch를 생성하면 파일이 분기하는 위치가 repository에 기록된다.  branch는 다른 branch와 분리하여 내용에 대한 변경사항을 지속적으로 추적한다. 합쳐지거나 여러 개로 나뉠 수 있다. |
|  | |
| 1. *Origin* : | 단지 GIT가 복사해 온 저장소를 가리키기 위해 기본적으로 사용하는 이름이다. |
| 1. *Pushing* : | 개발자가 직접 변경사항을 중앙 저장소에 전송하려고 접속하는 것이다. |
| 1. *Pulling* : | 변경사항을 이용하기 위해 GIT 저장소에 변경사항을 가져온 후, 변경이력을 지역이력과 합치는 것이다. Subversion의 update와 유사하다. |

1. ***CVS, Subversion과의 비교 및 Git의 특징***

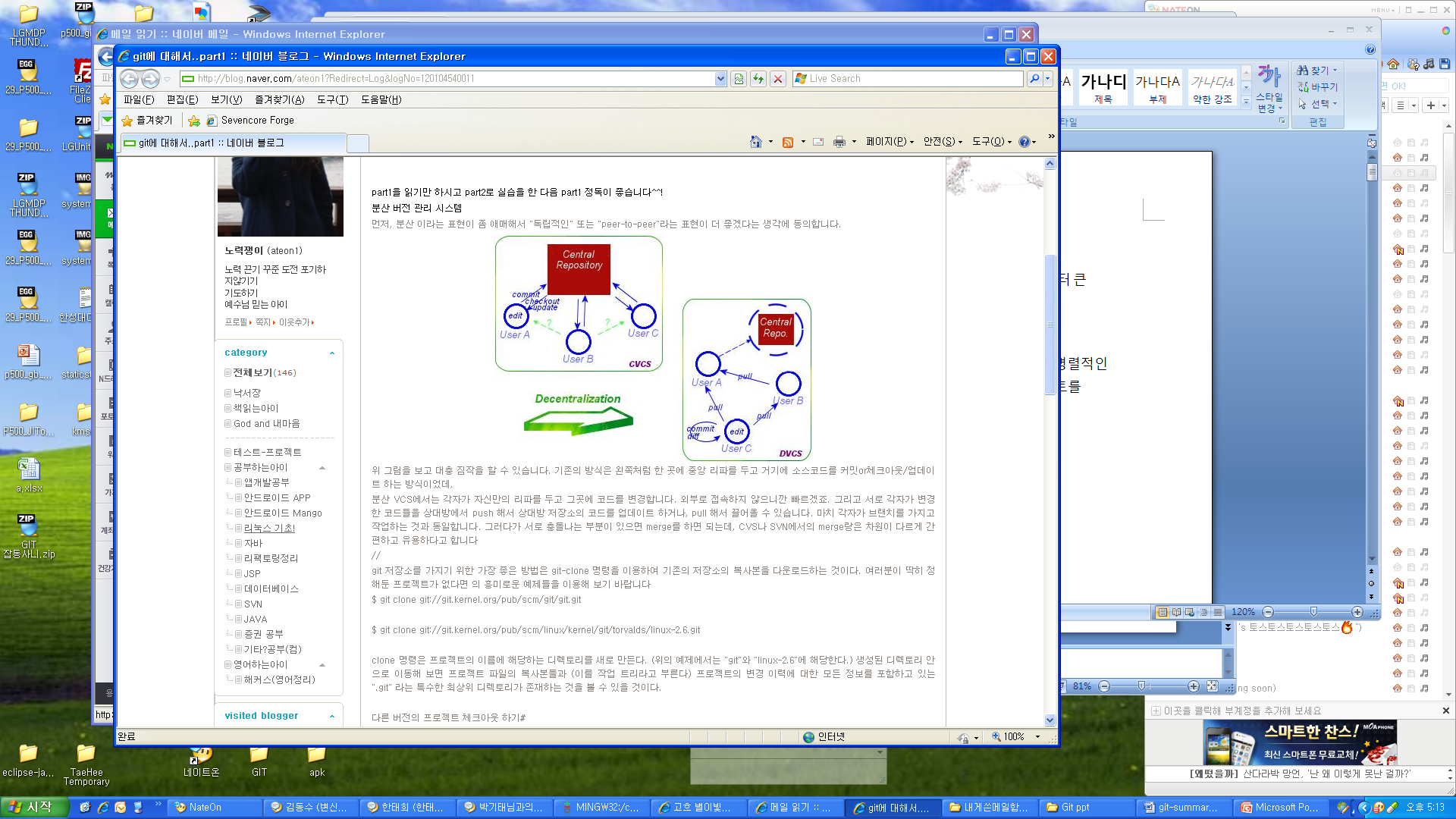
* ***CVS, Subversion 비교***



|  |
| --- |
| **CVS, Subversion** |
| * **Centralized** * Network너머 **SVN Server에 저장소 존재.** * **모질라의 경우**   : 12GB   * 개발자들은 원격에서 작업한 후 네트워크 연결을 통해 저장소에 변경 사항을 전송. * **중앙 집중식 저장모델**   : 하나의 중앙 저장소가 있고, 모두가 이 중앙 저장소에 변경 사항을 전송. 각 개발자는 저장소의 최신버전을 복사해서 가지고 있고, 이 복사본을 변경한 후에는 다시 중앙 저장소에 전송.   * **한계**   : 사용자는 최신버전의 코드만 가짐. 변경이력을 보려면 저장소에 정보를 요청.(즉, 네트워크 상에서 원격 저장소에 항상 접근 되어야 함) |



* ***GIT***



|  |
| --- |
| **GIT** |
| * **분산 버전 관리 시스템 (DVCS)** * File들의 변경이력을 Snapshot에 압축하여 모아두는 방식.   프로젝트의 Snapshot을 찍어 데이터를 저장한다.  소스코드를 commit할 때 또는 프로젝트의 현재 상태를 저장하고자 할 때, 기본적으로 그 시점의 모든 File상태에 대한 Snapshot을 찍어 그 Snapshot에 대한 정보를 저장한다.     * 효율성을 위해 변경되지 않은 파일은 새로운 Snapshot에 저장하지 않고, 기존의 파일에 대한 링크만 저장한다. (현재의 Snapshot에 링크만 함.) * GIT은 **완전한 분산환경**이므로 pushing하고 pulling하는 repository가 여러 개 일 수 있다. * 작업 트리를 처음 가져오기 위해 자신의 프로젝트를 저장소에 초기화하도록 요청하거나 기존 저장소의 프로젝트를 복제.      * 변경사항을 commit하면서 저장소에 새로운 revision추가, 무엇을 변경했는지 log메시지 저장. * 상위 저장소에 변경 사항을 pushing하면 바뀐 내용을 다른 개발자와 공유 가능. * 변경사항을 가져와 이용하려면  1. 변경사항을 가져온다. (remote repository에 있는 변경사항의 복사본이 생성된다. Pushing과 반대개념으로 변경사항을 저장소에 전송하는 것이 아니라, remote repository가 가지고 있는 변경사항을 요청.) 2. 가져온 변경사항을 지역 이력과 합친다. 보통 변경사항을 가져오고(fetch) 합치는 작업(merge)을 동시에 진행하기 때문에 이 두 가지 작업을 한번에 하도록 풀링(Pulling) 이라는 기능을 지원한다. (Pulling -- Subversion의 update와 유사. Pulling = fetch+ merge)  * GIT은 완전한 분산환경이므로 Pushing하고 Pulling하는 repository가 여러 개 일 수 있다.   **특징을 크게 5가지로 나누면**   1. **Distribute development** 2. **Strong support for non-linear development** 3. **Efficient handling of large projects** 4. **Cryptographic authentication of history** 5. **Toolkit design** |

* 정리하면

|  |  |
| --- | --- |
| **GIT** | **SVN** |
| Distribute | Centralized |
| Repository : local에 있다. | Repository : network너머 svn server에 있다. |
| * GIT의 performance가 더 압도적이다. | |
| 모질라의 경우 : 420MB | 모질라의 경우 : 12GB |
| * GIT이 더 작은 file space를 차지한다. | |
| GIT가 브랜치 컨트롤이 더 좋다. | |
| SVN의 user interface가 강력하다. | |

* ***Distributed***
* Linus’ Vision of Git



* ***GIT 특징 5가지***

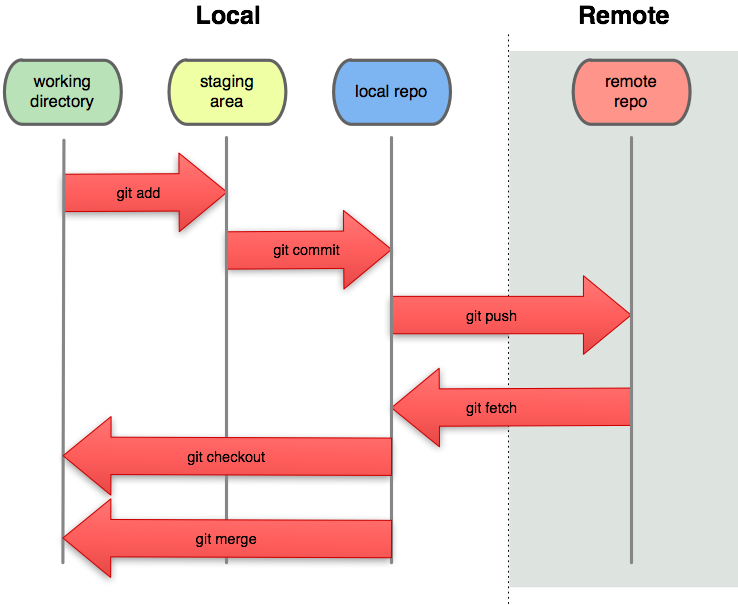
1. **Distributed development**

각자가 프로젝트의 전체 이력이 있는 자신만의 저장소를 가진다. (local에 저장소를 가진다.)

Commit 할 때는 원격 저장소에 연결할 필요가 없이 변경사항을 지역 저장소에 기록한다. (프로젝트 변경 이력을 저장하기 위해 네트워크에 접근할 필요가 거의 없다.)

따라서 모든 작업을 자신의 PC에서 할 수 있다.

중앙 저장소에 전송하는 것을 Pushing이라고 한다. 변경 사항을 모아둔 패치를 만들어 프로젝트 관리자에게 제출할 수도 있다.



1. **Strong support for non-linear development**

빠르고 편한 branch와 merge기능을 지원한다.

Non-linear한 개발 이력을 시각화하고 탐색할 수 있는 강력한 틀(gitk)을 제공한다.

1. **Efficient handling of large projects**

매우 빠르고, 크고 history내역이 긴 프로젝트에 매우 잘 작동한다.

다른 오픈 소스 VCS보다 빠르며, 효율적인 revision 관리를 수행한다.

1. **Cryptographic authentication of history**

GIT history는 revision이란 특수한 이름으로 저장한다.

revision은 그 commit까지의 완전한 개발 history에 의존한다. 일단 만들어지면, 오래된 버전의 revision은 통지 없인 변경이 불가능하다. 또한 tag들은 암호화되어 저장된다.

1. **Toolkit design**

GIT은 C언어로 작성된 많은 Tool들의 집합체이다.

개발자가 사용하기 쉽도록, 또 새로운 기능을 수행할 수 있도록 script 작성을 쉽게 하기 위해 tool들을 제공한다.

* ***GIT의 강점***
* **분산 아키텍처**

: 완전히 연결이 끊어진 상태에서, 항상 인터넷에 연결 되어야 하는 고통 없이 동작한다.

* **쉬운 branch 만들기와 합치기**

: 다른 VCS와는 달리 branch를 여러 번 나눈 경우라도 간단히 합칠 수 있다.

* **서브 버전과 통신**

: GIT은 서브버전 repository의 모든 이력을 가져올 수 있으며, 변경사항을 다시 보낼 수 있다.

* ***GIT Workflow***
* Fetch or clone
* Modify the files in the local branch
* Stage the file
* Commit the files locally
* Push change to remote repository

1. ***Git 설치하기***
2. ***Windows***
   * msysgit 설치

☞ http://code.google.com/p/msysgit 에서 설치파일을 받아 실행

* + cygwin을 통한 설치

1. ***Linux***
   * 소스파일을 이용한 설치

☞ 의존적 라이브러리 설치

sudo apt-get build-dep git-core git-doc

☞ http://git-scm.com/download에서 소스파일을 가져온 후

tar -zxf git-1.7.6.tar.bz2

cd git-1.7.6

make prefix=/usr/local all doc

sudo make install install-doc

* + 배포본 설치

☞ apt-get install git-core

1. ***MacIntosh***
   * GUI방식 Gitinstaller 사용

☞ http://code.google.com/p/git-osx-installer 에서 다운로드

* + MacPorts이용

☞ sudo port install git-core +svn +doc명령어로 설치

1. ***Git 환경설정***

Git를 이용하려면 몇 가지 정보가 필요 분산 환경이라는 특성 때문에 사용자 이름과 이메일 주소를 제공하는 중앙 저장소가 없으므로 config명령을 통해 이름과 이메일 주소를 설정한다.

전역값으로 설정하려면 --global 옵션을 이용한다.

특정 폴더에서만 이용하고 싶을 경우 --global 옵션을 빼고 입력한다.

이외에도 color옵션 등 130개 이상의 값들을 설정할 수 있으나 기본적인 옵션은 위 두 가지면 된다.

- git config --global user.name “USER\_NAME”

- git config --global user.email “USER\_EMAIL”

- git config --global --list

칼라옵션은 기본 값으로는 꺼져 있는데 아래와 같이 auto를 입력하면 칼라 옵션이 활성화 된다. - git config --global color.ui “auto”

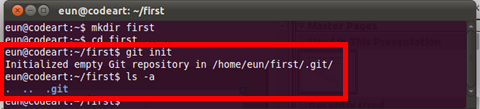


1. ***Git 저장소 만들기***

- git init : 저장소 생성

- 초기 브랜치 : master branch

- .git 폴더 생성



Git 에서 저장소를 생성하는 일은 간단하다. 이는 원격 저장소가 아닌 자신만의 지역 저장소로서 원하는 폴더에 가서 git init 명령을 입력한다면, 저장소가 생성이 된다.

이 초기화 과정은 .git 디렉터리를 생성하고 여기에 저장소의 메타데이터를 모두 저장한다.

그리고 현재 비어있는 first디렉터리는 앞으로 코드를 작업할 작업트리(Working directory)가 된다.

1. ***Git 저장소 관리목록 추가 및 등록하기***

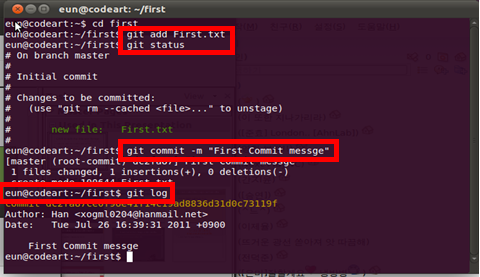
- git add FILE\_NAME : Git Staging Area에 등록

- git status : 현재 상태 확인

- git commit -m “LOG\_MESSAGE” : 저장소에 저장

(이 때 commit으로 저장된 저장소는 자신의 PC 저장소이다.)

- git log : 지금까지 commit한 Log 확인



* ***git status 명령어***

git 저장소에 파일을 새로 추가하거나 변경하면 Working Directory가 변경되었음을 감지한다.

이 때 git status를 확인해보면 빨간색으로 변경사항이 있다고 표시된다.

* ***git add 명령어***

git add를 이용해 작업트리에 변경사항을 알리면 Staging area에 등록되고, git status를 입력하면 초록색으로 변경사항을 알려준다.

git add -i를 입력하면 대화형 add를 할 수있다.

* ***git commit 명령어***

커밋은 저장소에 저장된 개별적인 이력으로, 각 커밋은 코드의 진행 상태를 기록한다.

아무 옵션 없이 git Commit을 이용 시 Staging area의 변경 값들만 저장소에 올린다.

-m 옵션을 쓰지 않으면 에디터 창이 열려서 커밋할 로그 내용을 적을 수 있으며, -m 매개변수를 이용한다면 command창에서 직접 로그내용을 적을 수 있다. -m명령어는 여러번 중복해서 여러줄을 입력할 수도 있다.

git commit -a 명령어를 이용 시 Working Directory의 변경 사항 전부를 저장소에 올려준다.

따라서 –a 옵션을 이용하지 않을 경우, Staging area는 변경사항을 저장하기 전 마지막 다듬는 작업을 가능하게 한다.

1. ***Git 이력확인 및 차이점 살펴보기***

- git diff : Staging, Commit 되지 않은 Working directory변경사항과의 차이 출력

- git diff --cached : Staging 된 변경사항과의 차이 출력

- git diff HEAD : 현재 HEAD 이후의 모든 변경사항 출력

- git diff COMMIT\_NAME ANOTHER\_COMMIT : Commit 간의 차이점 출력

- git diff --stat COMMIT\_NAME ANOTHER\_COMMIT: Commit간의 차이점 통계화

- git log 18f822e.. : 해당 Commit 부터 HEAD까지의 차이점 출력

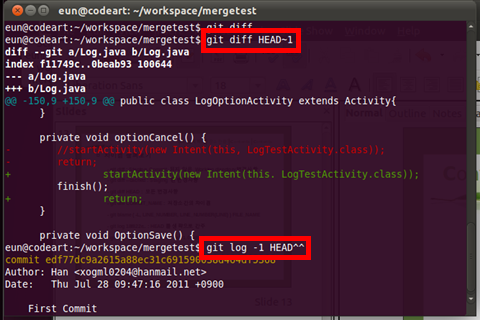
- git log --pretty=format:”%h %s” 1.0..HEAD

- git log -1 HEAD^^^

- git log -1 HEAD~3

- git log --since=”5 hours”

- git blame ( -L, LINE\_NUMBER, LINE\_NUMBER(LINE) ) FILE\_NAME



* ***git diff 명령어***

git diff 명령어는 작업트리의 변경사항, 스테이징돼서 커밋하려는 변경사항, Commit간의 차이점을 보여줄 수 있다.

매개변수없이 실행하면 스테이징되지 않은 작업트리와 스테이징 영역을 비교한다.

--cached 명령을 추가하면 스테이징 영역과 저장소의 차이점을 확인한다.

작업트리, 스테이징된 변경사항, 저장소의 모든 차이점을 비교하고 싶다면 HEAD를 추가

해당 커밋과의 차이점을 보고 싶다면 커밋명을 쓴다.

두번째 인자를 주지 않는다면 HEAD와 비교한다.

--stat를 이용하면 저장소간의 차이를 통계화하여 보여준다.

* ***git log 명령어***

git log명령어는 저장소에 어떤 변화가 있었는지 알려고 할때 일반적으로 확인하는 내용이다. 커밋할 때마다 로그를 남기기 때문에 자신이 무엇 때문에 커밋했는지 알 수 있다. log명령어는 다양한 포맷과 범위를 지원하기 때문에 쉽게 로그를 찾을 수 있도록 해준다.

일반적으로 git log명령어를 입력하면 (커밋명, 변경자, 날짜, 로그내용) 이 4가지를 각 줄마다 표시해 주게 된다.

위 로그메시지는 블로그의 글 목록과 마찬가지로 최신 커밋이 가장 먼저 나타난다.

--pretty=format=oneline : 이 옵션을 주게 된다면 로그만 한줄로 표시된다.

--pretty=format:”%h %s” 옵션을 추가하면 짧은 해시와 커밋 로그의 첫번째 줄을 보여준다.

-N 옵션을 준다면 사용자가 입력한 N의 개수만큼만 최신순부터 출력하게 된다.

그 뒤에 HEAD나 커밋명이 올 수 있는데, ^나 ~N을 이용하여 해당 커밋의 몇번째 뒤의 커밋을 지정할 수 있다. 예를 들어 HEAD~3은 HEAD로부터 2개 전의 커밋을 의미한다.

--since와 --before를 이용하여 “24hours”, “1minute”, “2008-01-01”등 영어식 표현을 모두 이해하므로 원하는 기간만큼의 로그를 확인할 수 있다.

* ***git blame 명령어***

마지막으로 ..을 이용하여 ..사이의 두 커밋 사이에 있는 커밋들을 볼 수도 있다.

git blame은 파일의 각 줄 앞에 커밋명, 커밋한 사람, 시간정보를 추가해서 보여준다.

-L옵션은 <시작> <끝>의 매개변수 하나씩을 받는다. 첫번째는 시작 줄의 번호 두번째는 끝 줄의 번호나 +N, -N으로 라인수를 지정할 수 있다.

1. ***브랜치 / 태그 사용하기***

- git branch NEW\_BRANCH ORIGINAL\_BRANCH

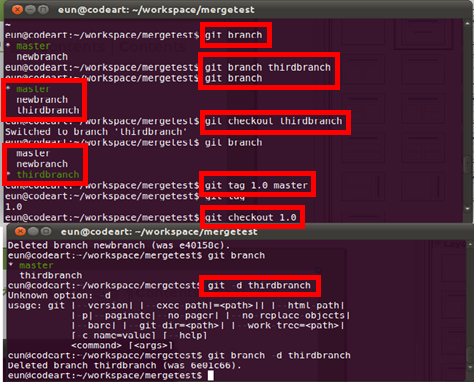
- git checkout BRANCH\_NAME

- git tag TAG\_NAME BRANCH\_NAME

- git branch

- git tag

- git checkout -b NEW\_BRANCH ORIGINAL\_BRANCH



실제로 유용한 형태의 브랜치는 두 가지가 있다. 프로젝트의 여러 버전을 브랜치 별로 관리하기 위해 생성한 브랜치와 특정 기능을 다루는 주제 브랜치다.

* ***git branch 명령어***

git branch 명령어를 이용하여 브랜치를 생성할 수 있다. git branch의 첫 번째 인자에 새로 만들 브랜치명을 적고 두 번째에 새로 만들 브랜치명을 적는다. 적지 않는다면 현재 위치에서 브랜치를 생성한다.

브랜치가 생성되었는지 확인하려면 git branch를 입력하면 현재 생성되어있는 branch의 목록을 보여준다.

기본적으로 릴리스 브랜치를 생성할 때는 앞에 RB\_를 붙여주는 것이 관례이다.

* ***git checkout 명령어***

git checkout으로 브랜치나 Tag를 옮겨갈 수 있다.

git checkout 브랜치명 또는 태그명을 쓰면 해당위치로 이동한다.

* ***git tag 명령어***

태그는 특정 마일스톤에 도달하였을 때 붙이는 것이 일반적이다.

Tag를 붙일 시 해당 Tag의 위치는 직접 수정이 불가능하여 일관성 유지에 도움을 준다.

Tag에서 수정하기 위해서는 브랜치를 새로 생성하여 수정 가능하다.

1. ***브랜치 간의 변경사항 합치기***
2. ***바로 합치기***

☞ git merge BRANCH\_NAME

1. ***커밋 합치기***

☞ git merge --squash BRANCH\_NAME

☞ git commit -m LOG\_COMMENT

1. ***선택하여 합치기***

☞ git cherry-pick COMMIT\_NUMBER

☞ git commit -m LOG\_COMMENT

각 브랜치는 저장소를 구성하는데 매우 중요한 요소이지만, 브랜치 간의 변경사항을 공유할 필요도 있다. 이런 경우 합치기를 이용한다.

바로 합치기는 하나의 브랜치를 선택해서 다른 브랜치와 합친다. 해당 브랜치의 전체 이력을 다른 브랜치에 반영하고자 할 때 사용한다.

현재 브랜치를 합치고자 하는 브랜치로 전환 한 뒤에 git merge 브랜치명

커밋 합치기는 새로운 기능이나 버그를 수정 한 뒤에 합칠 경우 마지막 최종본만 필요로 할 경우에 사용한다. 합친 후에 COMMIT을 해야 최종적으로 합쳐진다.

선택하여 합치기는 브랜치 간에 전체를 합치는 대신 하나의 커밋만 합칠 필요가 있을 때 이용한다. 브랜치에 아직 사용할 수 없는 기능이 있거나 준비가 안된 변경 사항이 있을 경우 주로 이용된다.

1. ***충돌 다루기***

- 커밋을 자동적으로 합칠 수 없는 경우

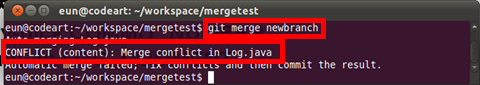
- 각 브랜치에서 동일한 영역을 다르게 변경할 경우가 빈번

- 합칠내용이 간단할 경우 직접 수정

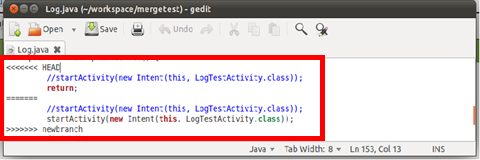
- git mergetool 이용.

- 수정 후 변경사항이 올라갈 수 있도록 commit을 해준다.

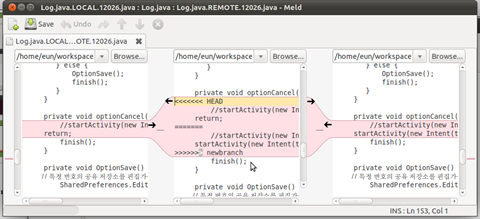
* CONFLICT가 발생하여 merge가 중단된 경우



* 충돌이 발생하면 해당파일에 직접 충돌이 있음을 알려준다. <<<HEAD는 해당 브랜치의 HEAD에서 수정한 내용이고, =====으로 분할된 아래부분은 >>>>>newbranch에서 수정한 내용이다. 이 두 개의 브랜치에서 충돌이 일어난 모습을 보여주고 있다.



* mergetool을 실행하면 위 편집기로 실행한 것과 비슷한 화면을 보여준다. Merge툴은 여러가지가 있는데 자신이 원하는 머지툴을 설치하고 이용하면 된다.



1. ***이력 고쳐쓰기***

- git commit -C HEAD -a --amend

- git revert -n HEAD

- git revert -n 540ecb7

- git reset --soft HEAD^

- git reset --hard 540ecb7~3

- git rebase -i HEAD~3

- git rebase -i 0bbdfb^

- git rebase TAG(BRANCH)

- git rebase --continue

- git rebase --skip

- git rebase --abort

이력을 변경하는 동작은 때로는 주의를 요한다. 푸싱을 하기 전 지역 저장소에서 이력을 변경하는 작업은 괜찮으나 이미 push한 이력을 변경하는 작업은 위험하다. 이미 다른 사람이 fetch한 경우 이력을 변경 할 경우 문제가 생길 수 있다.

* ***--amend 옵션***

git commit -C --amend 이용 시 원래 사용한 로그메시지 이용하여 변경을 한다.

소문자 -c를 이용할 시에는 로그메시지를 변경할 수 있도록 에디트 창이 뜨게 된다.

HEAD대신 다른 커밋명을 사용 가능하나 수정할 수 있는 것은 마지막으로 실행한 커밋에만 이용된다. 그 외에는 revert나 rebase명령어를 이용한다.

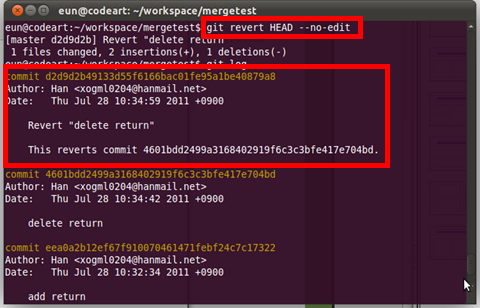


* ***git revert 명령어***

실수 이후 여러 번 커밋한 경우 git revert를 사용하여 변경할 수 있다.

-n 매개변수는 여러 번 되돌릴 때 유용하다 revert로 되돌리고 바로 커밋되지 않고 스테이징 영역에서 기다린다. 이 때 여러 번 revert를 한 뒤에 커밋을 할 수 있다. 주의할 점은 항상 역순으로 돌려놓아야 한다는 것이다.

--no-edit 이용 시 기본 메시지를 이용하여 커밋 한다. 되돌린 것은 전의 커밋들이 삭제되는게 아니라 되돌리며 커밋이 되는 것이다.

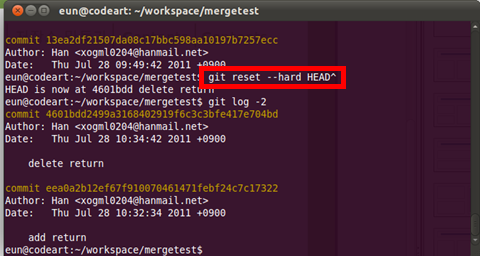


* ***git reset 명령어***

reset 명령은 사용자가 원하는 상태로 저장소를 재설정 할 수 있다.

--soft옵션은 이전 커밋을 모두 스테이징하지만 커밋하지는 않는다.

--hard옵션은 저장소와 작업 트리에서 커밋을 제거해버린다. 이는 커밋 삭제라고 생각하면 된다.

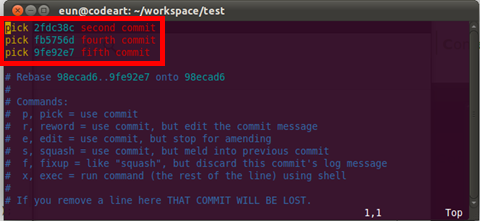


* ***git rebase 명령어***

rebase 명령어를 이용하여 순서를 바꾸거나 합치거나 나눌 수 있다.

rebase명령을 사용 할 때는 원하는 범위를 HEAD~3처럼 지정하면 된다. 그러면 HEAD를 포함하여 이전 3개를 수정 할 수 있게 된다. 명령어를 치면 에디트 창이 생성되는데, pick로 된 것의 순서를 바꾸면 commit의 순서가 바뀌게 되고 pick를 squash로 바꾸면 위의 pick와 합쳐지게 된다. Edit로 바꾸고 편집기를 종료하면 재정렬 작업이 시작되고 Edit을 만나면 stop표시가 뜨게 된다. 이 때 git reset HEAD^을 이용하여 전 단계로 간 뒤에 변경사항을 변경하고 commit을 두 번 해준뒤에 git rebase --continue를 하게 되면 커밋이 나누어지게 된다. 위 stop표시가 나왔을 때, git rebase --skip을 하게 되면 건너뛰고 재정렬을 하게되며 --abort를 이용하게 되면 재정렬이 중지된다.

- git rebase -i HEAD~3 을 실행한 화면



- squash를 이용하여 커밋을 합친 화면



1. ***원격 저장소 이용하기***

- git clone URL

- git pull ORIGIN\_REPO LOCAL\_REPO\_BRANCH

- git push ORIGIN\_REPO LOCAL\_REPO\_BRANCH

- git remote add NAME REPO\_URL

- git remote show

- git remote rm

git는 네트워크를 통해 원격 저장소와 통신한다. 사용하는 네트워크는 내부 LAN이나 VPN, 인터넷이 될 수있다.

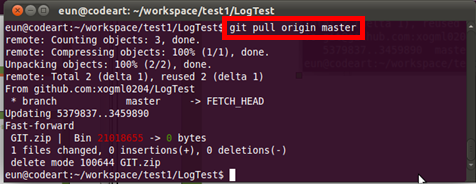
사용가능한 프로토콜은 3가지이다. SSH, git, HTTP/HTTPS가 사용된다. 3가지를 병행하여 사용할 수도 있다. 뒤에 소개할 Github는 git는 공개용 읽기전용, SSH는 읽기쓰기용으로 이용한다.

* ***git clone 명령어***

앞에서 보았듯이 git clone명령어는 원격 저장소에 있는 전부를 복사하의 나의 지역저장소에 넣는다.

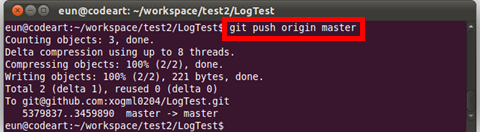
* ***git pull 명령어***

Git pull은 원격브랜치로 부터 최신 변경사항을 가져와서 지역 브랜치에 합친다.



* ***git push 명령어***

Git push는 원격저장소에 지역저장소의 내용을 추가하는 것이다. 만약 원격저장소에 변경된 사항이 있다면 git pull을 이용하여 최신사항으로 업데이트 한 후에 push를 해야 한다.

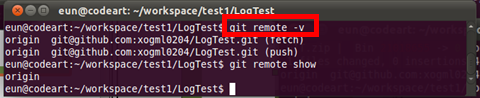


* ***git remote 명령어***

git remote 명령어는 원격 저장소의 주소를 별칭으로 저장하는 역할을 한다.

git remote add <별칭> <URL>을 적으면 등록이 되며 git push나 git pull을 할 때 별칭을 사용할 수 있다.

git remote show를 하면 현재 등록된 별칭의 목록이 나오며 git remote rm으로 삭제할 수도 있다.



1. ***SVN에서 Git로 옮겨가기***
2. ***SVN과 Git 명령어 비교***

- svnadmin create → git init

- svn checkout → git clone

- svn update → git pull

- svn add → git add

- svn commit → git add, git commrheit

- svn status → git status

- svn switch BRANCH → git checkout BRANCH

- svn merge → git merge

- svn revert FILE → git checkout FILE

1. ***SVN 서버 이용하기***

git-svn은 서브버전 저장소와 통신할 때 사용하는 도구이다. Git-svn이 설치되었는지 --version옵션으로 확인하고 설치되어 있지 않다면 아래와 같이 명령어를 이용하여 설치하면 된다.

***- git-svn 설치***

☞ git svn --version

☞ sudo apt-get install git-svn

☞ sudo apt-get installl subversion-perl

저장소를 가져오는 일은 git명령어와 유사하다 --prefix옵션은 서브버전에서 가져온 모든 브랜치에 접두어를 추가한다. Branch와 tag, 트렁크의 위치가 다르다면 -b, -t, -T옵션을 이용하여 따로 지정할 수도 있다.

***- 저장소 가져오기***

☞ git svn clone --prefix svn/ -s svn://URL

☞ -b, -t, -T옵션을 이용하여 브랜치, 태그, 트렁크 위치 지정

최신상태를 유지하기 위해서 fetch후 rebase를 하거나 바로 rebase를 한다면 최신상태 유지가 된다.

***- 최신상태 유지하기***

☞ git svn fetch → git rebase

☞ git svn rebase

푸싱하는 일도 git와 유사하다 git svn dcommit 명령어를 이용하여 푸싱할 수 있다.

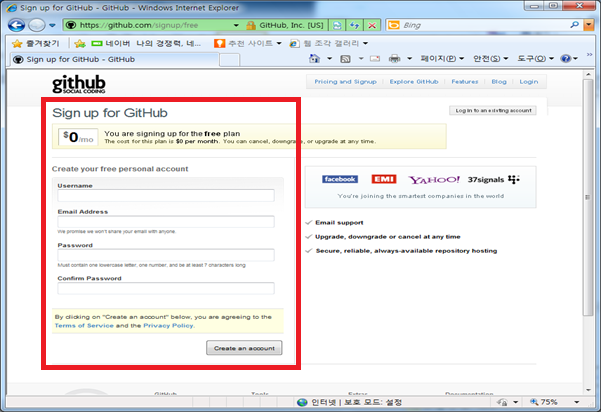
***- SVN에 푸싱하기***

☞ git svn dcommit

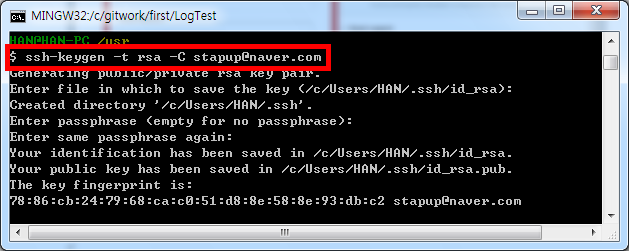
1. ***github 이용하기***

github는 몇 가지 제약이 있긴 하지만 무료로 Git 저장소를 호스팅 해주는 서비스이다. Git는 원격이 아닌 단독으로 사용해도 유용하지만 원격저장소와 함께 사용하면 매우 좋다. 무료로 원격 저장소를 제공해주는 GitHub는 유용하게 쓰인다.

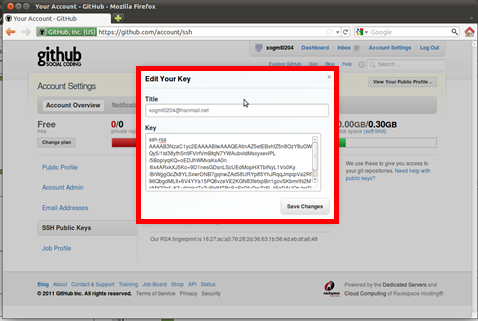
* 1. [***http://github.com***](http://github.com) ***회원가입***

******

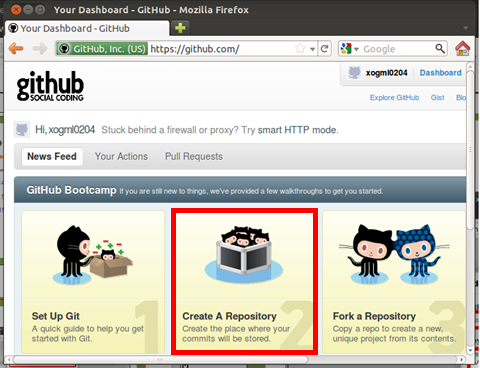
* 1. ***ssh-keygen -t rsa -C EMAIL\_NAME : ssh-key 생성***

******

* 1. ***github 홈페이지에 ssh-key 등록***

******

* 1. ***github에 원격 저장소를 생성***

******

* 1. ***git명령어를 이용하여 원격저장소에 등록하거나 가져올 수 있다.***

# ***부록. Git 명령어 요약***

## ***A.1 설정과 초기화***

* 전역 사용자명/이메일 구성하기
  + git config --global user.name “Your Name”
  + git config --global user.email [you@example.com](mailto:you@example.com)
* 해당 폴더에 사용자명/이메일 구성하기
  + git config user.name “Your Name”
  + git config user.email “you@example.com”
* Git 출력결과 색상 활성화하기
  + git config --global color.ui “auto”
* 저장소 초기화하기
  + git init
* 저장소 복제하기
  + git clone <저장소 url>
* 원격 저장소 추가하기
  + git remote add <원격 저장소명> <원격저장소 url>

## ***A.2 기본적인 사용법***

* 새로운 파일을 추가하거나 스테이징 하기
  + git add <파일>
* add명령 대화모드 사용하기
  + git add -i
* 커밋하기
  + git commit -m “<메시지>”
* 수정되고 추적되는 모든 파일의 변경 사항 커밋하기
  + git commit -m “<메시지>” -a
* 마지막 커밋 고치기
  + git commit -m “<메시지>” --amend
* 이전 커밋을 수정하고 커밋 메시지 재사용하기
  + git commit -C HEAD --amend

## ***A.3 브랜치***

* 지역 브랜치 목록 보기
  + git branch
* 원격 브랜치 목록 보기
  + git branch -r
* 지역과 원격을 포함한 모든 브랜치 목록 보기
  + git branch -a
* 현재 브랜치에서 새로운 브랜치 생성하기
  + git branch <새로운 브랜치>
* 다른 브랜치 체크아웃 하기
  + git checkout <브랜치>
* 현재 브랜치에서 새로운 브랜치 생성하고 체크아웃하기
  + git checkout -b <새로운 브랜치>
* 다른 시작 지점에서 브랜치 생성하기
  + git branch <새로운 브랜치> <브랜치를 생성할 위치>
* 기존의 브랜치를 새로운 브랜치로 덮어쓰기
  + git branch -f <기존브랜치> <브랜치를 생성할 위치>
* 브랜치를 옮기거나 브랜치명 변경하기
  + git checkout -m <기존브랜치> <새로운 브랜치>
* 다른 브랜치를 현재 브랜치로 합치기
  + git merge <브랜치>
* 커밋하지 않고 합치기
  + git merge --no-commit <브랜치>
* 선택하여 합치기
  + git cherry-pick <커밋명>
* 커밋하지 않고 선택하여 합치기
  + git cherry-pick -n <커밋명>
* 브랜치의 이력을 다른 브랜치에 합차기
  + git merge --squash <브랜치>
* 삭제할 브랜치가 현제 브랜치에 합쳐졌을 경우에만 삭제
  + git branch -d <삭제할 브랜치>
* 브랜치를 무조건 삭제하기
  + git branch -D <삭제할 브랜치>

## ***A.4 이력***

* 모든 이력보기
  + git log
* 변경 사항을 보여주는 패치와 함께 로그 표시하기
  + git log -p
* 1개의 항목만 보이도록 로그 개수 제한하기
  + git log -1
* 20개의 항목과 패치만 보이도록 로그 제한하기
  + git log -20 -p
* 6시간 동안의 커밋 로그 보기
  + git log --since=”6hours”
* 이틀 전까지의 커밋 로그 보기
  + git log --before=”2days”
* HEAD보다 세 개 이전의 커밋 로그 보기
  + git log -1 HEAD~3
  + git log -1 HEAD^^^
  + git log -1 HEAD~1^^
* 두 지점 사이의 커밋 로그 보기
  + git log <시작 지점>...<끝 지점>
* 각 항목의 로그 이력 한 줄씩 보기
  + git log --pretty=oneline
* 각 항목마다 영향 받은 줄의 통계 보기
  + git log -stat
* 커밋할 시점의 파일 상태 보기
  + git log --name-status
* 현재 작업 트리와 인덱스의 차이점 보기
  + git diff
* 인덱스와 저장소의 차이점 보기
  + git diff --cached
* 작업 트리와 저장소의 차이점 보기
  + git diff HEAD
* 작업트리와 특정 위치간의 차이점 보기
  + git diff <시작지점>
* 저장소의 두 지점 사이의 차이점 보기
  + git diff <시작 지점> <끝 지점>
* 차이점의 통계 보기
  + git diff -stat <시작 지점> <끝 지점>
* 파일의 커밋 정보 줄 단위로 보기
  + git blame <파일>
* 파일의 줄 단위의 복사, 붙여넣기, 이동 정보 보기
  + git blame -M <파일>
* 파일의 줄 단위의 이동과 원본 파일 정보 보기
  + git blame -C -C <파일>
* 로그에서 복사와 붙여 넣은 정보 보기
  + git log -C -C -p -1 <특정 지점>

## ***A.5 원격 저장소***

* 저장소 복제하기
  + git clone <저장소>
* 마지막 200개의 커밋만 포함하여 저장소 복제하기
  + git clone –depth 200 <저장소>
* 새로운 원격 저장소 추가하기
  + git remote add <원격 저장소> <저장소 URL>
* 모든 원격 브랜치 목록 보기
  + git branch -r
* 원격 브랜치에서 지역 브랜치 생성하기
  + git branch <새로운 브랜치> <원격 브랜치>
* 원격 태그에서 지역 브랜치 생성하기
  + git branch <새로운 브랜치> <원격 태그>
* origin 저장소에서 합치지 않고 지역 브랜치로 변경사항 가져오기
  + git fetch <원격 저장소>
* 원격 저장소에서 변경 사항을 가져와 현재 브랜치에 합치기
  + git pull <원격 저장소>
* origin 저장소에서 변경사항을 가져와 현재 브랜치에 합치기
  + git pull
* 지역 브랜치를 원격 브랜치에 푸싱하기
  + git push <원격 저장소> <지역브랜치>:<원격브랜치>
* 지역 브랜치를 동일한 이름의 원격 브랜치에 푸싱하기
  + git push <원격 저장소> <지역브랜치>
* 새로운 로컬 브랜치를 원격 저장소에 푸싱하기
  + git push <원격 저장소> <지역 브랜치>
* 지역 변경 사항을 origin 저장소에 푸싱하기
  + git push
* 원격 브랜치 삭제하기
  + git push <원격 저장소> :<원격 브랜치>
* 원격 저장소에서 쓸모가 없어진 원격 브랜치 제거하기
  + git remote prune <원격 저장소>
* 원격 저장소를 제거하고 관련된 브랜치도 제거하기
  + git remote rm <원격 저장소>

## ***A.6 Git와 SVN 연결하기***

* 서브버전 저장소 전체를 복제하기
  + git svn clone <svn 저장소>
* 표준 레이아웃을 사용하는 서브버전 저장소 전체를 복제하기
  + git svn clone –s <svn repository>
* 비 표준 레이아웃을 사용하는 서브버전 저장소 전체를 복제하기
  + git svn clone –T <트렁크 경로> -b <브랜치 경로> -t <태그경로> <svn 저장소>
* 표준 레이아웃을 사용하는 서브버전 저장소의 리비전 2321을 복제하기
  + git svn clone -s –r 2321
* 표준 레이아웃을 사용하는 서브버전 저장소를 복제하고 모든 원격 브랜치에 접두어 추가하기
  + git svn clone -s --prefix svn/ <svn 저장소>
* 상위의 서브버전 저장소에서 갱신하고 재정렬 하기
  + git svn rebase
* 상위의 서브버전 저장소에 커밋 푸싱하기
  + git svn dcommit
* 상위의 서브버전 저장소에 푸싱될 커밋 목록 보기
  + git svn dcommit -n
* 서브버전 저장소 로그 보기
  + git svn log
* 서브버전 저장소 파일의 커밋정보 줄 단위로 보기
  + git svn blame <파일>