





Dongwoo Kang
kangdw@dankook.ac.kr

What is the *git*?

git

About git

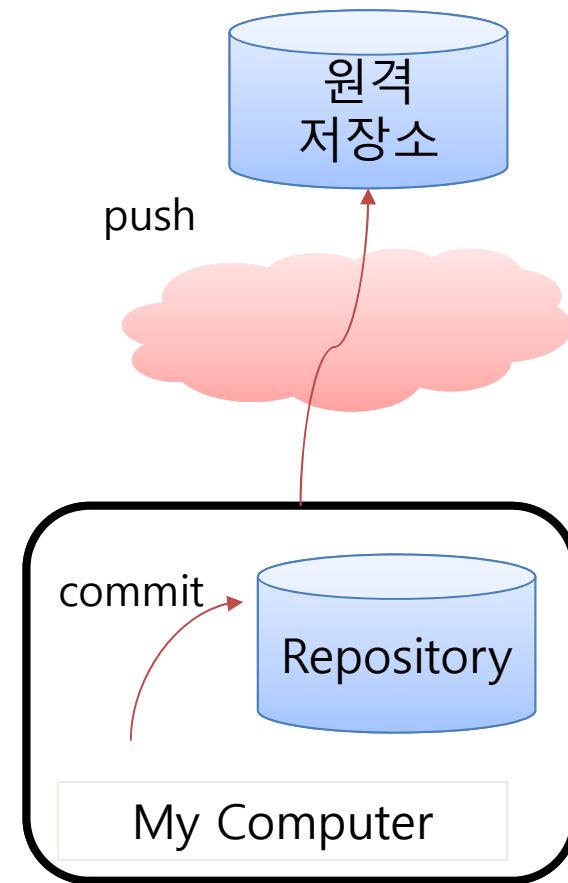
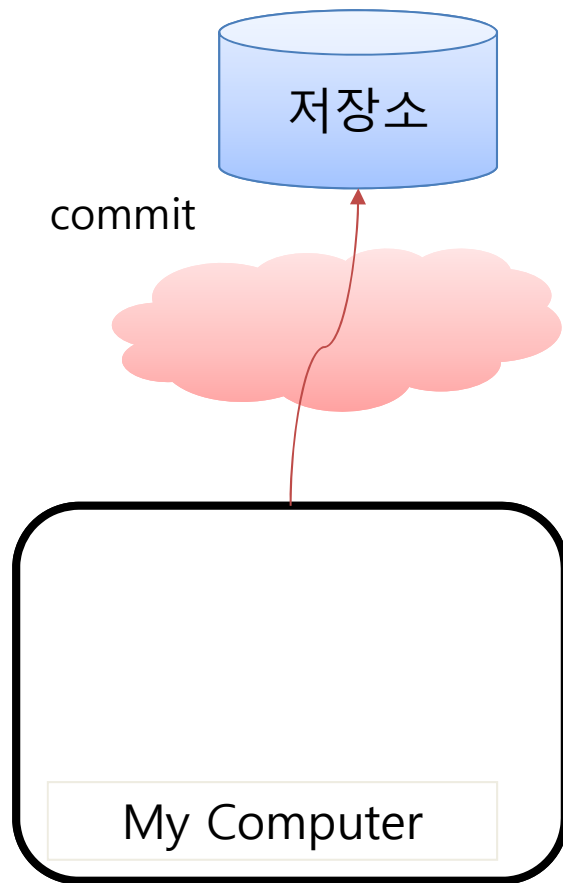
- DVCS(Distributed Version Control Systems)
- 2002년 BitKeeper 사용
- 2005년 BitKeeper 무료 사용 제고
- 2005년, 리눅스 토발즈 개발
- 목표
 - 빠른 속도
 - 단순한 구조
 - 비선형적인 개발(수천 개의 동시 다발적인 브랜치)
 - 완벽한 분산
 - 리눅스 커널 같은 대형 프로젝트에도 유용

[git](#) 미국·영국 [gɪt]  영국식  [+단어장추가](#)
[명사] (英 속어) 재수 없는 [명칭한] 놈

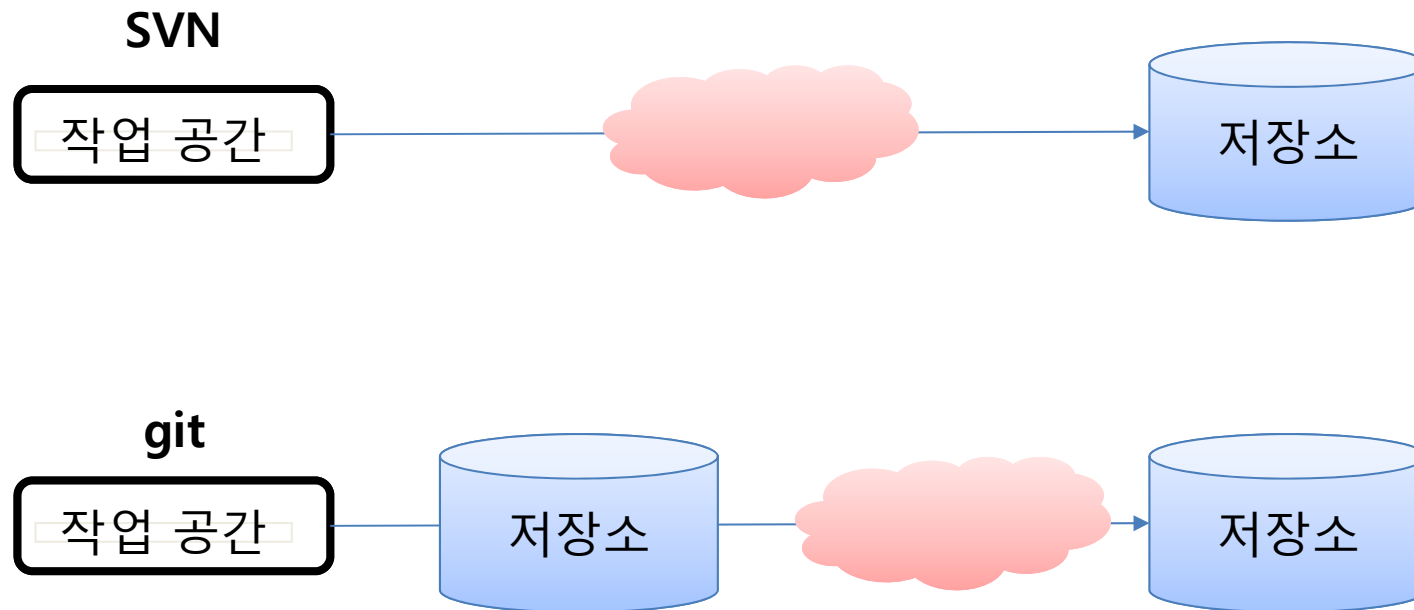
```
GIT(1)                               Git Manual
GIT(1)

NAME
git - the stupid content tracker
```

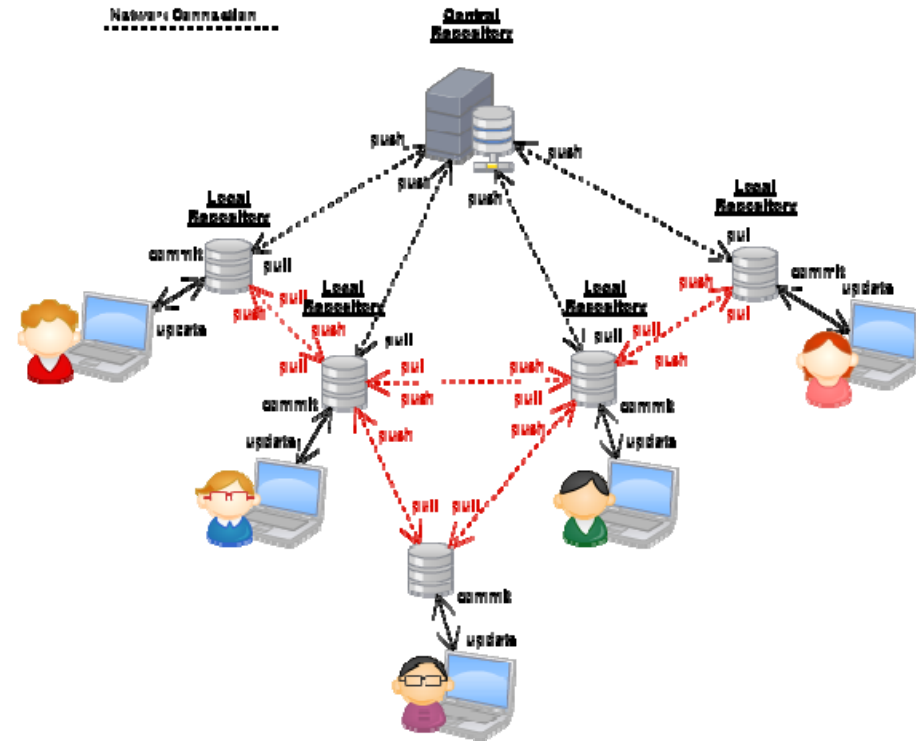
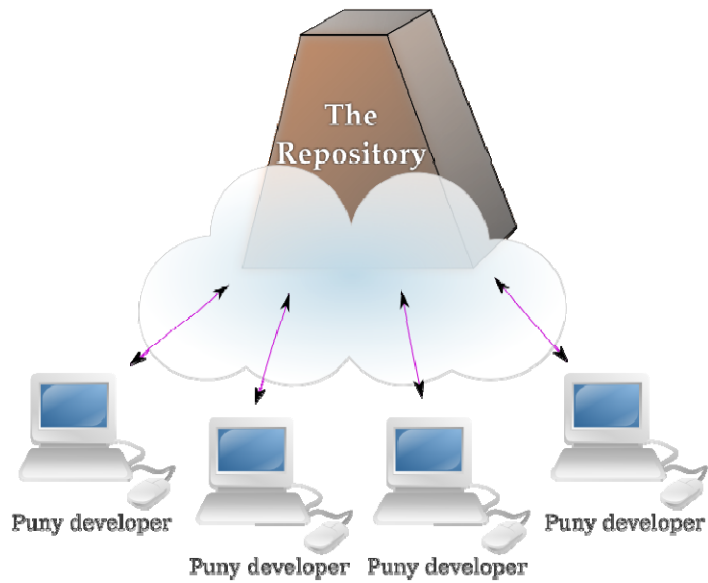
- CVCS(Centralized Version Control Systems)



- DVCS(Distributed Version Control Systems)



svn vs git



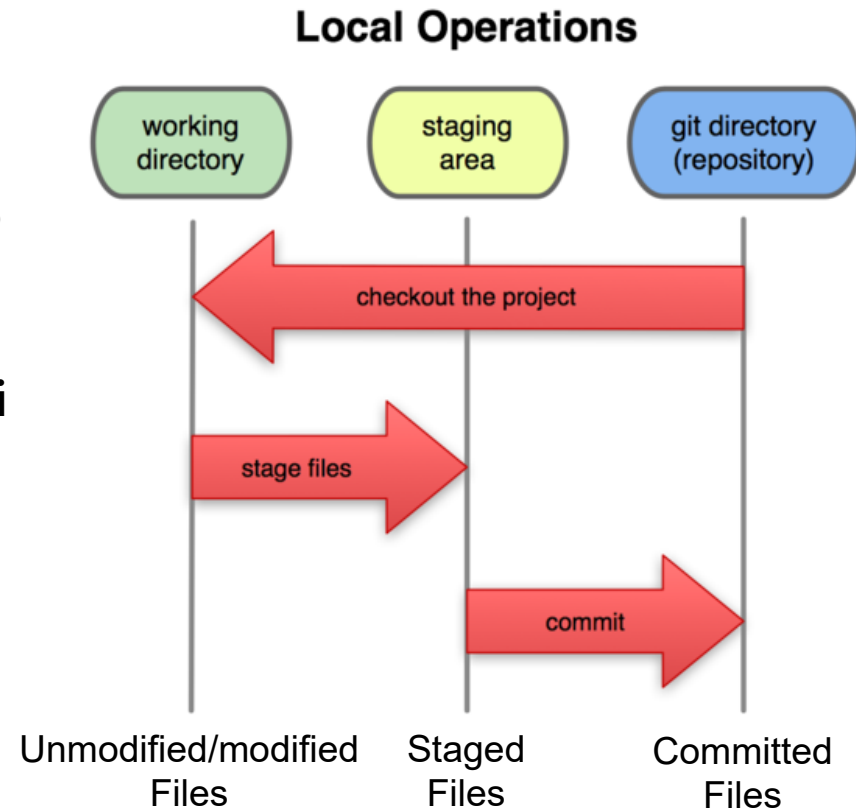
- 빠른 속도
 - 서버에 문제가 생기더라도 복구 가능
 - Remote 저장소(repository)와 연결된 모든 local 저장소는 사본 보유
 - 자유로운 branch
 - Local에서 실행
 - Git은 프로젝트의 history를 조회할 때 server 없이 조회
-

git의 무결성

- Git은 모든 데이터를 저장하기 전에 체크섬(checksum 또는 hash)을 구하고 그 체크섬으로 데이터를 관리
 - 체크섬 없이 어떠한 파일이나 directory도 변경할 수 없다.
 - 체크섬은 Git에서 사용하는 가장 기본적인(atomic) 데이터 단위이자 Git의 기본 철학
 - Git 없이는 체크섬을 다룰 수 없어서 파일의 상태도 알 수 없고 데이터를 잃어버릴 수도 없다.
 - Git은 SHA-1 hash를 사용하여 체크섬을 만든다. 만든 체크섬은 40자 길이의 16진수 문자열
 - Git은 모든 것을 hash로 식별하기 때문에 이런 값은 여기저기서 보일 것이다. 실제로 Git은 파일을 이름으로 저장하지 않고 해당 파일의 hash로 저장한다.
-

Staging Area

- **Staging Area**
 - Commit 전 단계, index라고도 불림
- **Working Directory**에서 파일을 수정
- **Staging Area**에 파일을 Stage 해서 commit할 snapshot을 만듦.
- **Staging Area**에 있는 파일들을 commit해서 git Directory에 영구적인 snapshot으로 저장



Staging Area

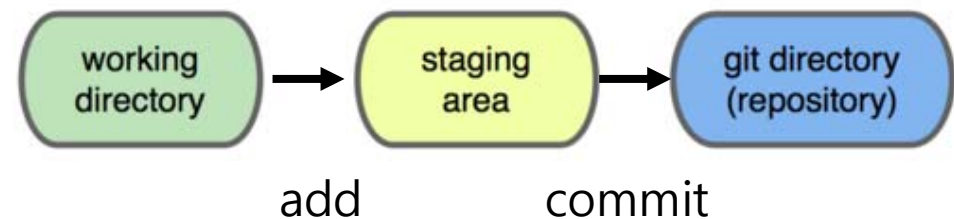
- 비번 찾기 기능 추가가 다음 빌드로 미루어 진다면??

File List

- **login.c** 비번 찾기 기능 추가
- main.c
- **func1.c** 버그 수정
- **func2.c**
- func3.c

1. 수정된 login.c 백업
2. svn revert를 이용해 원래 상태로 복원
3. 수정 내역 전체를 커밋
4. 백업해 두었던 login.c 다시 복구

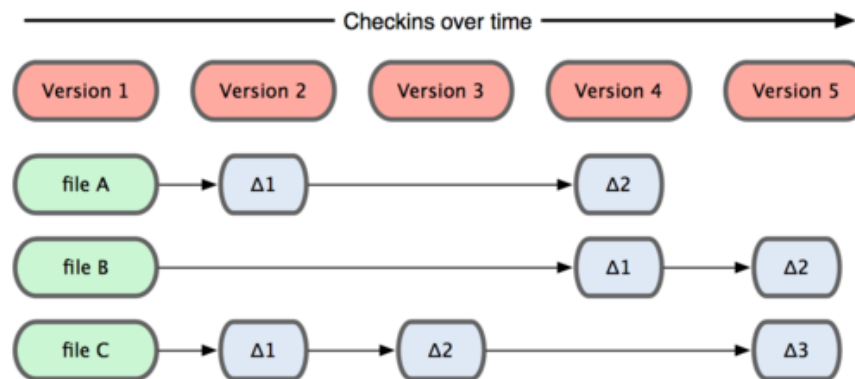
1. 커밋할 파일들만 Staging Area에 추가
2. 로컬 저장소로 커밋



git takes snapshots

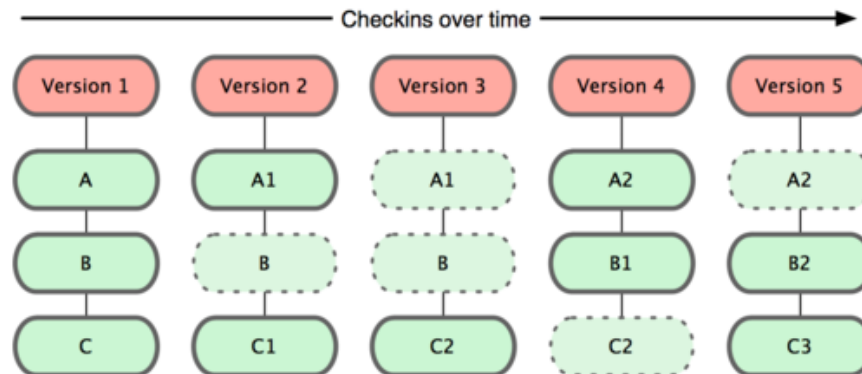
- Just Snapshot, Not Differences

Subversion



- 각 파일의 변화를 시간순으로 관리

Git



- git의 데이터는 파일 시스템의 스냅샷
- 파일이 달라지지 않았을 경우, 이전 상태의 파일에 대한 링크만 저장

git install

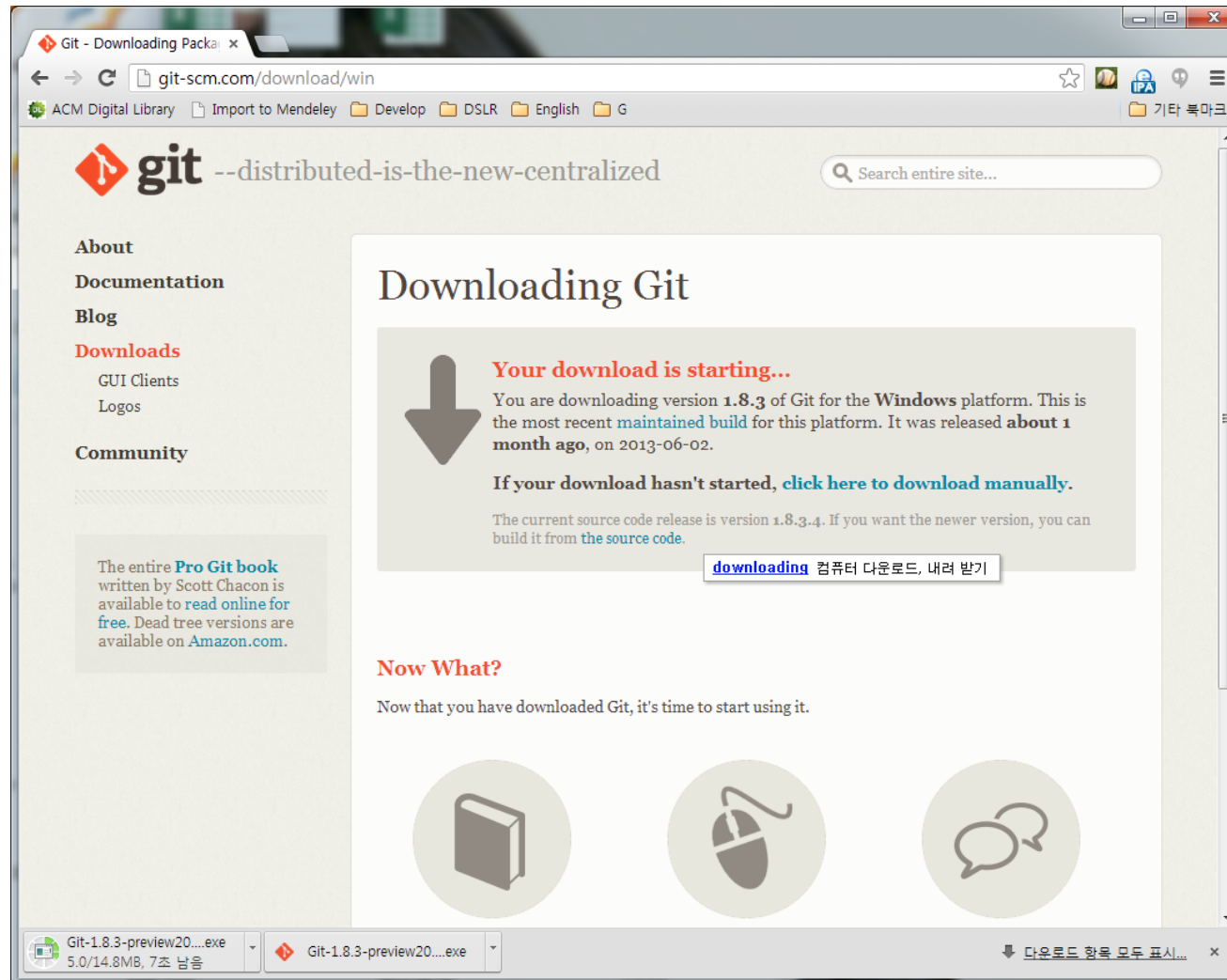
- **CentOS**
 - `$yum install git`
 - **Fedora**
 - `$ yum install curl-devel expat-devel gettext-devel openssl-devel zlib-devel`
 - `$ yum install git-core`
 - **Ubuntu**
 - `$ apt-get install libcurl4-gnutls-dev libexpat1-dev gettext libz-dev libssl-dev`
 - `$ apt-get install git`
-

Install -Windows

- Download
- <http://git-scm.com/>



Install -Windows

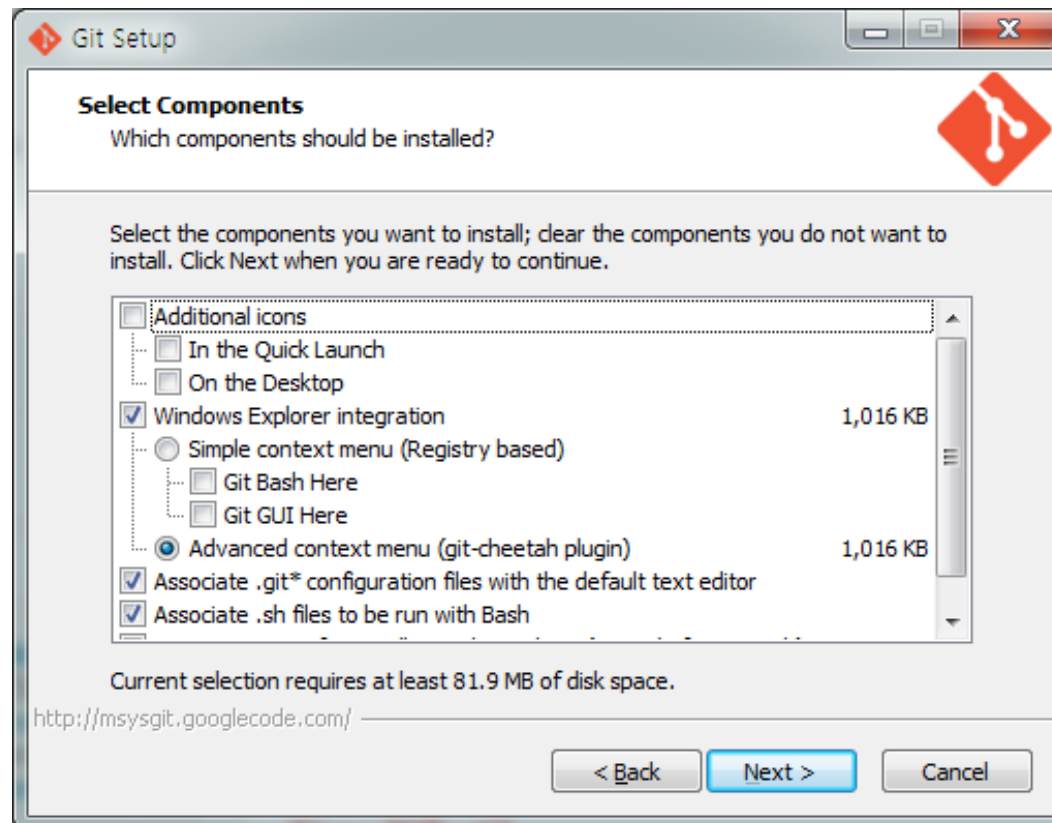


Install -Windows

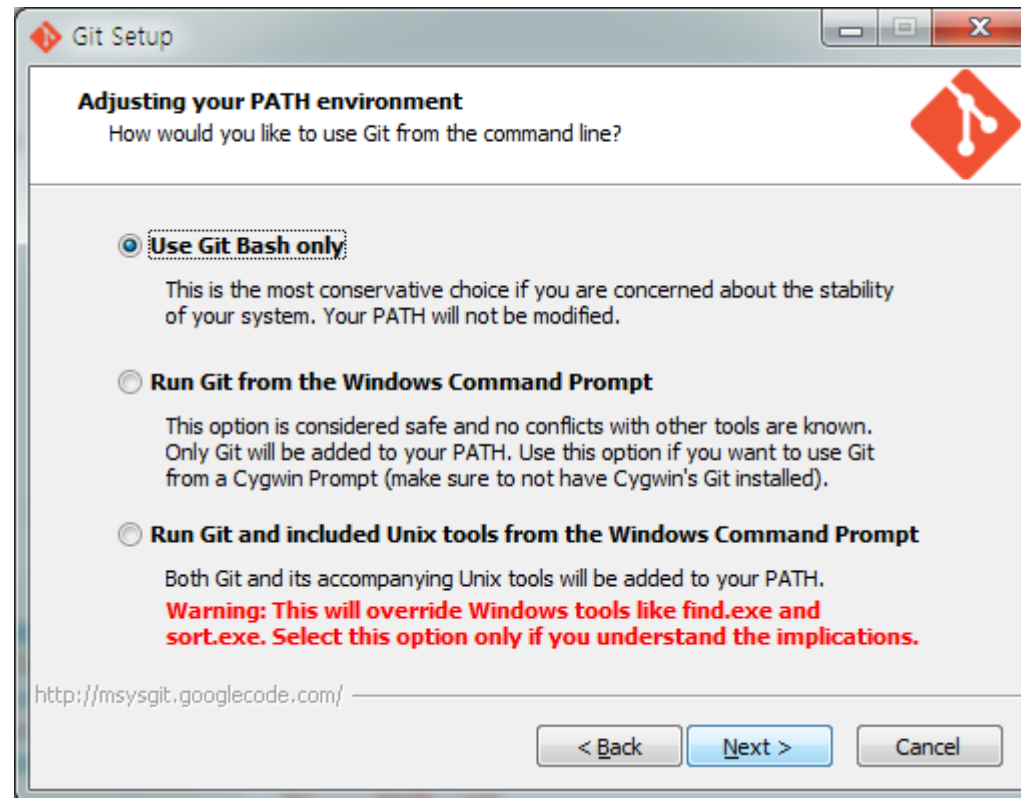
- 설치



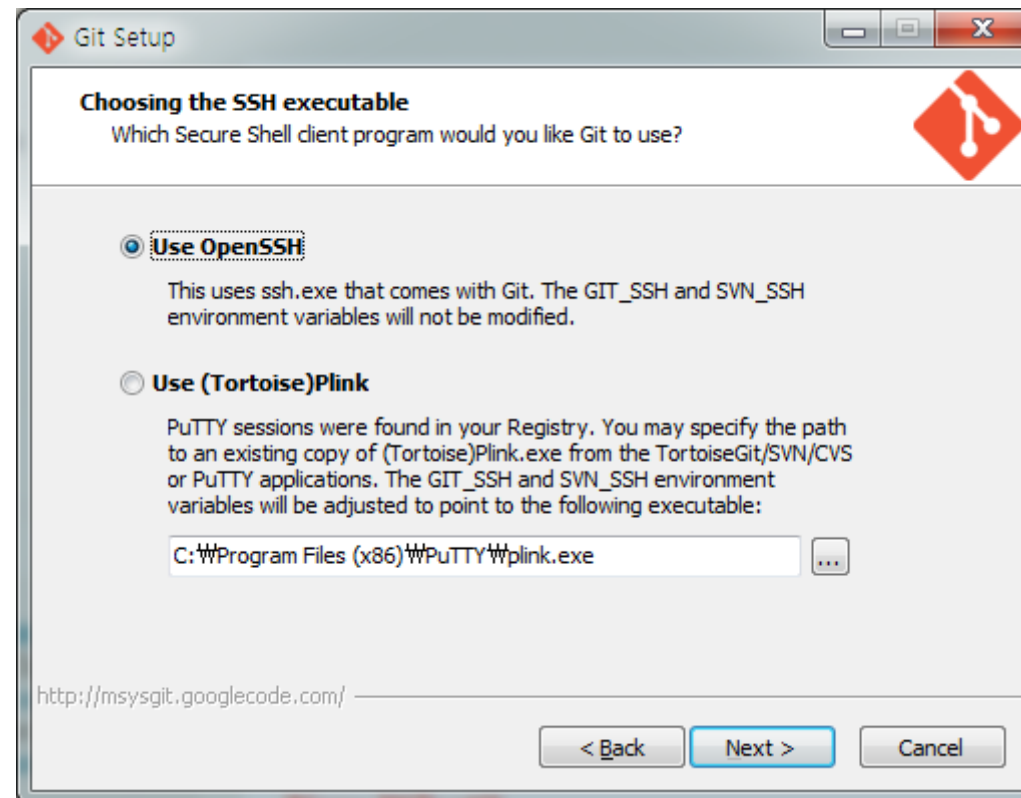
Install -Windows



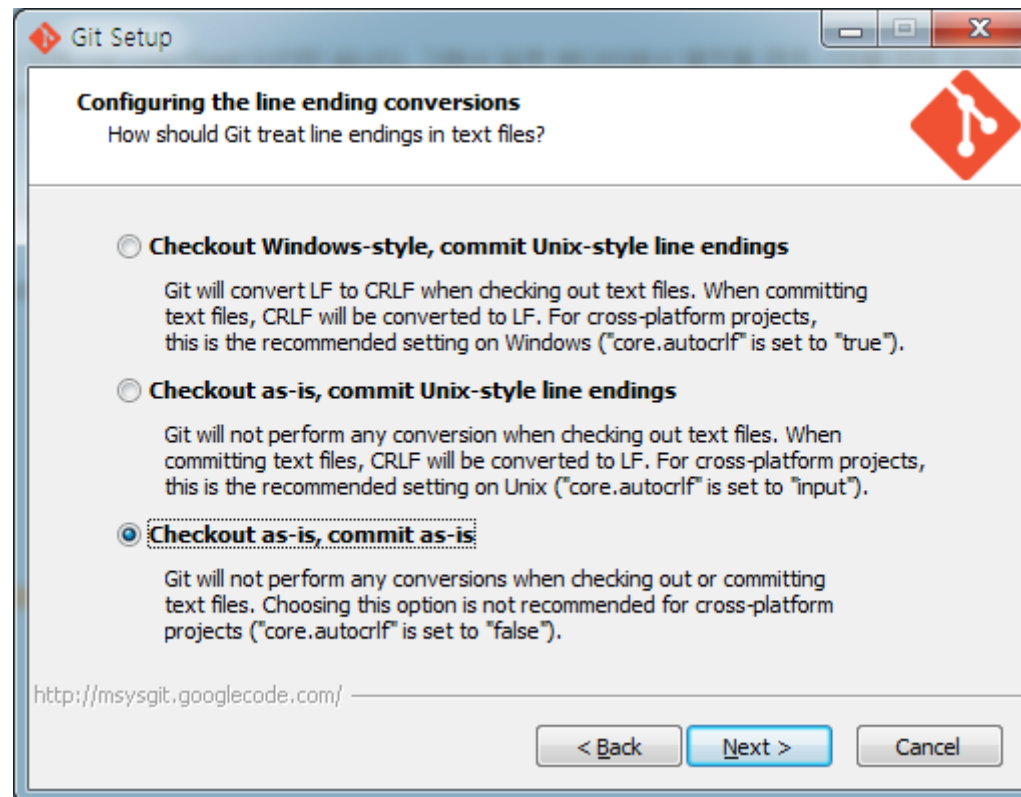
Install -Windows



Install -Windows



Install -Windows



git 환경 설정

- Git을 설치하고 나면 Git의 사용 환경을 적절히 설정해 주어야 한다. 한 번만 설정하면 된다. 설정한 내용은 Git을 업그레이드해도 유지된다. 또한, 명령어로 언제든지 다시 바꿀 수 있다.
 - Git은 'git config'라는 도구로 설정 내용을 확인하고 변경할 수 있다.
 - Git은 이 설정에 따라 동작한다. 이 설정 파일은 세 가지나 된다.
 - 전역 설정 : 시스템의 모든 사용자와 모든 저장소에 적용되는 설정이다. `git config --system` 옵션으로 이 파일을 읽고 쓸 수 있다.
 - Linux : /etc/gitconfig 파일: 시스템의 모든 사용자와 모든 저장소에 적용되는 설정이다. `git config --system` 옵션으로 이 파일을 읽고 쓸 수 있다.
 - Windows : C:\Program Files (x86)\Git\etc\gitconfig
 - 특정 사용자 : 특정 사용자에게만 적용되는 설정이다. `git config --global` 옵션으로 이 파일을 읽고 쓸 수 있다.
 - Linux : ~/.gitconfig 파일
 - Windows : \$HOME Directory(C:\Documents and Settings\%USER)에 있는 .gitconfig 파일
 - 특정 저장소에만 적용
 - .git/config: 이 파일은 git Directory에 있고 특정 저장소(혹은 현재 작업 중인 프로젝트)에만 적용된다. 각 설정은 역순으로 우선시 된다. 그래서 .git/config가 /etc/gitconfig보다 우선한다.
-

Configure git

- 사용자 등록

```
$ git config --global user.name "Your Name"
```

```
$ git config --global user.email youremail@whatever.com
```

- --global 옵션으로 설정한 것은 딱 한 번만 수행하면 충분(계정별 git 설정)
- 해당 시스템에서 해당 사용자가 사용할 때에는 이 정보를 사용한다.
- 만약 프로젝트마다 다른 이름과 이메일 주소를 사용하고 싶으면 --global 옵션을 빼고 명령을 실행한다.

```
CDANG@CDANG-WIN /c
$ mkdir test

CDANG@CDANG-WIN /c
$ cd test

CDANG@CDANG-WIN /c/test
$ pwd
/c/test

CDANG@CDANG-WIN /c/test
$ git config --global user.name "Dongwoo Kang"

CDANG@CDANG-WIN /c/test
$ git config --global user.email kangdw@dankook.ac.kr

CDANG@CDANG-WIN /c/test (master)
$ git config --global --list
user.name=Dongwoo Kang
user.email=kangdw@dankook.ac.kr

CDANG@CDANG-WIN /c/test (master)
$
```

- 설정 확인
\$ git config --list

- 모든 설정 내용을 확인할 수 있다:

user.name=Dongwoo Kang

user.email=kangdw@dankook.ac.kr

core.repositoryformatversion=0

core.filemode=true

core.bare=false

core.logallrefupdates=true

color.ui=true

git 편집기

- 사용자 정보를 설정하고 나면 Git에서 사용할 텍스트 편집기를 고른다. 기본적으로 Git은 시스템의 기본 편집기를 사용하고 보통 이것은 Vi나 Vim이다. 하지만, Emacs 같은 다른 텍스트 편집기를 사용하고 싶다면 다음과 같이 실행
 - `$ git config --global core.editor emacs`
-

- Merge 충돌을 해결하기 위해 사용하는 Diff 도구를 설정
 - vimdiff를 사용하고 싶으면 다음과 같이 실행한다:
 - `$ git config --global merge.tool vimdiff`
 - 이렇게 kdiff3, tkdiff, meld, xxdif, emerge, vimdiff, gvimdiff, ecmerge, opendiff를 사용할 수있다. 물론 다른 도구도 사용할 수 있다.
-

- 도움말 보기
 - 명령어에 대한 도움말이 필요할 때 도움말을 보는 방법은 세 가지이다:
 - `$ git help <verb>`
 - `$ git <verb> --help`
 - `$ man git-<verb>`

 - 예를 들어 다음과 같이 실행하면 `config` 명령에 대한 도움말을 볼 수 있다:
 - `$ git help config`
-

git 기초

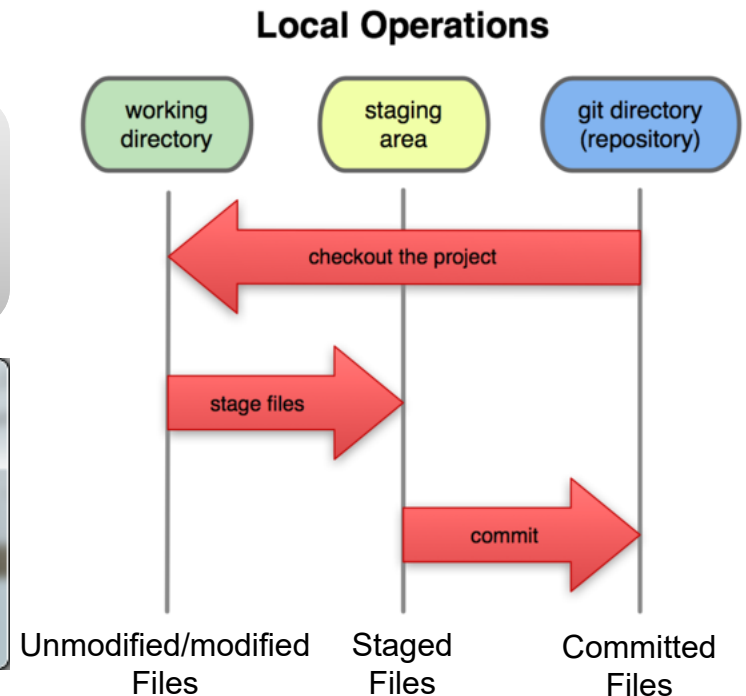
git 저장소 만들기

git 저장소 만들기
• \$ git init

- **.git 하위 디렉토리 생성**
 - 저장소를 이루는 파일(Skeleton)
- **저장소를 만들었기 때문에 추가적으로 추적 및 저장소에 파일들을 추가 해야 한다.**

git 파일 추가(Staging Area에 추가)
\$ git add *.c
\$ git add README
\$ git commit -m 'initial project version'

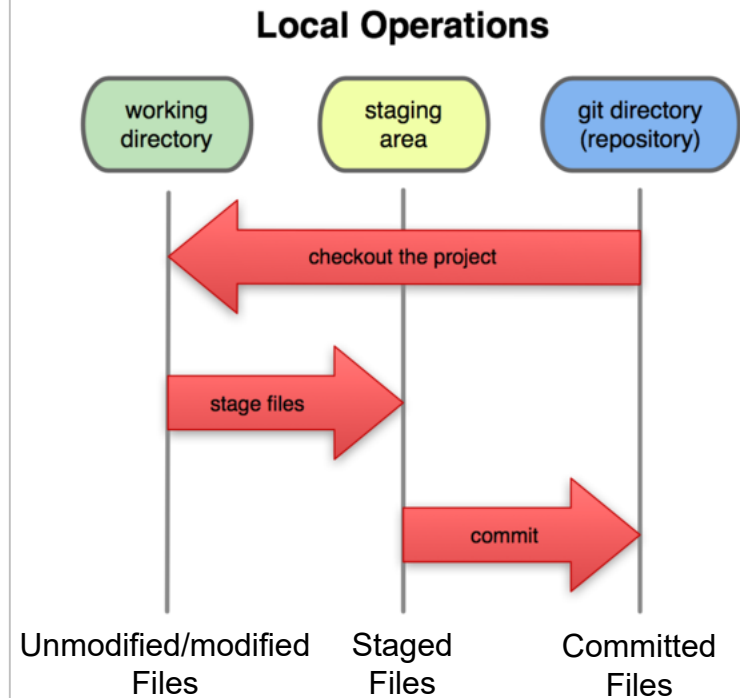
```
C:\Windows\system32\cmd.exe
C:\W>mkdir test
C:\W>cd test
C:\Wtest>git init
Initialized empty Git repository in C:/test/.git/
C:\Wtest>
```



git commit

- **git commit**

```
# Please enter the commit message for your changes. Lines
# starting
# with '#' will be ignored, and an empty message aborts the
# commit.
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       new file:   README_2.txt
#       modified:   main.c
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       .gitignore
#
# ~
# ~
# ~
# ~
```



git commit

```
rediori@localhost:project.sample $ git commit -a -m "4th commit"  
[master 92e7b55] 4th commit  
1 files changed, 2 insertions(+), 0 deletions(-)
```

git 저장소

```
ca. C:\Windows\system32\cmd.exe
파일을 찾을 수 없습니다.

C:\wtest>dir /a
C 드라이브의 볼륨에는 이름이 없습니다.
볼륨 일련 번호: B608-6732

C:\wtest 디렉터리

2013-07-27 오후 12:29 <DIR>      .
2013-07-27 오후 12:29 <DIR>      ..
2013-07-27 오후 11:42 <DIR>      .git
                0개 파일                0 바이트
                3개 디렉터리  19,027,509,248 바이트 남음

C:\wtest>cd .git

C:\wtest\.git>dir
C 드라이브의 볼륨에는 이름이 없습니다.
볼륨 일련 번호: B608-6732

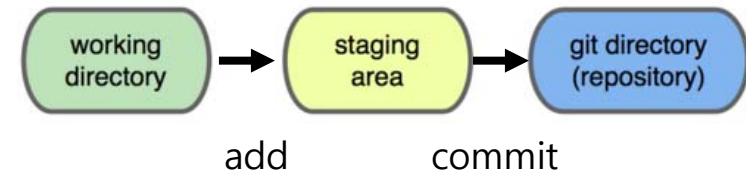
C:\wtest\.git 디렉터리

2013-07-27 오후 12:29          157 config
2013-07-27 오후 12:29          73 description
2013-07-27 오후 12:29          23 HEAD
2013-07-27 오후 12:29 <DIR>      hooks
2013-07-27 오후 12:29 <DIR>      info
2013-07-27 오후 12:29 <DIR>      objects
2013-07-27 오후 12:29 <DIR>      refs
                3개 파일                253 바이트
                4개 디렉터리  19,027,509,248 바이트 남음

C:\wtest\.git>
```

git 기초

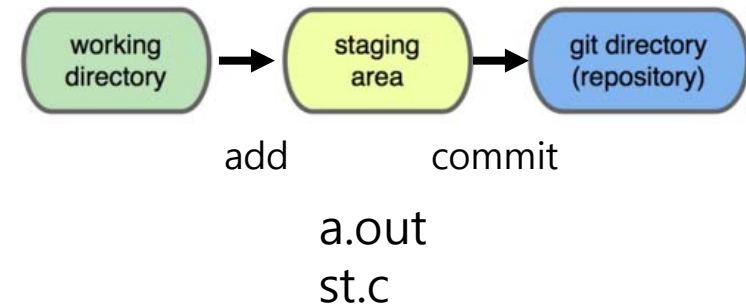
```
rediori@localhost:test $ git init
Initialized empty Git repository in
/home/rediori/src/test/.git/
rediori@localhost:test $ git status
# On branch master
#
# Initial commit
#
# Untracked files:
#   (use "git add <file>..." to include in what
#   will be committed)
#
#       a.out
#       st.c
nothing added to commit but untracked files
present (use "git add" to track)
```



a.out
st.c

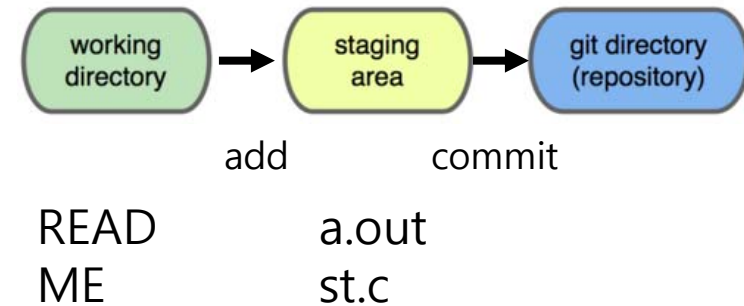
git 기초

```
rediori@localhost:test $ git add .
rediori@localhost:test $ git status
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
#       new file:   a.out
#       new file:   st.c
#
```



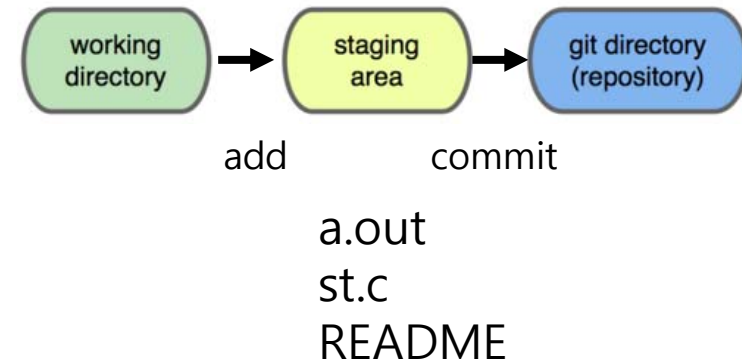
git 기초

```
rediori@localhost:test $ touch README
rediori@localhost:test $ vi README
rediori@localhost:test $ git status
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
#       new file:   a.out
#       new file:   st.c
#
# Untracked files:
#   (use "git add <file>..." to include in what will
#   be committed)
#
#       README
```

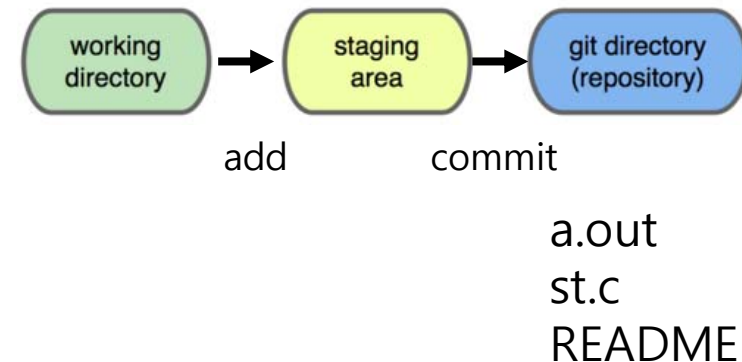


git 기초

```
rediori@localhost:test $ git add README
rediori@localhost:test $ git status
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
#       new file:   README
#       new file:   a.out
#       new file:   st.c
#
```



```
rediori@localhost:test $ git commit -m 'initial project version'
[master (root-commit) 3de2c07] initial project version
3 files changed, 39 insertions(+), 0 deletions(-)
create mode 100644 README
create mode 100755 a.out
create mode 100644 st.c
rediori@localhost:test $ git status
# On branch master
nothing to commit (working directory clean)
```



- rediori@localhost:test \$ git status
 - # On branch master
 - nothing to commit (working directory clean)
 - rediori@localhost:test \$ vi st.c
 - rediori@localhost:test \$ git status
 - # On branch master
 - # Changes not staged for commit:
 - # (use "git add <file>..." to update what will be committed)
 - # (use "git checkout -- <file>..." to discard changes in working directory)
 - #
 - # **modified: st.c**
 - #
 - no changes added to commit (use "git add" and/or "git commit -a")
 - rediori@localhost:test \$
-

Example 2

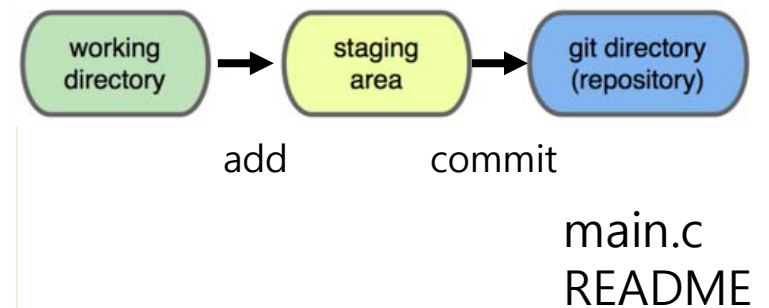
```
rediori@localhost: ~/project.sample
rediori@localhost:project.sample $ git config --global user.name "Dongwoo Kang"
rediori@localhost:project.sample $ git config --global user.email kangdw@dankook.ac.kr
rediori@localhost:project.sample $ git init
Initialized empty Git repository in /home/rediori/project.sample/.git/
rediori@localhost:project.sample $ git status
# On branch master
#
# Initial commit
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       README
#       main.c
nothing added to commit but untracked files present (use "git add" to track)
rediori@localhost:project.sample $ git add *.c
rediori@localhost:project.sample $ git add README
rediori@localhost:project.sample $ git status
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
#       new file:   README
#       new file:   main.c
#
rediori@localhost:project.sample $
```

```
graph LR
    A(working directory) -- add --> B(staging area)
    B -- commit --> C(git directory (repository))
```

main.c
README

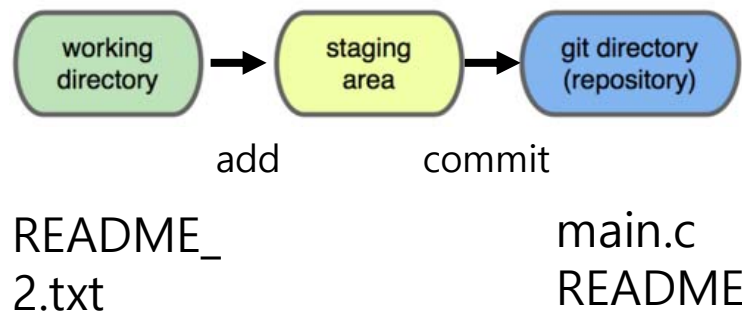
Example 2

```
rediori@localhost:project.sample $ git commit -m  
'Initial Project 버전'  
[master (root-commit) 52aa809] Initial Project 버전  
0 files changed, 0 insertions(+), 0 deletions(-)  
create mode 100644 README  
create mode 100644 main.c  
rediori@localhost:project.sample $ git status  
# On branch master  
nothing to commit (working directory clean)
```



Example 2

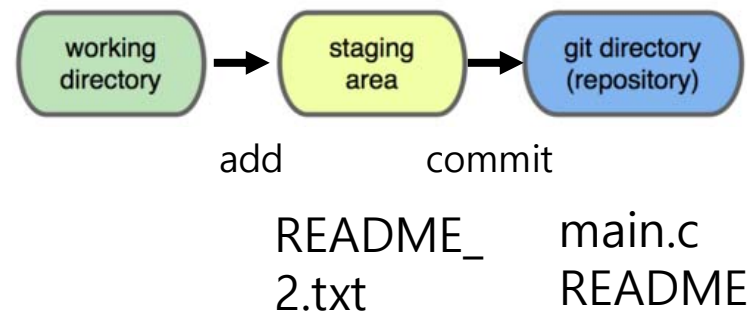
```
rediori@localhost:project.sample $ touch README_2.txt
rediori@localhost:project.sample $ vi README_2.txt
rediori@localhost:project.sample $ git status
# On branch master
# Untracked files:
#   (use "git add <file>..." to include in what will be
#   committed)
#
#       README_2.txt
nothing added to commit but untracked files present (use
"git add" to track)
```



- 추적되지 않는 파일들 (Untracked files)에 속해 있는 README_v2
- git은 Untracked 파일을 아직 스냅샷(커밋)에 넣어지지 않은 파일이라고 본다.
- 파일이 Tracked 상태가 되기 전까지는 git은 절대 그 파일을 커밋하지 않는다
- 이는 작업을 하다보면 생성되는 바이너리 파일 등을 커밋하는 실수를 하지 않도록 한다.

Example 2

```
rediori@localhost:project.sample $ git add README_2.txt
rediori@localhost:project.sample $ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       new file:   README_2.txt
#
```

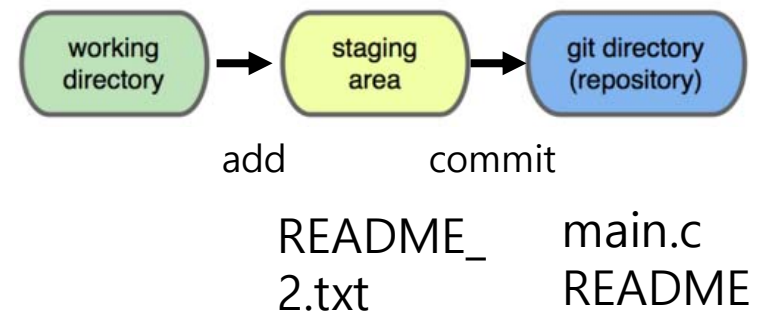


Example 2

- Tracked 상태 파일 수정 (main.c)

```
rediori@localhost:project.sample $ vi main.c
rediori@localhost:project.sample $ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       new file:   README_2.txt
#
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working
directory)
#
#       modified:   main.c
#
```

'Changes to be committed' 에 들어 있는 파일은 Staged 상태라는 것을 의미한다



Example 2

- git add는 파일을 새로 추적할 때도 사용하고 수정한 파일을 Staged 상태로 만들 때도 사용

```
rediori@localhost:project.sample $ git add main.c
rediori@localhost:project.sample $ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       new file:   README_2.txt
#       modified:   main.c
#
```

- main.c, README_2.txt 둘 다 Staged 상태이므로 다음 커밋 시 포함된다.
-

Example 2

- 커밋되지 않은 파일을 수정하게 되면?

```
rediori@localhost:project.sample $ vi main.c
rediori@localhost:project.sample $ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       new file:   README_2.txt
#      modified:   main.c
#
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   main.c
#
```

- **git add 명령을 실행하면 git은 파일을 바로 Staged 상태로 만듦**
 - **커밋을 하면 git commit 명령을 실행하는 시점의 버전이 커밋되는 것이 아니라 마지막으로 git add 명령을 실행했을 때의 버전이 커밋**
 - **즉, git add 명령을 실행한 후에 또 파일을 수정하면 git add 명령을 다시 실행해서 최신 버전을 Staged 상태로 만들어야 한다:**
-

파일 무시하기

- 어떤 파일은 Git이 자동으로 추가하거나 Untracked 파일이라고 보여줄 필요가 없다. 보통 로그 파일이나 빌드 시스템이 자동으로 생성한 파일이 그렇다. 그런 파일을 무시하려면 `.gitignore` 파일을 만들고 그 안에 무시할 파일 패턴을 적는다. 아래는 `.gitignore` 파일의 예이다:

```
$ cat .gitignore
```

```
*.[oa]
```

```
*~
```

- `.o`와 `.a`는 각각 빌드 시스템이 만들어내는 오브젝트와 아카이브 파일
 - `~`로 끝나는 파일은 Emacs나 VI 같은 텍스트 편집기가 임시로 만들어내는 파일
 - `log`, `tmp`, `pid` 같은 디렉토리나, 자동으로 생성하는 문서 같은 것들도 추가할 수 있다. `.gitignore` 파일은 보통 처음에 만들어 두는 것이 편리
 - `.git/info/exclude`
-

파일 무시하기

- **.gitignore** 파일에 입력하는 패턴은 아래 규칙을 따른다:
 - 아무것도 없는 줄이나, #로 시작하는 줄은 무시한다.
 - 표준 Glob 패턴을 사용한다.
 - 디렉토리는 슬래시(/)를 끝에 사용하는 것으로 표현한다.
 - 느낌표(!)로 시작하는 패턴의 파일은 무시하지 않는다.
 - # a comment - 이 줄은 무시한다.
 - # 확장자가 .a인 파일 무시
 - *.a
 - # 윗 줄에서 확장자가 .a인 파일은 무시하게 했지만 lib.a는 무시하지 않는다.
 - !lib.a
 - # 루트 디렉토리에 있는 TODO파일은 무시하고 subdir/TODO처럼 하위디렉토리에 있는 파일은 무시하지 않는다.
 - /TODO
 - # build/ 디렉토리에 있는 모든 파일은 무시한다.
 - build/
 - # `doc/notes.txt`같은 파일은 무시하고 doc/server/arch.txt같은 파일은 무시하지 않는다.
 - doc/*.txt
-

파일 삭제

- **\$git rm**
Tracked 상태의 파일을 삭제한 후 (Staging Area에서 삭제) 커밋
위 명령은 Working Directory에 있는 파일도 삭제 (주의요망)
만약 rm으로 파일만 삭제할 경우, 해당 파일은 Unstaged (Changes not staged for commit) 에
속한다는 것을 확인할 수 있다.
 - **\$ git rm testfile**
 - **rm 'testfile'**
 - **\$ git status**
 - **# On branch master**
 - **#**
 - **# Changes to be committed:**
 - **# (use "git reset HEAD <file>..." to unstage)**
 - **#**
 - **# deleted: grit.gemspec**
 - 커밋하면 파일은 삭제되고 Git은 이 파일을 더는 추적하지 않는다. 이미 파일을 수정했거나 Index
에(Staging Area을 Git Index라고도 부른다) 추가했다면 -f옵션을 주어 강제로 삭제해야 한다. 이
점은 실수로 데이터를 삭제하지 못하도록 하는 안전장치다. 한 번도 커밋한적 없는 데이터는 Git
으로 복구할 수 없다.
-

파일 삭제

- Staging Area에서만 제거하고 워킹 디렉토리에 있는 파일은 지우지 않고 남겨둘 수 있다. 다시 말해서 하드디스크에 있는 파일은 그대로 두고 git만 추적하지 않게 한다. 이것은 .gitignore 파일에 추가하는 것을 빼먹었거나 대용량 로그 파일이나 컴파일된 파일인 .a 파일 같은 것을 실수로 추가했을 때 쓴다. --cached 옵션을 사용하여 명령을 실행한다:
 - **\$ git rm --cached readme.txt**
 - 여러 개의 파일이나 디렉토리를 한꺼번에 삭제할 수도 있다. 아래와 같이 git rm 명령에 file-glob 패턴을 사용한다:
 - **\$ git rm log/W*.log**
 - *앞에 W를 사용
 - 파일명 확장 기능은 쉘에만 있는 것이 아니라 git 자체에도 있기 때문에 필요하다. 이 명령은 log/ 디렉토리에 있는 .log 파일을 모두 삭제한다. 아래의 예제처럼 할 수도 있다:
 - **\$ git rm W*~** 이 명령은 ~로 끝나는 파일을 모두 삭제한다.
-

파일 이름 변경

- `$ git mv file_from file_to`
 - `$ git mv README.txt README`
 - `$ git status`
 - `# On branch master`
 - `# Your branch is ahead of 'origin/master' by 1 commit.`
 - `#`
 - `# Changes to be committed:`
 - `# (use "git reset HEAD <file>..." to unstage)`
 - `#`
 - `# renamed: README.txt -> README`
 - `#`
 - `git mv` 명령은 아래 명령어들을 수행한 것과 완전히 똑같다:
 - `$ mv README.txt README`
 - `$ git rm README.txt`
 - `$ git add README`
-

커밋 히스토리 조회

```
rediori@localhost:project.sample $ git log  
commit 92e7b55848675fe7ad770aea7863f3b474d48460  
Author: Dongwoo Kang <kangdw@dankook.ac.kr>  
Date: Sun Jul 28 23:47:00 2013 +0900
```

4th commit

```
commit 5276d320d5836148075e6a2693ff0c33f0476058  
Author: Dongwoo Kang <kangdw@dankook.ac.kr>  
Date: Sun Jul 28 23:45:58 2013 +0900
```

Third commit

```
commit 4a840a807ca6d7eea391c2963d98181b5135642a  
Author: Dongwoo Kang <kangdw@dankook.ac.kr>  
Date: Sun Jul 28 23:44:36 2013 +0900
```

Second

```
commit 52aa809a70da8e91a063b103c305cc09089ee796  
Author: Dongwoo Kang <kangdw@dankook.ac.kr>  
Date: Sun Jul 28 20:24:06 2013 +0900
```

Initial Project 버전

커밋 히스토리 조회

- `-p`가 가장 유용한 옵션 중 하나다. `-p`는 각 커밋의 diff 결과를 보여준다. 게다가 `-2`는 최근 두 개의 결과만 보여주는 옵션이다:

```
rediori@localhost:project.sample $ git log -p -2
commit f400a5e6d271c5166ee8fe5c5b418514c9bdb0cb
Merge: fcd1fe9 584d084
Author: Dongwoo Kang <kangdw@dankook.ac.kr>
Date: Mon Jul 29 14:05:10 2013 +0900
```

Merge branch 'master' into testing

```
commit 584d08401dc36384cbd4392da1201aac6f3cf4d
Author: Dongwoo Kang <kangdw@dankook.ac.kr>
Date: Mon Jul 29 14:02:44 2013 +0900
```

branch master

```
diff --git a/README b/README
index e69de29..66fa3e9 100644
--- a/README
+++ b/README
@@ -0,0 +1 @@
+master branch
```

- 이 옵션은 직접 diff를 실행한 것과 같은 결과를 출력하기 때문에 동료가 무엇을 커밋했는지 리뷰하고 빨리 조회하는데 유용하다.

커밋 히스토리 조회

- 또 `git log` 명령에는 히스토리의 통계를 보여주는 옵션도 있다. `--stat` 옵션으로 각 커밋의 통계 정보를 조회할 수 있다:

```
rediori@localhost:project.sample $ git log --stat
commit f400a5e6d271c5166ee8fe5c5b418514c9bdb0cb
Merge: fcd1fe9 584d084
Author: Dongwoo Kang <kangdw@dankook.ac.kr>
Date: Mon Jul 29 14:05:10 2013 +0900
```

```
Merge branch 'master' into testing
```

```
commit 584d08401dc36384cbd4392da1201aac6f3cf4d
Author: Dongwoo Kang <kangdw@dankook.ac.kr>
Date: Mon Jul 29 14:02:44 2013 +0900
```

```
branch master
```

```
README | 1 +
```

```
1 files changed, 1 insertions(+), 0 deletions(-)
```

커밋 히스토리 조회

- 다른 또 유용한 옵션은 `--pretty` 옵션이다. 이 옵션을 통해 `log`의 내용을 보여줄 때 기본 형식 이외에 여러 가지 중에 하나를 선택할 수 있다. `oneline` 옵션은 각 커밋을 한 줄로 보여준다. 이 옵션은 많은 커밋을 한 번에 조회할 때 유용하다. 추가로 `short`, `full`, `fuller` 옵션도 있는데 이것은 정보를 조금씩 가감해서 보여준다:

```
rediori@localhost:project.sample $ git log --pretty=oneline
f400a5e6d271c5166ee8fe5c5b418514c9bdb0cb Merge branch 'master' into testing
584d08401dc36384cbd4392da1201aac6f3cf4d branch master
fcd1fe986d9088847480914474490e2c1c501b0e branch testing
92e7b55848675fe7ad770aea7863f3b474d48460 4th commit
5276d320d5836148075e6a2693ff0c33f0476058 Third commit
4a840a807ca6d7eea391c2963d98181b5135642a Second
52aa809a70da8e91a063b103c305cc09089ee796 Initial Project 버전
```

커밋 히스토리 조회

- `format` 옵션이다. 나만의 포맷으로 결과를 출력하고 싶을 때 사용한다. 특히 결과를 다른 프로그램으로 파싱하고자 할 때 유용하다. 이 옵션을 사용하면 포맷을 정확하게 일치시킬 수 있기 때문에 `git`을 새 버전으로 바꿔도 결과 포맷이 바뀌지 않는다:

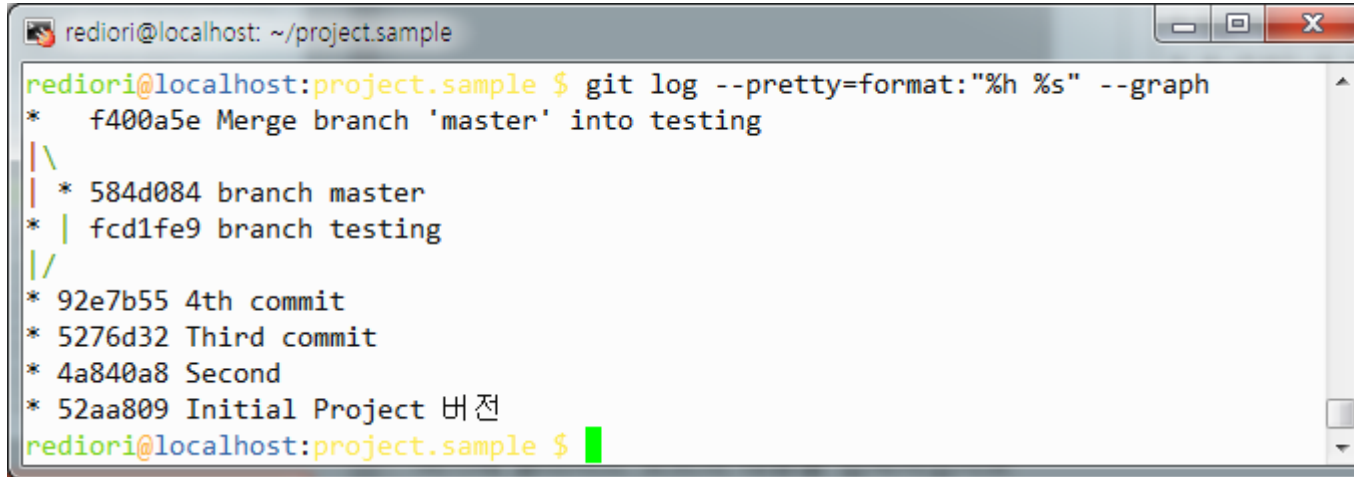
```
rediori@localhost:project.sample $ git log --pretty=format:"%h -
%an, %ar : %s"
f400a5e - Dongwoo Kang, 12 hours ago : Merge branch 'master' into testing
584d084 - Dongwoo Kang, 12 hours ago : branch master
fcd1fe9 - Dongwoo Kang, 12 hours ago : branch testing
92e7b55 - Dongwoo Kang, 26 hours ago : 4th commit
5276d32 - Dongwoo Kang, 26 hours ago : Third commit
4a840a8 - Dongwoo Kang, 27 hours ago : Second
52aa809 - Dongwoo Kang, 30 hours ago : Initial Project 버전
```

커밋 히스토리 조회

- **Option** Description of Output
 - **%H** Commit hash
 - **%h** Abbreviated commit hash
 - **%T** Tree hash
 - **%t** Abbreviated tree hash
 - **%P** Parent hashes
 - **%p** Abbreviated parent hashes
 - **%an** Author name
 - **%ae** Author e-mail
 - **%ad** Author date (format respects the --date= option)
 - **%ar** Author date, relative
 - **%cn** Committer name
 - **%ce** Committer email
 - **%cd** Committer date
 - **%cr** Committer date, relative
 - **%s** Subject
-

커밋 히스토리 조회

- `git log --pretty=format:"%h %s" --graph`



```
rediori@localhost: ~/project.sample
rediori@localhost:project.sample $ git log --pretty=format:"%h %s" --graph
* f400a5e Merge branch 'master' into testing
| \
|  * 584d084 branch master
|  * | fcd1fe9 branch testing
| /
* 92e7b55 4th commit
* 5276d32 Third commit
* 4a840a8 Second
* 52aa809 Initial Project 버전
rediori@localhost:project.sample $
```

옵션 설명

- `-p` 각 커밋에 적용된 패치를 보여준다.
- `--stat` 각 커밋에서 수정된 파일의 통계정보를 보여준다.
- `--shortstat` `--stat` 명령의 결과 중에서 수정한 파일, 추가된 줄, 삭제된 줄만 보여준다.
- `--name-only` 커밋 정보중에서 수정된 파일의 목록만 보여준다.
- `--name-status` 수정된 파일의 목록을 보여줄 뿐만 아니라 파일을 추가한 것인지, 수정한 것인지, 삭제한 것인지도 보여준다.
- `--abbrev-commit` 40자 짜리 SHA-1 체크섬을 전부 보여주는 것이 아니라 처음 몇 자만 보여준다.
- `--relative-date` 정확한 시간을 보여주는 것이 아니라 '2 주전'처럼 상대적인 형식으로 보여준다.
- `--graph` 브랜치와 머지 히스토리 정보까지 아스키 그래프로 보여준다.
- `--pretty` 지정한 형식으로 보여준다. 이 옵션에는 `oneline`, `short`, `full`, `fuller`, `format`이 있다. `format`은 원하는 형식으로 출력하고자 할 때 사용한다.

커밋 히스토리 조회

- 조회 제한조건
- `$ git log --since=2.weeks`
 - 2008-01-15같이 정확한 날짜도 사용할 수 있고 2 years 1 day 3 minutes ago같이 상대적인 기간을 사용할 수도 있다.
- `--author` 옵션으로 저자를 지정하여 검색
- `--grep` 옵션으로 커밋 메시지에서 키워드를 검색 (`author`와 `grep` 옵션을 나눠서 지정하고 싶지 않으면 `--all-match` 옵션으로 한 번에 검색할 수 있다).
- 파일 경로로 검색하는 옵션
 - 디렉토리나 파일 이름을 사용하여 그 파일이 변경된 log의 결과를 검색할 수 있다. 이 옵션은 `--`와 함께 경로 이름을 사용하는데 명령어 끝 부분에 쓴다(역주, `git log --path1 path2`).

옵션 설명

`-(n)` 최근 n 개의 커밋만 조회한다.

`--since`, `--after` 명시한 날짜 이후의 커밋만 검색한다.

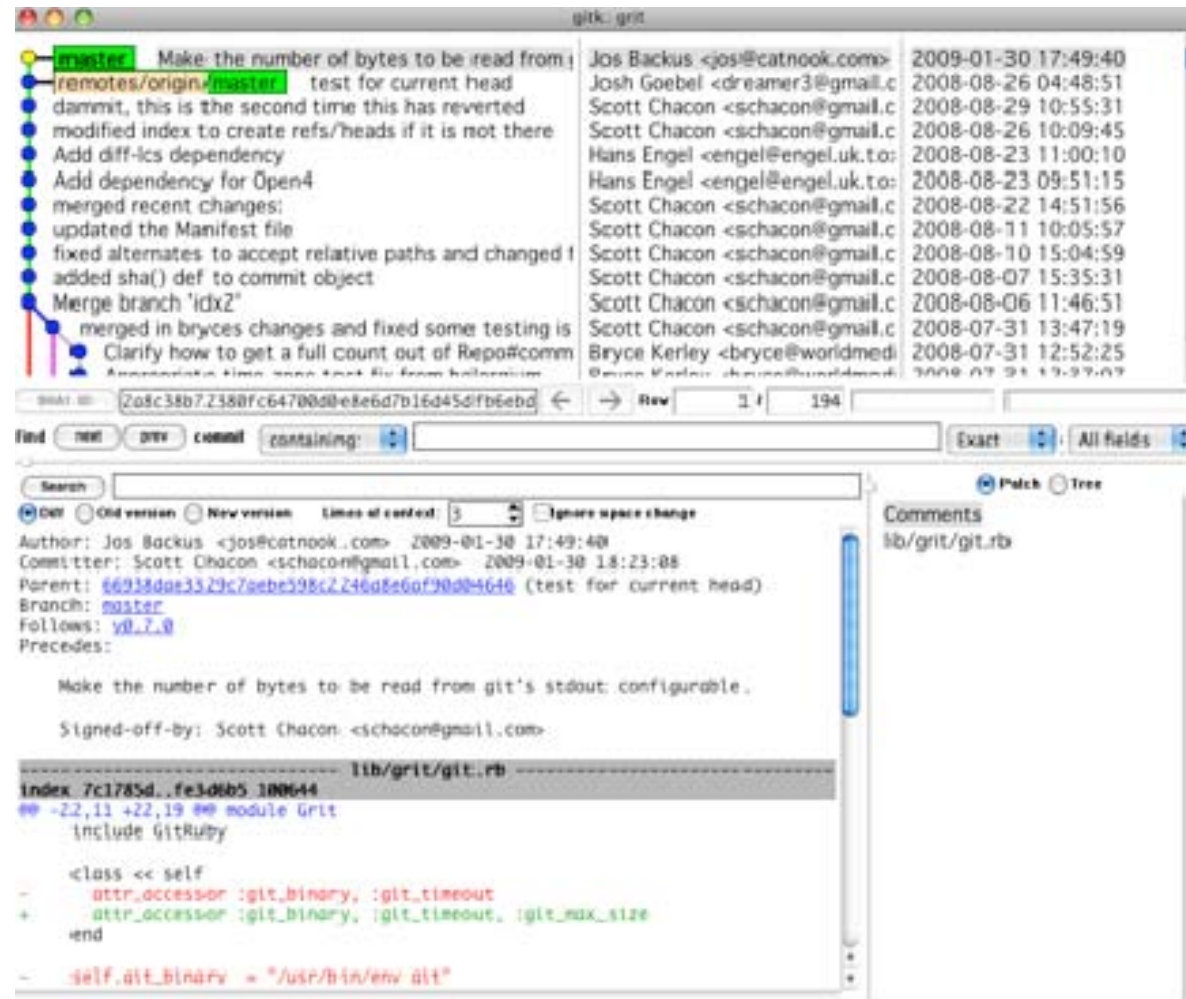
`--until`, `--before` 명시한 날짜 이전의 커밋만 조회한다.

`--author` 입력한 저자의 커밋만 보여준다.

`--committer` 입력한 커미터의 커밋만 보여준다.

git log

- yum install gitk



The screenshot shows the gitk graphical user interface. The top window displays a list of commits with their authors and timestamps. The selected commit is 2a5c38b72380fc64700d0e8e6d7b16d45dfb6ebd, committed by Scott Chacon on 2009-01-30 18:23:08. The commit message is "Merge branch 'idx2'".

The bottom window shows the diff for the selected commit, comparing the old version (7c1785d..fe3d6b5) with the new version (180644). The diff shows changes to the file lib/grit/git.rb, including the addition of the attr_accessor :git_max_size and the update of the self.git_binary path to */usr/bin/env git.

```
Author: Jos Backus <jos@catnook.com> 2009-01-30 17:49:40
Committer: Scott Chacon <schacon@gmail.com> 2009-01-30 18:23:08
Parent: 66938dae3329c7aebc988c2246a8e6af90d04646 (test for current head)
Branch: master
Follows: y0.2.0
Precedes:

    Make the number of bytes to be read from git's stdout configurable.

Signed-off-by: Scott Chacon <schacon@gmail.com>

----- lib/grit/git.rb -----
index 7c1785d..fe3d6b5 180644
@@ -22,11 +22,19 @@ module Grit
   include GitRuby

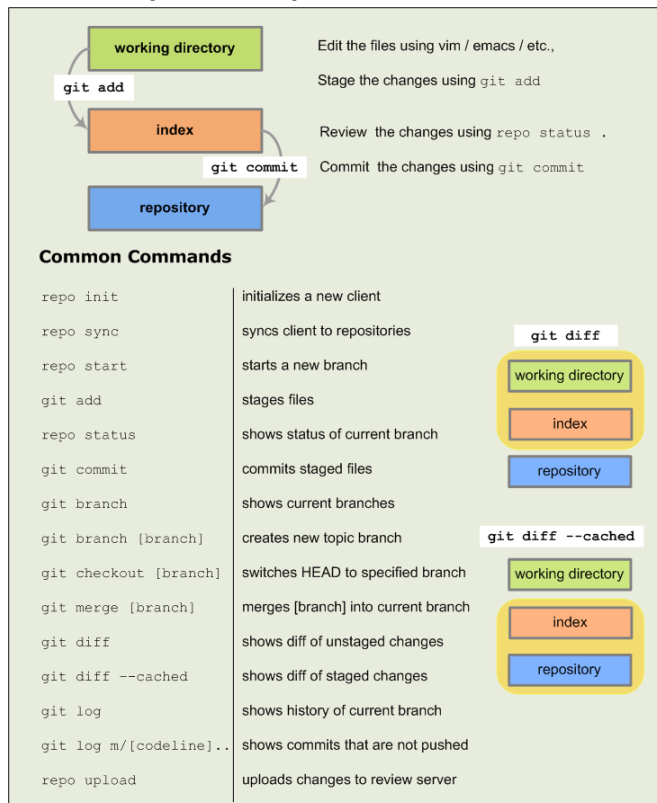
   <class << self
-     attr_accessor :git_binary, :git_timeout
+     attr_accessor :git_binary, :git_timeout, :git_max_size
   end

-   self.git_binary = */usr/bin/env git"
```



```
rediori@localhost:project.sample $ git diff
diff --git a/main.c b/main.c
index e3acaec..d551492 100644
--- a/main.c
+++ b/main.c
@@ -3,6 +3,6 @@
 int main( int argc, char* argv[])
 {
     printf("hello git\n");
-    printf("test\n");
+    //printf("test\n");
     return 0;
 }
```

- git diff : Unstaged 상태인 것들만 보여준다.
- git diff --cached HEAD : stage 영역에 있는 내용과 저장소에 있는 내용을 비교한다.



```
PS C:\Users\Faith\Desktop\Git> git diff
diff --git a/f.txt b/f.txt
index 5691884..a94366f 100644
--- a/f.txt
+++ b/f.txt
@@ -1,2 @@
-This is a new text.
\# No newline at end of file
+This is a new text.
+This is really good for Git.
\# No newline at end of file
diff --git a/text.txt b/text.txt
index 515168f..6fac708 100644
--- a/text.txt
+++ b/text.txt
@@ -1,3 +1,4 @@
 This is a new text file
 Added a new line.
-Added an another line.
\# No newline at end of file
+Added an another line.
+Hello Git World!
\# No newline at end of file
warning: LF will be replaced by CRLF in f.txt.
The file will have its original line endings in your working directory.
warning: LF will be replaced by CRLF in text.txt.
The file will have its original line endings in your working directory.
```

```
PS C:\Users\Faith\Desktop\Git> git diff --cached HEAD
diff --git a/f.txt b/f.txt
index 5691884..a94366f 100644
--- a/f.txt
+++ b/f.txt
@@ -1,2 @@
-This is a new text.
\# No newline at end of file
+This is a new text.
+This is really good for Git.
\# No newline at end of file
warning: LF will be replaced by CRLF in f.txt.
The file will have its original line endings in your working directory.
```

- `git diff --stat` : 현재 무엇이 얼마나 변경되었는지 보여줍니다.

```
text.txt | 3 +--  
1 files changed, 2 insertions(+), 1 deletions(-)  
warning: LF will be replaced by CRLF in text.txt.  
The file will have its original line endings in your working directory.
```

text.txt가 세 줄 변경되었음을 알 수 있습니다.

- **git blame 파일명**

누가 어디를 얼마나 수정했는지 보여줍니다.

```
PS C:\Users\Faith\Desktop\Git> git blame text.txt  
7a0fee88 (sainthkh 2011-10-01 18:46:56 +0900 1) This is a new text file  
7a0fee88 (sainthkh 2011-10-01 18:46:56 +0900 2) Added a new line.  
00000000 (Not Committed Yet 2011-10-01 19:07:08 +0900 3) Added an another line.  
00000000 (Not Committed Yet 2011-10-01 19:07:08 +0900 4) Hello Git World!
```

정리

```
$ git configure --global user.name "USERNAME" #계정의 유저 이름 설정
$ git configure --global user.email user@email.com #계정의 유저 메일 설정

$ git init      # 저장소 만들기
$ git status   # 상태 보기
$ git add      # 파일 추가(Staging Area로)
$ git commit   # Staging Area에 있는 파일들을 스냅샷으로 저장
    • -a -m    # -a : 변경된 파일을 자동으로 추가 후 commit
                # -m : ㄱ
$ git log      # 커밋 히스토리 보기
$ git rm [file-name] # 파일 삭제
$ git mv [file-from] [file-to] #파일 이름 변경
```

git undoing



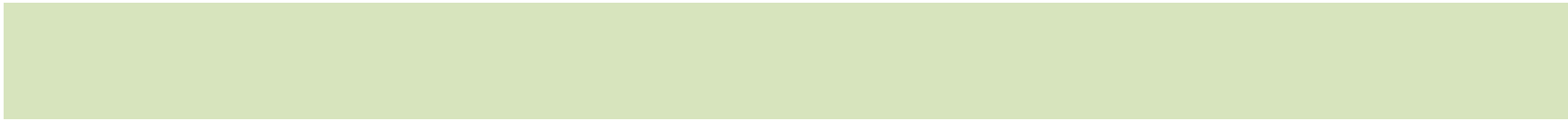
커밋 수정

- **\$ git commit --amend**
 - 이 명령은 Staging Area를 사용하여 커밋한다. 만약 마지막으로 커밋하고 나서 수정한 것이 없다면 (커밋하자마자 바로 이 명령을 실행하는 경우) 조금 전에 한 커밋과 모든 것이 같다. 이때는 커밋 메시지만 수정한다.
 - 편집기가 실행되면 이전 커밋 메시지가 자동으로 포함된다. 메시지를 수정하지 않고 그대로 커밋해도 기존의 커밋을 덮어쓴다.
 - 커밋을 했는데 Stage하는 것을 깜빡하고 빠트린 파일이 있으면 아래와 같이 고칠 수 있다:
 - \$ git commit -m 'initial commit'
 - \$ git add forgotten_file
 - \$ git commit --amend
 - 여기서 실행한 명령어 3개는 모두 하나의 커밋으로 기록된다. 두 번째 커밋은 첫 번째 커밋을 덮어쓴다
-

파일을 Unstage로 변경

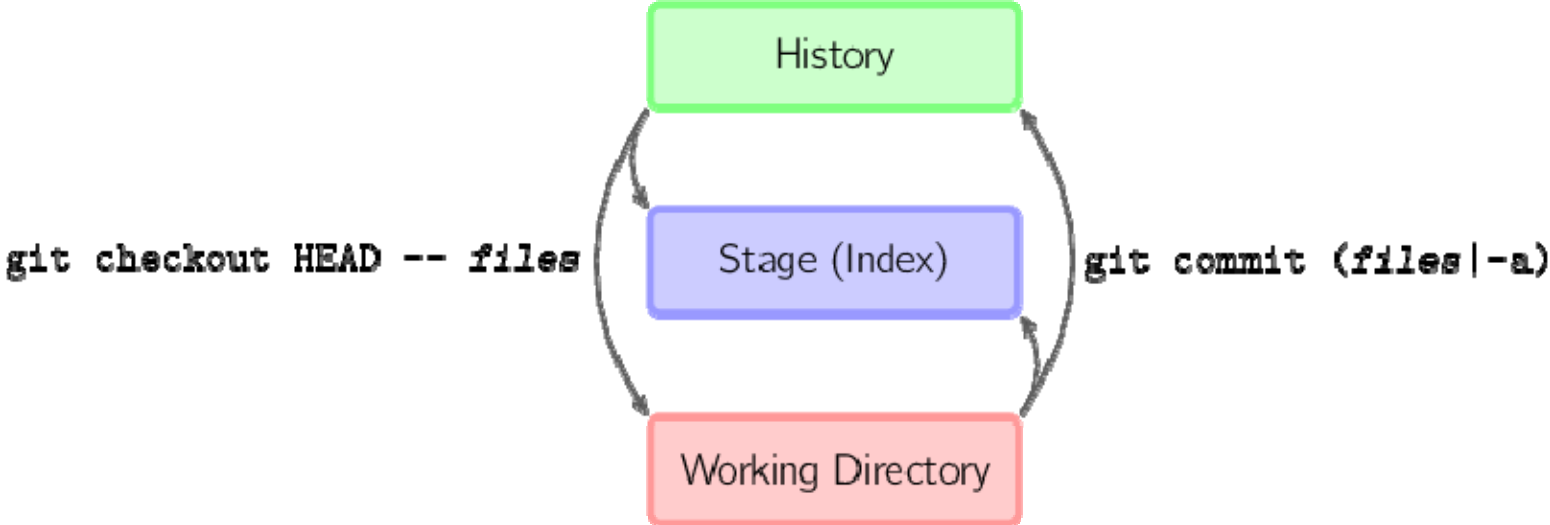
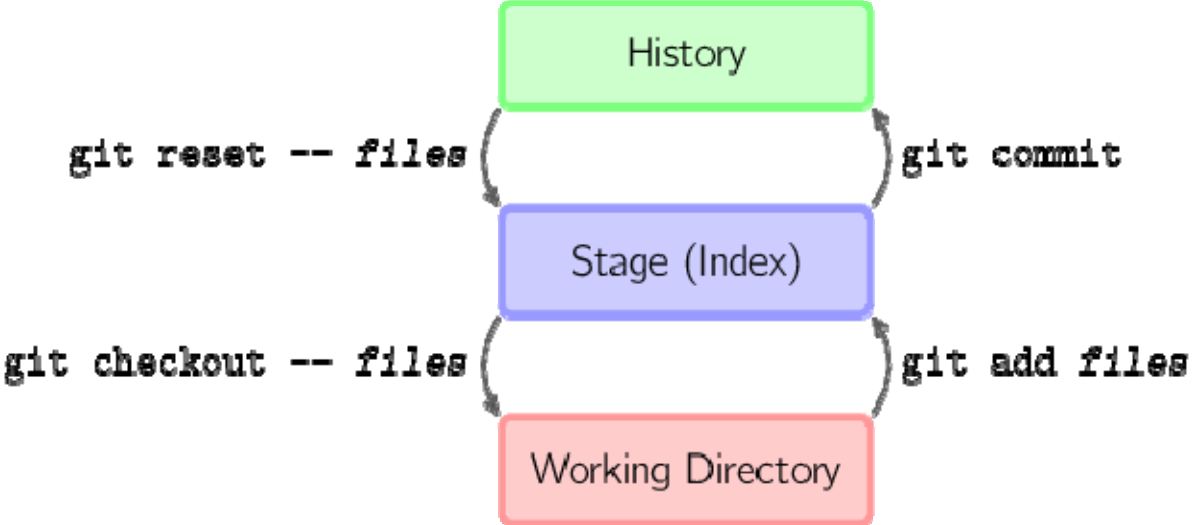
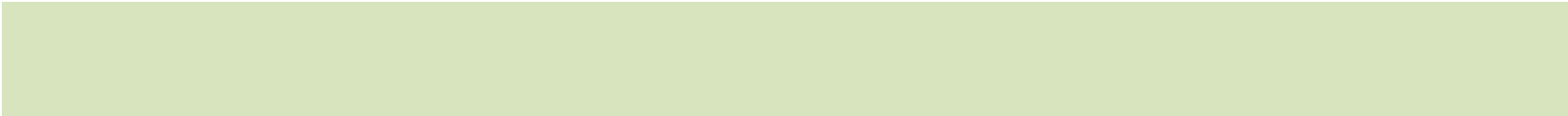
- `$ git add .`
 - `$ git status`
 - `# On branch master`
 - `# Changes to be committed:`
 - `# (use "git reset HEAD <file>..." to unstage)`
 - `#`
 - `# modified: README.txt`
 - `# modified: benchmarks.rb`
 - `#`

 - `$ git reset HEAD benchmarks.rb`
 - `benchmarks.rb: locally modified`
-



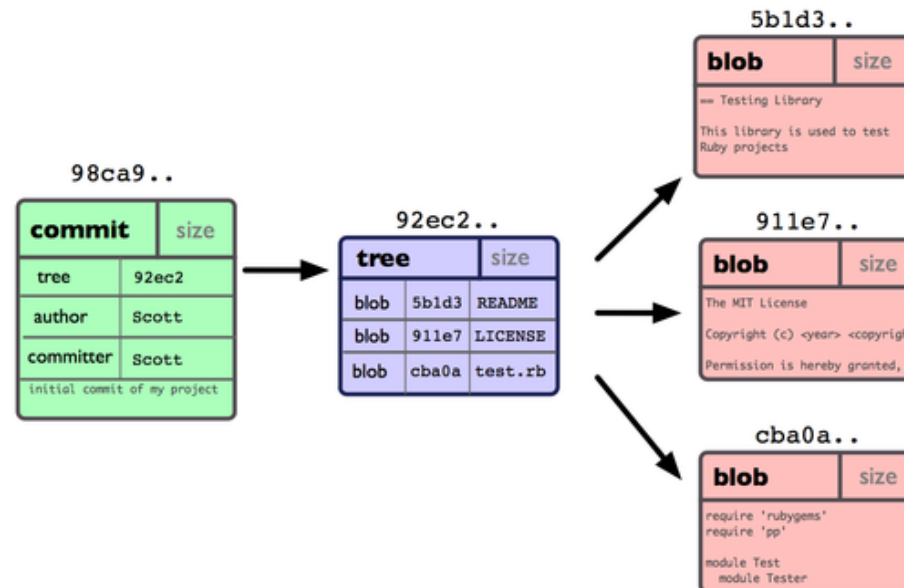
수정 파일 되돌리기

- `$ git status`
 - `# On branch master`
 - `# Changes to be committed:`
 - `# (use "git reset HEAD <file>..." to unstage)`
 - `#`
 - `# modified: README.txt`
 - `#`
 - `# Changes not staged for commit:`
 - `# (use "git add <file>..." to update what will be committed)`
 - `# (use "git checkout -- <file>..." to discard changes in working directory)`
 - `#`
 - `# modified: benchmarks.rb`
 - `#`
 - `$ git checkout -- benchmarks.rb`
-

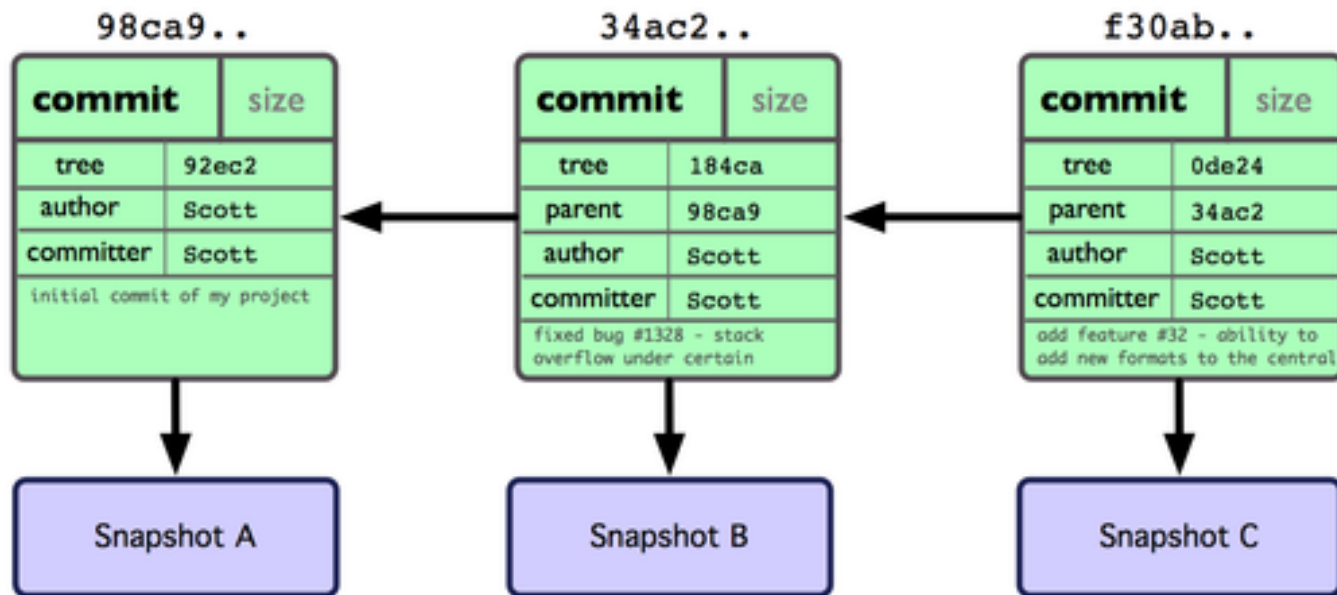


git branch

- git은 데이터를 스냅샷으로 기록
 - Change set이나 변경사항으로 기록하지 않는다
- commit :메타데이터와 루트 트리를 가리키는 포인터가 담긴 커밋 개체
- tree : 파일과 디렉토리 구조가 들어 있는 트리 개체
- blob : git 저장소에 있는 Staged 된 파일

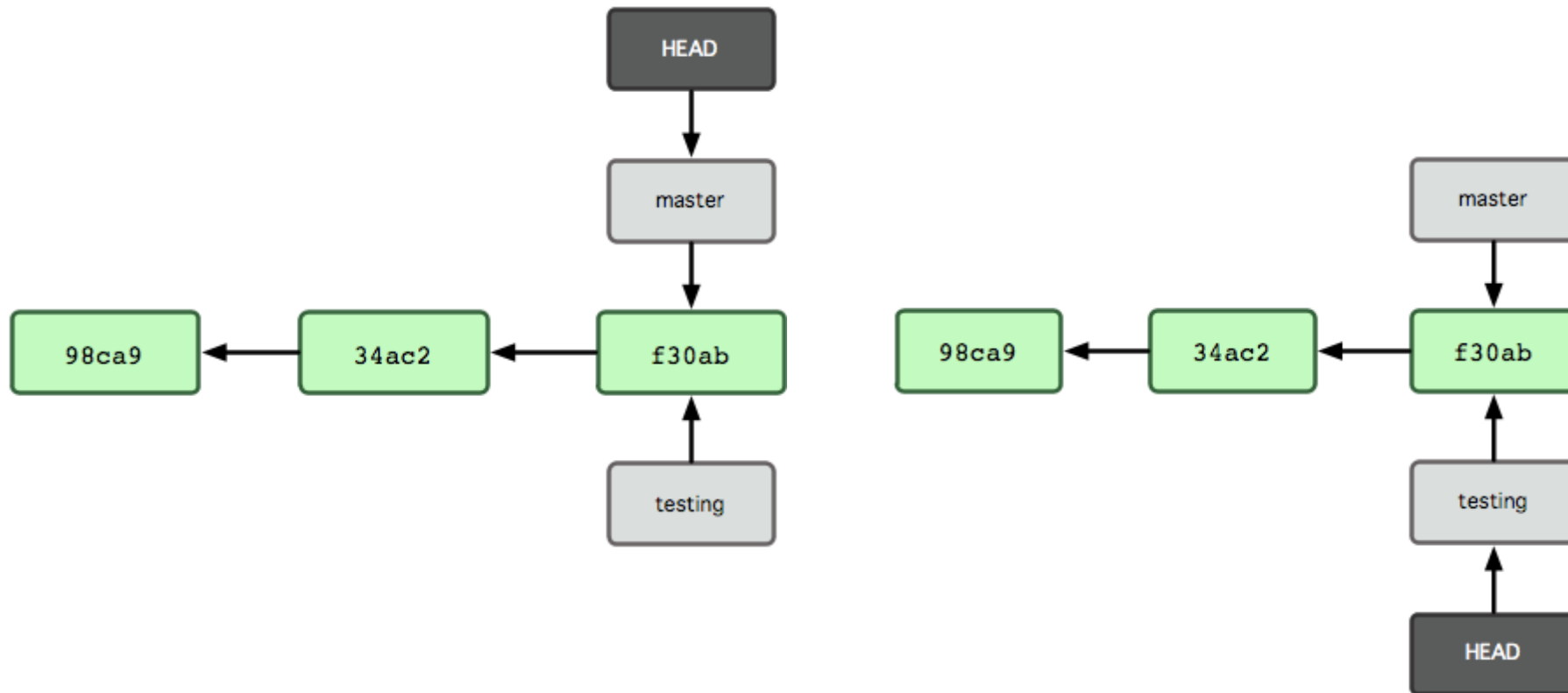


Commits

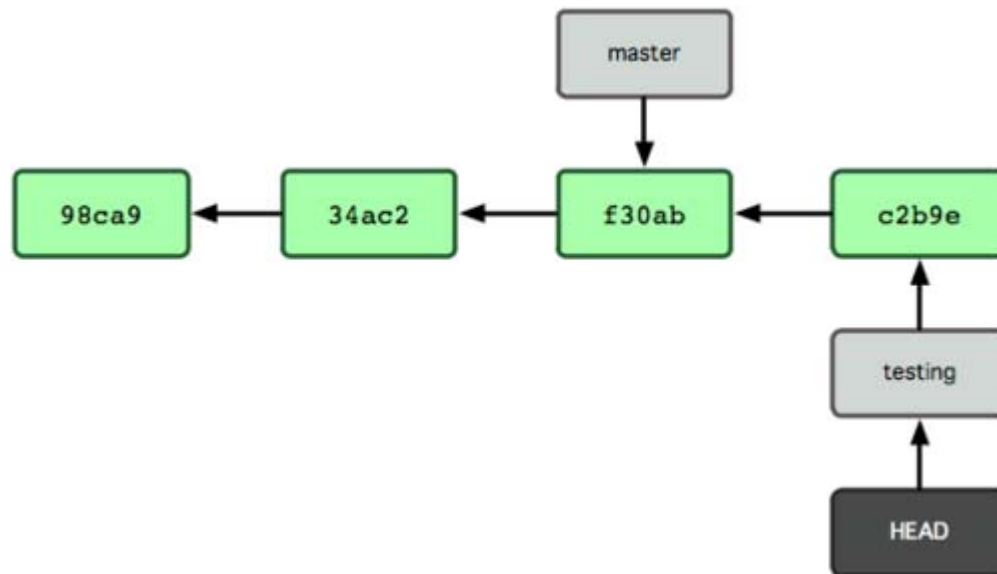


Make branch

- \$git branch testing
- \$git checkout testing



- `git commit -a -m 'change branch'`



branch

```
rediori@localhost:project.sample $ git branch
```

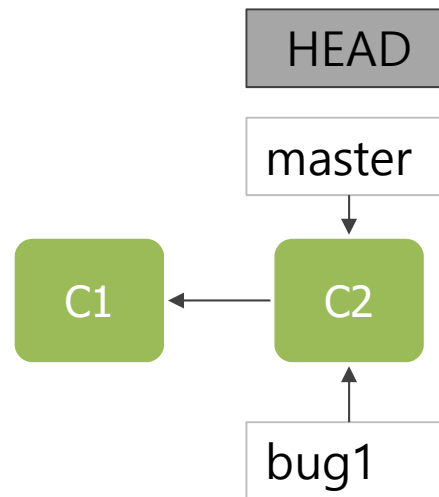
```
* master
```

```
rediori@localhost:project.sample $ git branch bug1
```

```
rediori@localhost:project.sample $ git branch
```

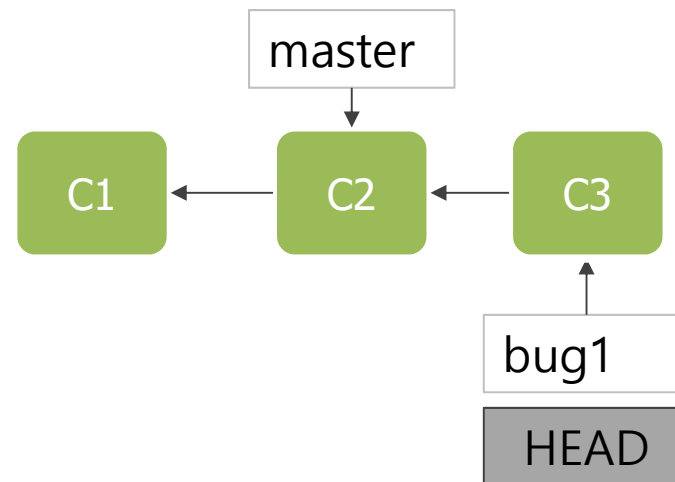
```
* master
```

```
bug1
```



branch

```
rediori@localhost:project.sample $ git checkout bug1
Switched to branch 'bug1'
rediori@localhost:project.sample $ vi main.c
rediori@localhost:project.sample $ git commit -a -m 'branch testing'
[testing fcd1fe9] branch testing
1 files changed, 1 insertions(+), 0 deletions(-)
```



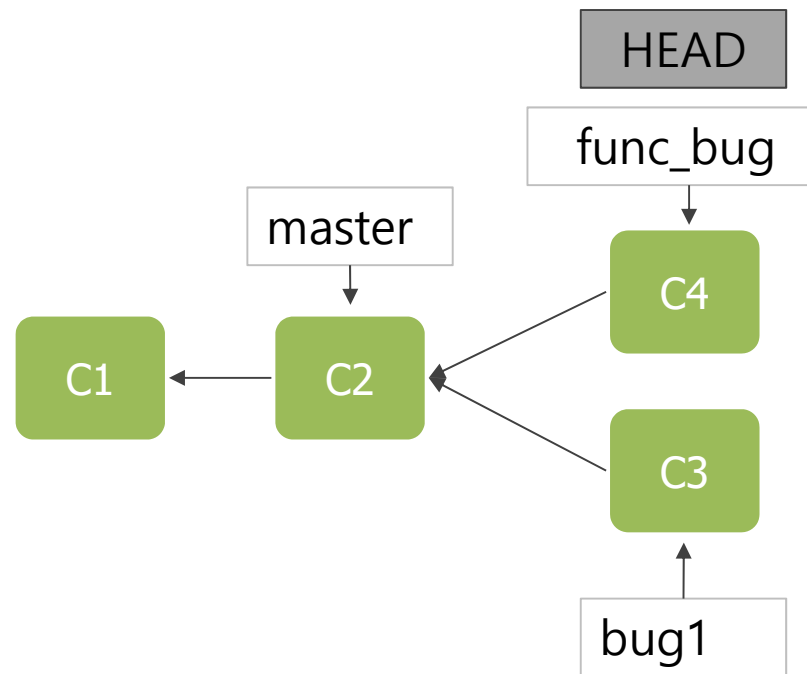
branch

```
rediori@localhost:project.sample $ git checkout master
```

```
Switched to branch 'master'
```

```
rediori@localhost:project.sample $ git checkout -b 'func_bug'
```

```
Switched to a new branch 'func_bug'[master 584d084] branch master  
1 files changed, 1 insertions(+), 0 deletions(-)
```



Fast forward merge

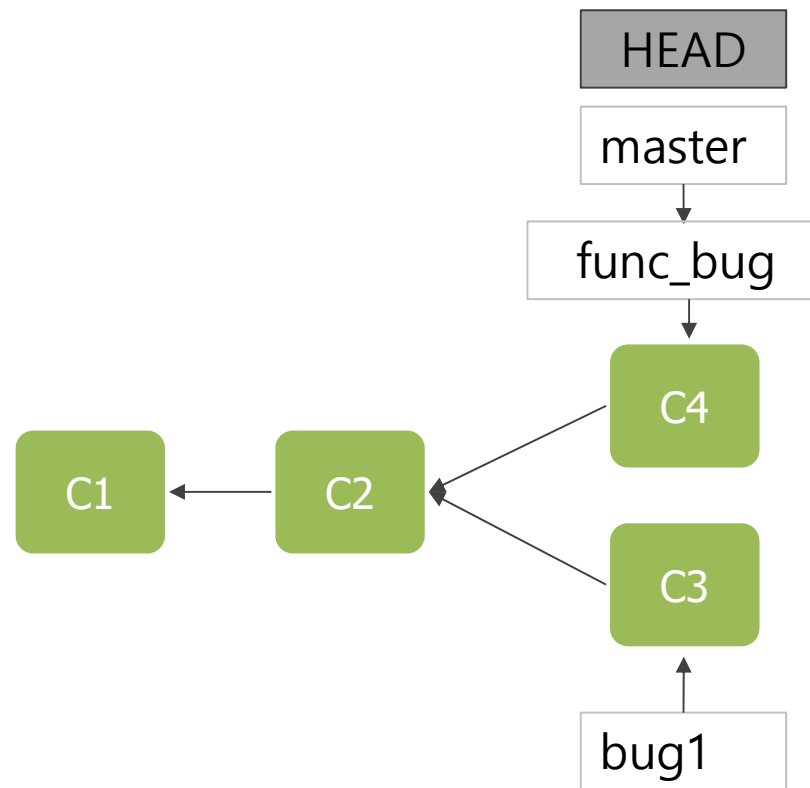
```
rediori@localhost:project.sample $ git merge func_bug
```

```
Updating 4382572..5df6b9d
```

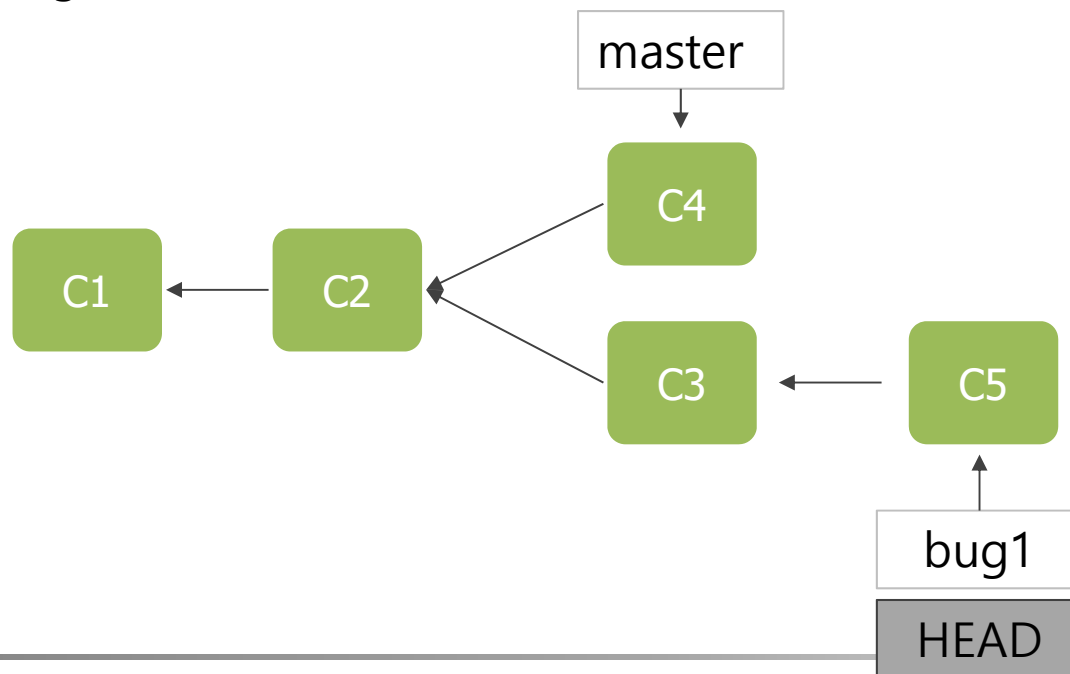
Fast-forward

```
func1.c | 1 +
```

```
1 files changed, 1 insertions(+), 0 deletions(-)
```

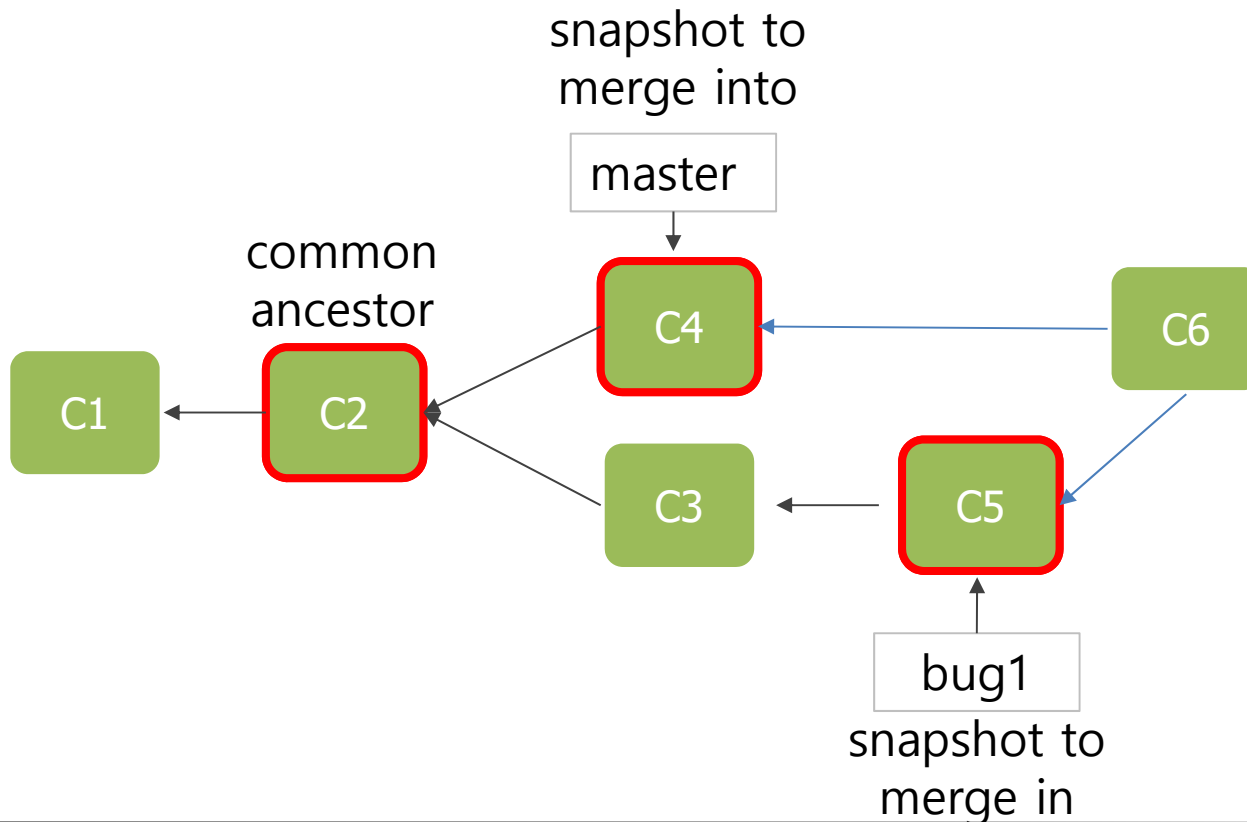


```
$ git branch -d func_bug
Deleted branch func_bug (was 5df6b9d).
$ git branch
bug1
* master
$ git checkout bug1
vi main.c
$ git commit -a -m 'finish bug1'
[bug1 628ca7c] finish bug1
1 files changed, 1 insertions(+), 0 deletions(-)
```



3-Way Merge

```
$ git merge bug1
Merge made by recursive.
 README | 1 +
 main.c | 3 ++-
 2 files changed, 3 insertions(+), 1 deletions(-)
$ git branch -d bug1
```



Merge Crash

- bug, func_bug 둘 다 main.c 를 수정 했을 경우
 - `$ git merge func_bug`
 - Updating 3ab0487..13e6525
 - Fast-forward
 - main.c | 1 +
 - 1 files changed, 1 insertions(+), 0 deletions(-)
 - `$ git merge bug1`
 - Auto-merging main.c
 - CONFLICT (content): Merge conflict in main.c
 - Automatic merge failed; fix conflicts and then commit the result.
-

Merge Crash

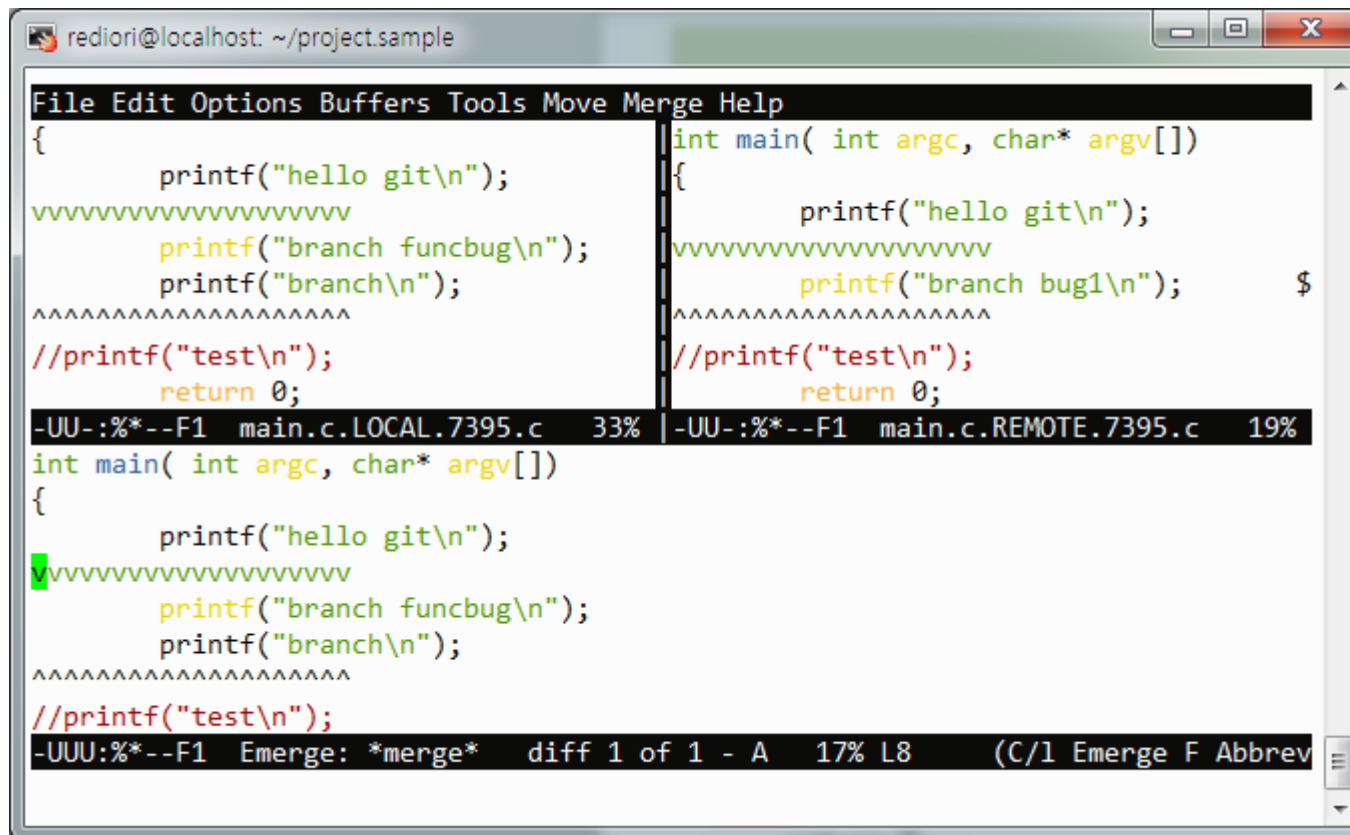
- **main.c**

```
#include <stdio.h>
// Main Function
```

```
int main( int argc, char* argv[])
{
    printf("hello git\n");
    <<<<<<< HEAD
    printf("branch funcbug\n");
    printf("branch\n");
    =====
    printf("branch bug1\n");
    >>>>>>> bug1
    //printf("test\n");
    return 0;
}
```

git mergetool

- \$git mergetool



The screenshot shows a diff tool window titled "rediori@localhost: ~/project.sample". The window displays a diff between two versions of a C file named "main.c". The top part of the window shows the diff output, with a vertical line indicating the merge point. The diff shows that the local version (LOCAL.7395.c) has a "branch funcbug" and "branch" message, while the remote version (REMOTE.7395.c) has a "branch bug1" message. The diff tool is currently showing the local version's code, which has been merged into the remote version's code. The status bar at the bottom indicates "diff 1 of 1 - A 17% L8 (C/1 Emerge F Abbrev".

```
File Edit Options Buffers Tools Move Merge Help
{
    printf("hello git\n");
vvvvvvvvvvvvvvvvvvvv
    printf("branch funcbug\n");
    printf("branch\n");
^^^^^^^^^^^^^^^^^^^^
//printf("test\n");
    return 0;
-UU-:~*--F1 main.c.LOCAL.7395.c 33% | -UU-:~*--F1 main.c.REMOTE.7395.c 19%
int main( int argc, char* argv[])
{
    printf("hello git\n");
vvvvvvvvvvvvvvvvvvvv
    printf("branch funcbug\n");
    printf("branch\n");
^^^^^^^^^^^^^^^^^^^^
//printf("test\n");
-UUU:~*--F1 Emerge: *merge* diff 1 of 1 - A 17% L8 (C/1 Emerge F Abbrev
```

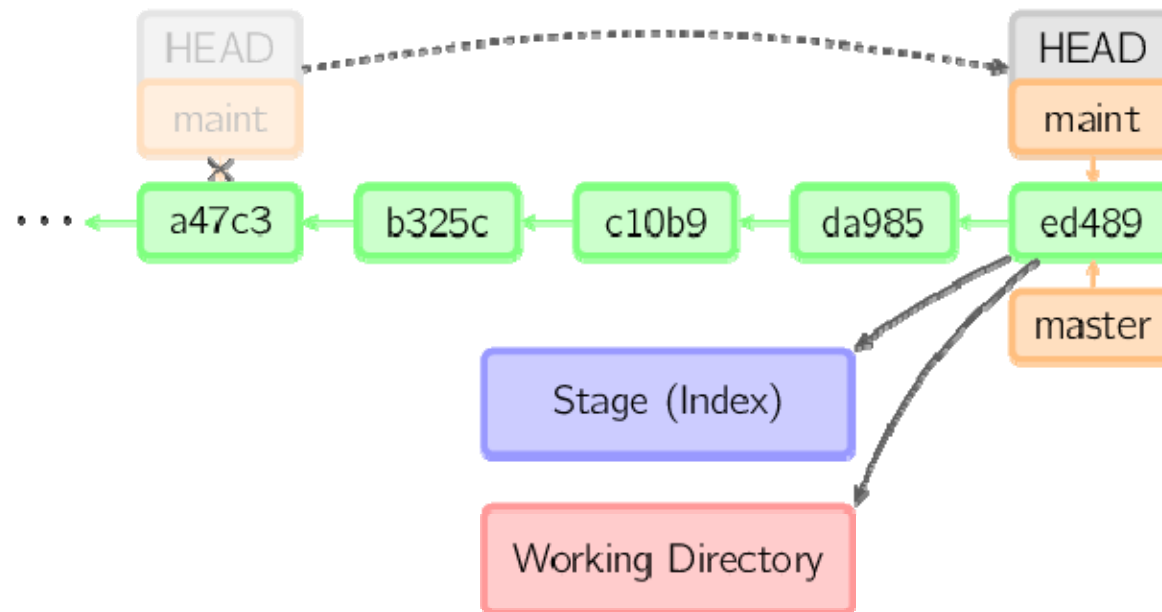

git branch

- `$git branch` : 브랜치 목록(*기호가 붙어 있는 브랜치가 현재 checkout해서 작업중인 브랜치)
 - `$git branch -v` : 커밋 메시지 함께 출력
 - `$git branch -merged` : merge된 branch
 - `$git branch -no-merged` : merge되지 않은 branch
-

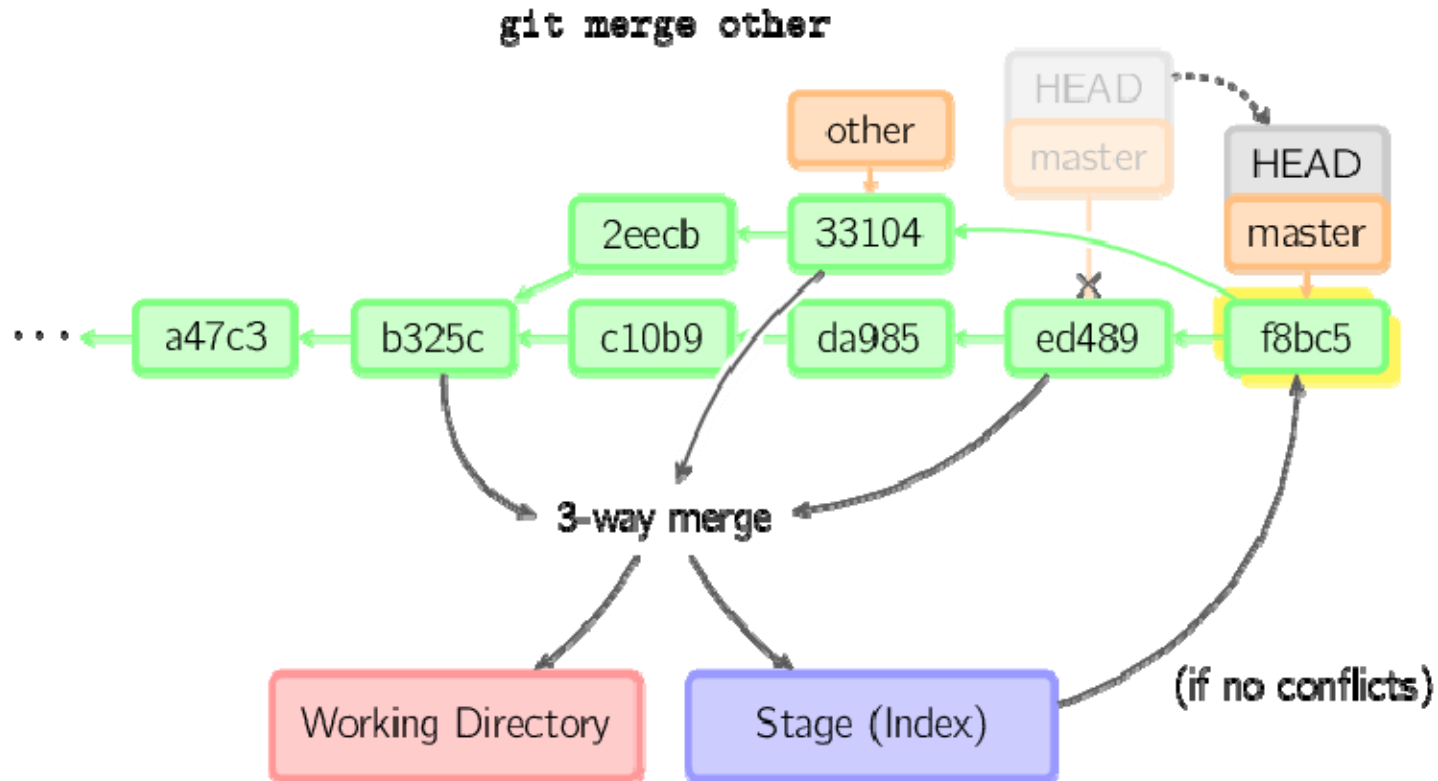
- **git branch [branch name]**
 - **git checkout [branch name]**
 - **git checkout -b [branch name]**
 - git branch [branch name] ; git checkout [branch name] ;
 - **git merge [branch name]**
-

FF Merge

`git merge master`



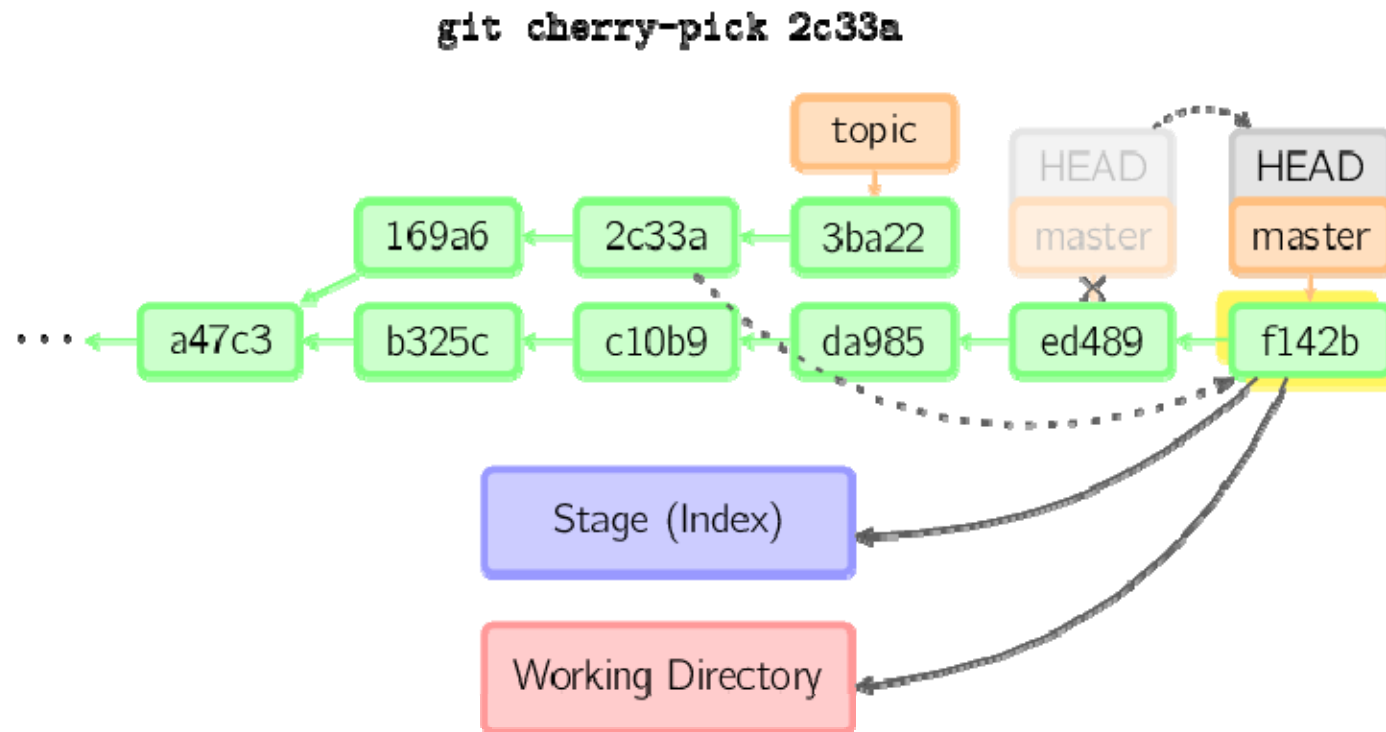
3-way Merge



cherry-pick

- **git cherry-pick**

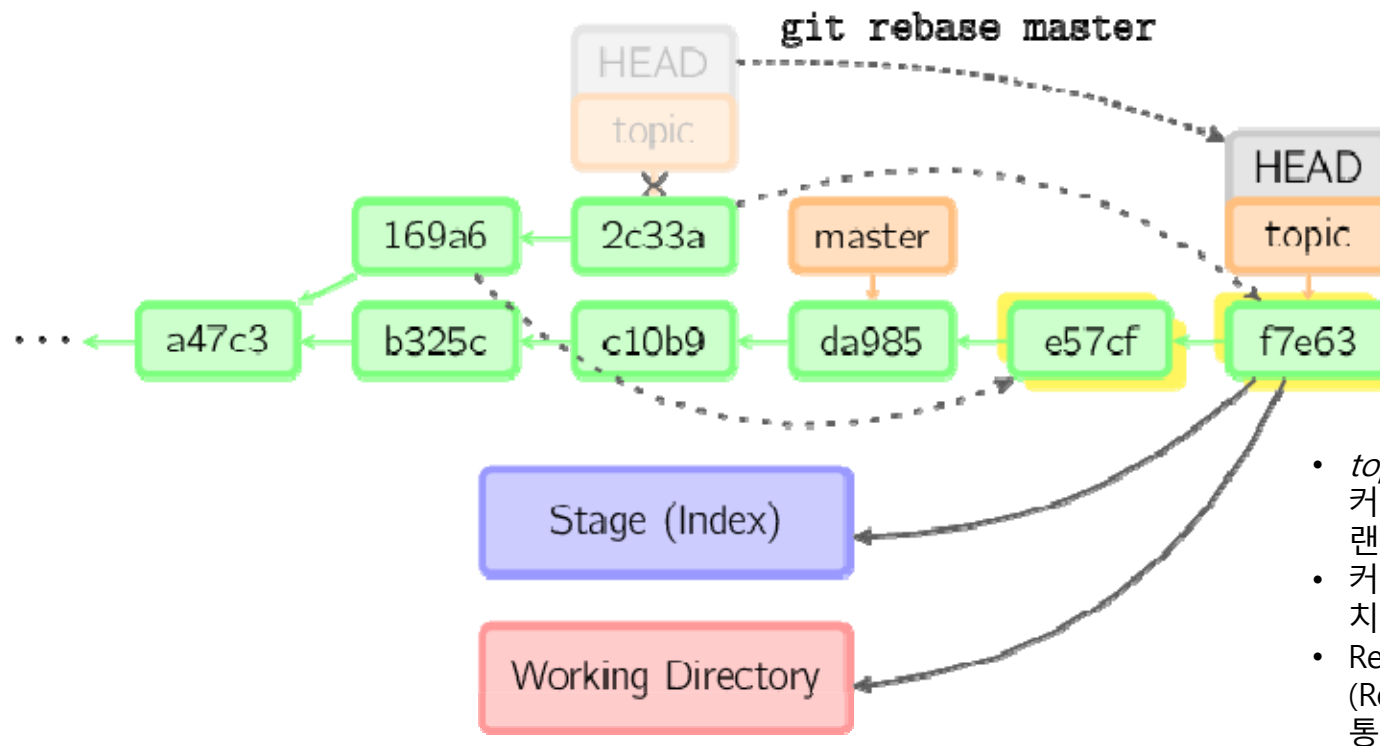
- Cherry-pick 명령은 커밋을 하나 꺼내서 현재 작업중인 브랜치 마지막 부분에 '복사'를 하면서 해당 커밋이 변경하는 부분을 적용하고 메시지나 저자 정보 등의 커밋 정보를 함께 저장



rebase

- **git rebase**

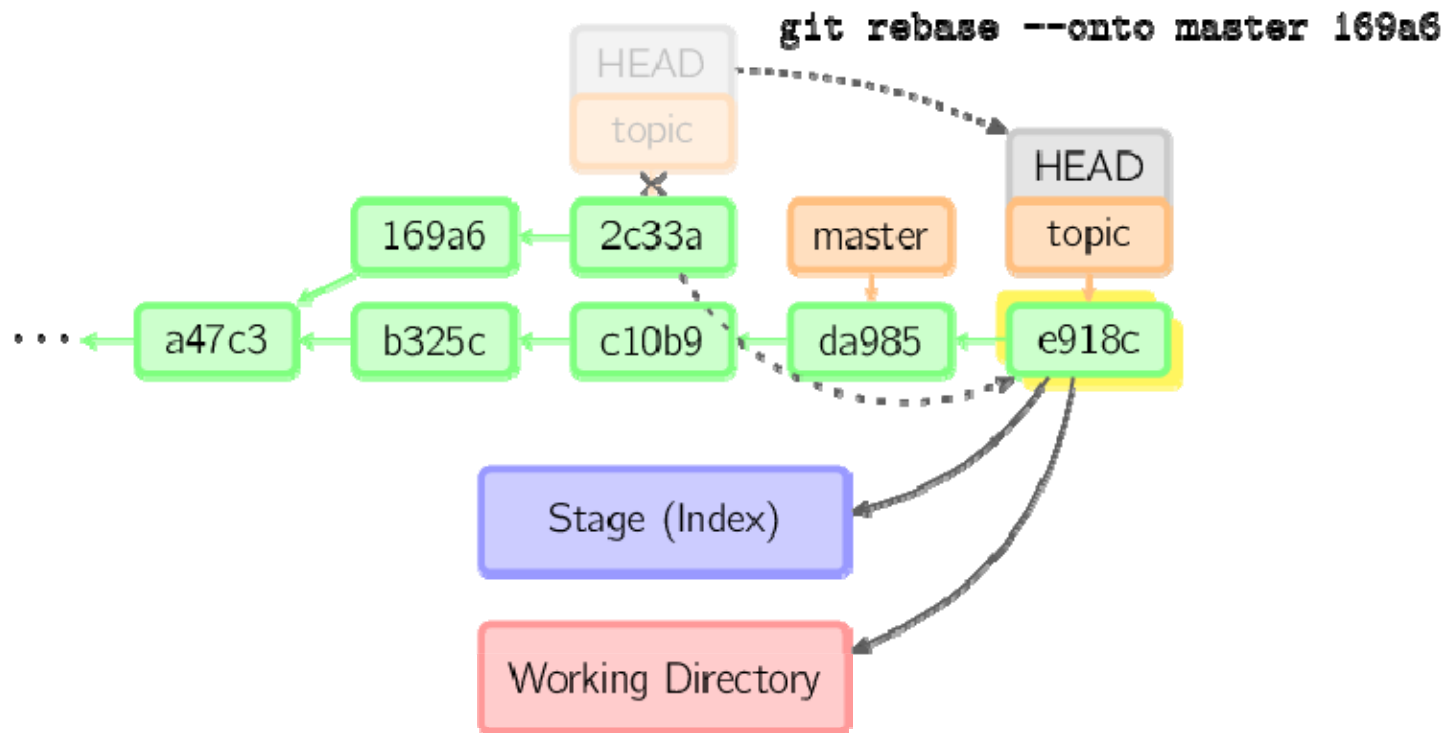
- 여러 브랜치를 하나로 모으고자 할 때 Rebase 명령을 Merge 명령 대신 사용
- Merge 명령은 두 부모를 가지는 하나의 새 커밋을 만들기 때문에 히스토리가 직선적이지 않음.
- Rebase 명령을 사용하면 커밋들을 하나씩 순차적으로 적용해나가면서 히스토리를 직선으로 만듦.
- Cherry-pick 명령을 자동으로 한번에 수행하는 것



- *topic* 브랜치에만 포함되어 있는 모든 커밋들(169a6와 2c33a)을 *master* 브랜치에 추가.
- 커밋들을 추가하고 나서 *master* 브랜치가 마지막 커밋을 가리키도록 이동
- Rebase하고 나서 더 이상 가리킬 (Reference) 수 없는 커밋들은 쓰레기 통으로 삭제

rebase

- --onto 옵션을 사용하면 Rebase에 사용할 커밋을 얼마나 오래 전까지의 커밋을 사용할 지 제한
- 아래 명령은 169a6 커밋 이후의 모든 커밋들(여기에서는 2c33a 커밋)을 master 브랜치에 적용



git 복구

특정 리비전으로 갔다가 되돌아 오기

- **git checkout HEAD~1**
- **git checkout [CHECKSUM]**
 - 6자리 정도만 입력해도 충분

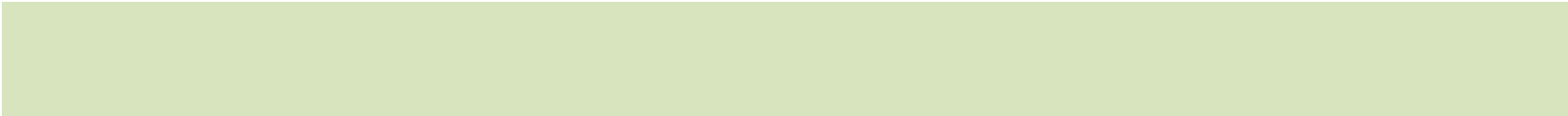
```
rediori@localhost:project.sample $ git log --pretty=oneline
403978ebf913ec9c60163439d12c7efa9183c091 fix conflict
13e6525103bf774eae851b89975cb352510968ca func bug fix
95bb952e3ea7dec4e7614ecb727a001aeb739a1f bug1 fix
3ab0487039b65aecffc647c7dd1be8d2c9e9dfbc c1
rediori@localhost:project.sample $ git checkout 3ab0
Note: checking out '3ab0'.
```

You are in 'detached HEAD' state. You can look around, make experimental changes and commit them, and you can discard any commits you make in this state without impacting any branches by performing another checkout.

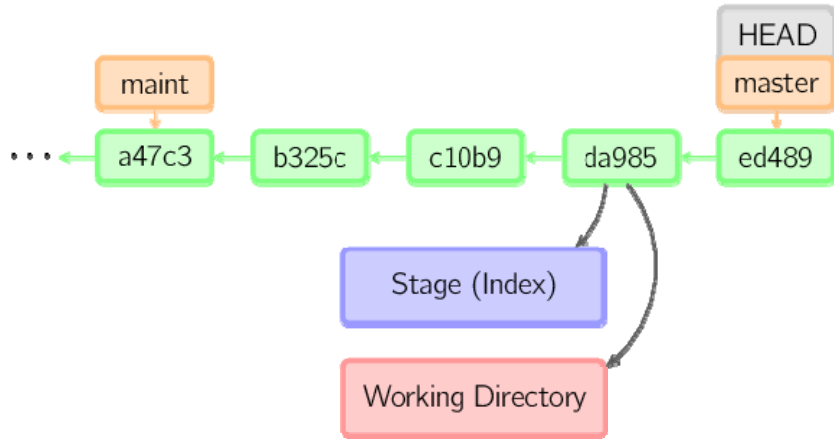
If you want to create a new branch to retain commits you create, you may do so (now or later) by using `-b` with the checkout command again. Example:

```
git checkout -b new_branch_name
```

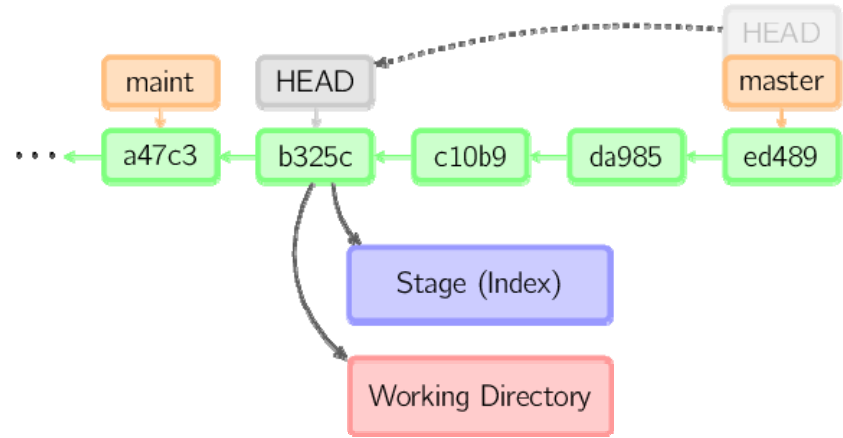
```
HEAD is now at 3ab0487... c1
```



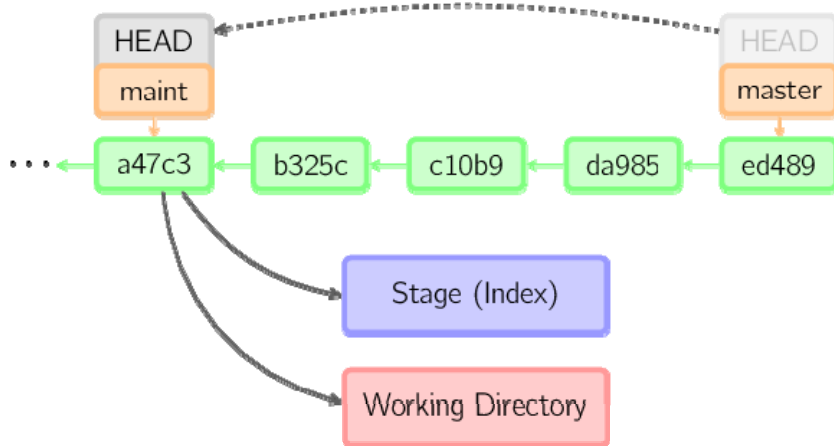
`git checkout HEAD^ files`



`git checkout master^3`



`git checkout maint`



git server

- **원격 저장소(Remote Repository)**
 - 워킹 디렉토리가 없는 Bare 저장소
 - 협업용이기 때문에 checkout이 필요 없다. (git 데이터만 필요)
 - 즉, Bare 저장소는 .git 디렉토리만 있는 저장소

 - **사용할 수 있는 프로토콜**
 - Local, SSH, git, HTTP
-

- **사용가능한 프로토콜**
 - Local, git, ssh, http
 - **Local**
 - 단순, 설정 쉬움, 팀 전체가 접근 가능한 파일시스템이 있을 경우 쉽게 구성 가능
 - git clone <file:///opt/git/project>
 - **ssh**
 - 설정이 편하며, 익숙
 - 익명 접근 불가(오픈소스에서 사용불가)
 - **git**
 - 9418 포트를 사용하며 ssh 프로토콜과 비슷하지만 인증 메커니즘이 없음
 - 가장 빠른 전송속도
-

- 기존 저장소를 Bare 저장소로 변경
 - git init, git init --bare
 - --shared 옵션을 쓸 경우, 자동으로 그룹쓰기 권한 추가
 - git clone --bare [git 저장소 경로] [Bare 저장소 경로]
 - 원격 저장소에서 가져오기
 - git clone [user]@[IP]:[DIRECTORY]
 - Public key 필요
 - git clone git://[IP]/[DIR]
-

- **Make Public Key**

- 사용자 계정의 “.ssh” 디렉토리에 저장되어야 함

User

Server

1. key 생성
 - (ssh-keygen)
 2. Server 로 Key값 전송
 - `scp ~/.ssh/id_rsa 계정@서버IP:/home/계정/.ssh`
 3. key값을 계정의 `.ssh/authorized_keys` 에 저장
 - `cat ~/.ssh/id_rsa >> ~/.ssh/authorized keys`
 4. ssh 로 인증 없이 로그인
-

Remote Repository

- 저장소를 Bare 저장소로 만들어야 한다
 - 기존의 저장소를 remote 저장소로 Clone
 - 기존의 저장소 경로 : /home/git/local
 - `$git clone --bare /home/git/local /home/git/remote.git`
 - 처음 부터 remote 저장소로 설정
 - `$git init --bare [directory]`
 - Ex) `git init --bare remote.git`
 - 이후 Client에서 push를 하게 될 경우, 다음과 같은 추가 설정 필요
 - `git config push.default tracking`
-

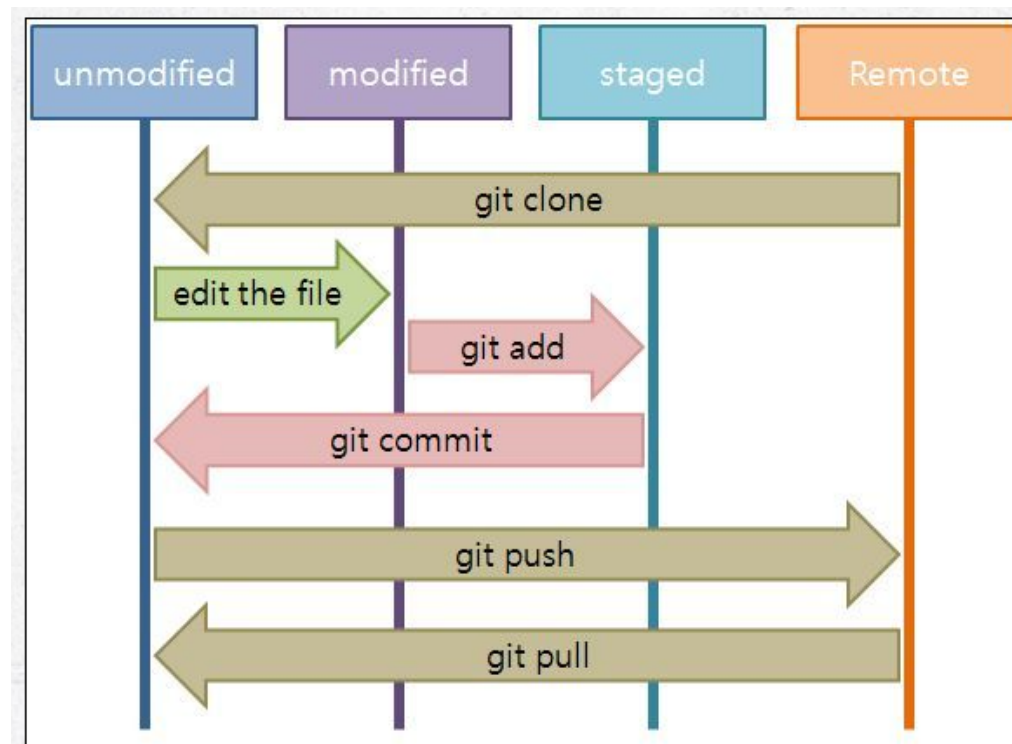
- 보안을 위해서 셸기능을 제한
 - `vim /etc/passwd`
 - `-- git:x:1000:1000::/home/git:/bin/sh`
 - `++ git:x:1000:1000::/home/git:/usr/bin/git-shell`
 - 이제 ssh를 이용해서 접속을 시도하면 서버가 아래와 같은 메시지를 출력하면서 거부한다.
 - `$ ssh git@gitserver`
 - `fatal: What do you think I am? A shell?`
 - `Connection to gitserver closed.`
-

- 2가지 방법 (ssh, git daemon)
 - **ssh protocol**
 - \$git clone [git@220.149.250.103:/home/git/remote.git](ssh://git@220.149.250.103:/home/git/remote.git)
 - **git protocol**
 - \$git clone git://220.149.250.103/remote.git
-

- **git daemon 설치**
 - `$yum install git-daemon`
 - **git daemon 돌릴 디렉토리에 파일 생성**
 - `touch [directory]/git-daemon-export-ok`
 - **push가 가능하도록 설정 변경**
 - `vi config`
 - [daemon]
 - `receivepack = true`
 - **git daemon 실행**
 - `$git daemon --reuseaddr --base-path=/home/git`
 - Ex) remote repo dir : `/home/git/remote.git`
 - `git clone git://[IP]/remote`
-

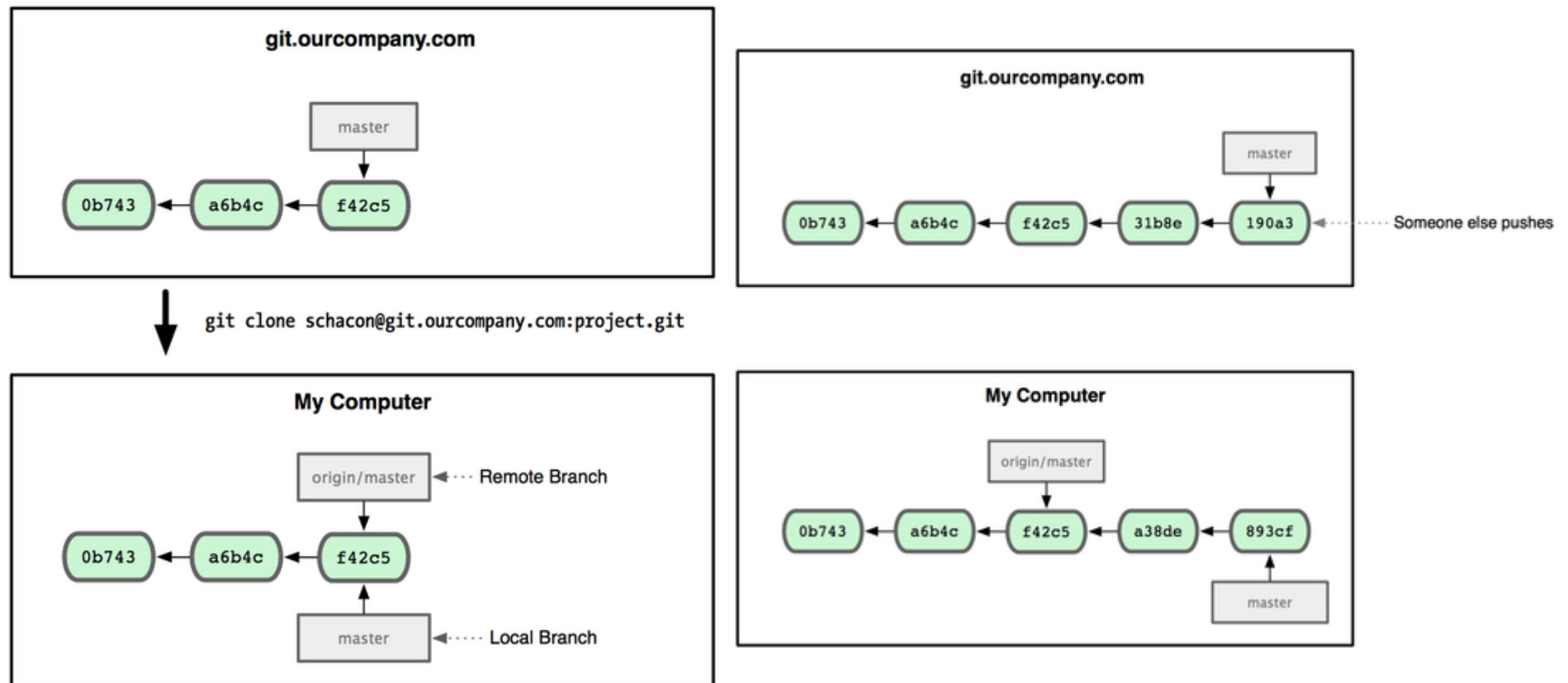
- 프로젝트 시작

- 로컬에서 관리하며 시작 (git init)
- 서버로부터 가져오면서 시작
 - git clone



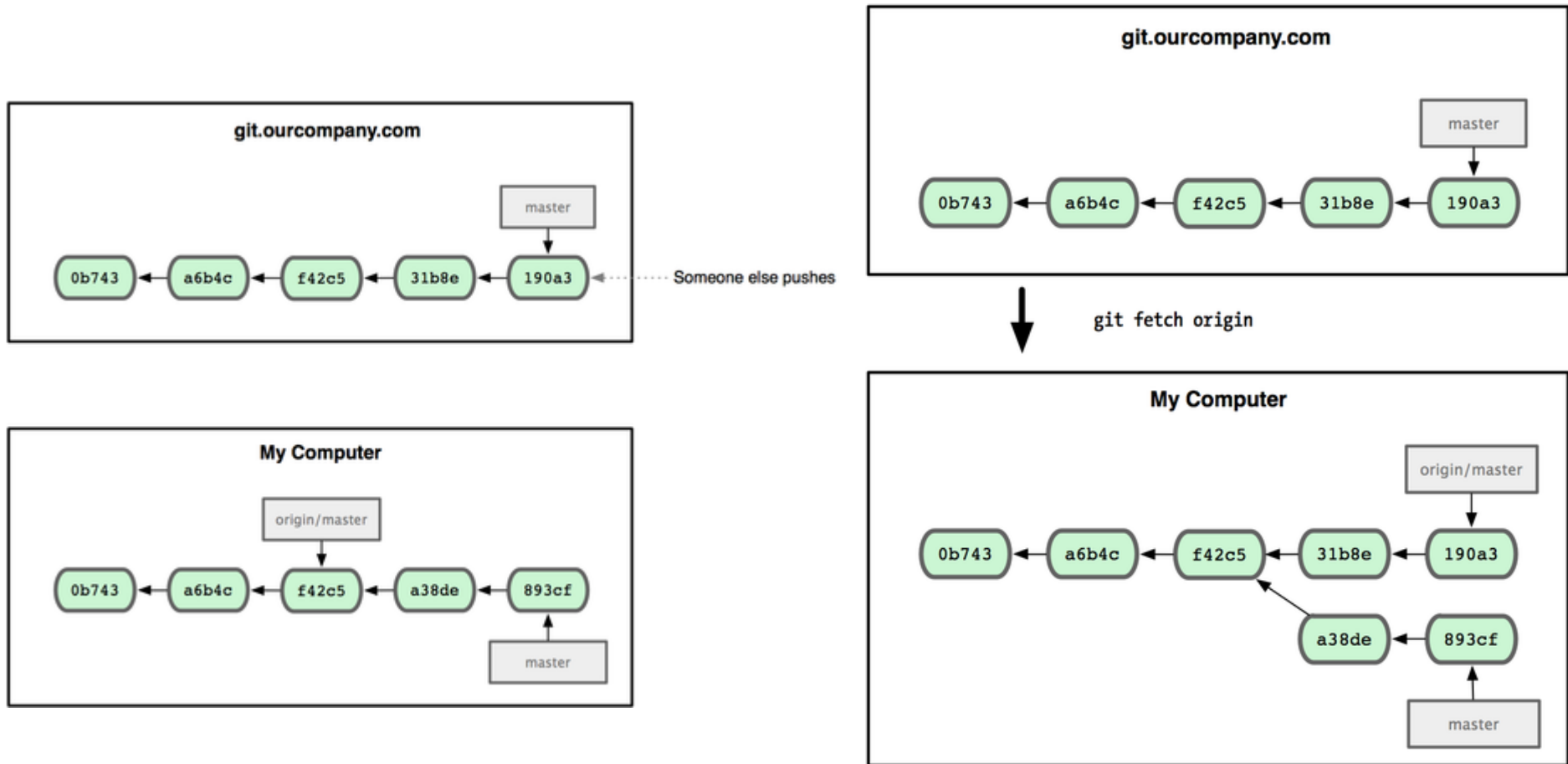
remote branch

- 리모트 브랜치 : (remote)/(branch) 형식
 - ex) 리모트 저장소 origin의 master 브랜치를 보고 싶을 경우 origin/master 사용

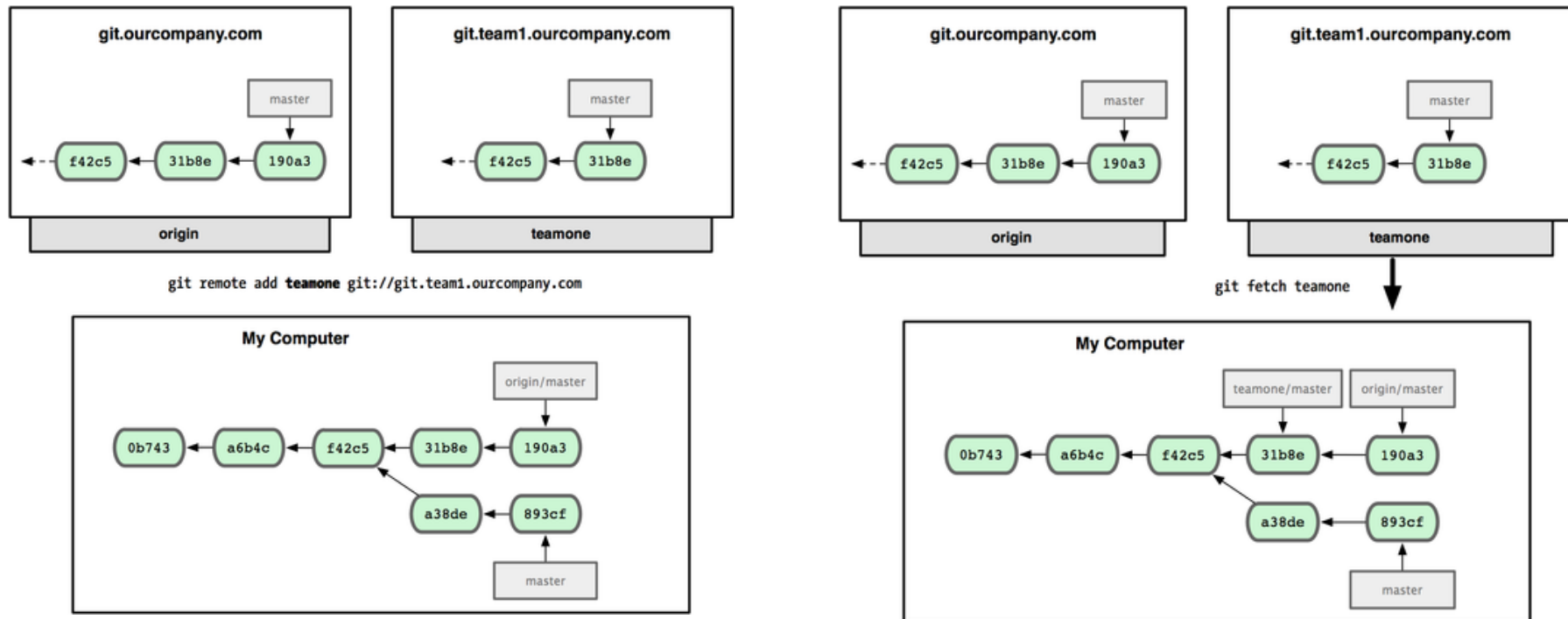


- **\$git fetch origin**

- 원격 저장소로부터 정보를 동기화



- **git remote add [NAME] [Address]**
 - 추가적인 저장소를 NAME으로 저장



- **remote branch**
 - `$git push (remote) (branch)`
 - 특정 브랜치를 다른 사람들과 공유하기 위해 브랜치를 push
 - Ex) `git push origin serverfix`
 - 이후 `git fetch`를 수행하게 되면 서버에 추가된 브랜치를 접근할 수 있다.
 - 하지만 수정 가능한 브랜치가 아니다. 이를 수정하기 위해서는 새롭게 받은 브랜치의 내용을 Merge해야 한다.
 - `$git merge origin/serverfix`
 - `git checkout -b serverfix origin/serverfix`
 - `git checkout --track origin/serverfix`
 - **`$git push (remote):(branch)`**
 - 특정 브랜치를 삭제
 - `$ git push origin :serverfix`
 - To `git@github.com:schacon/simplegit.git`
 - - [deleted] serverfix
-

git tag

- **Tag 붙이기**

git tag -a [이름]

Tag 이름은 중복이 안 됨. 중복될 경우 fatal: tag 'tip' already exists 오류 메시지
이걸 무시하고 싶으시다면, -f 옵션

- **Tag 지우기**

git tag -d [이름]

- **특정 Tag가 가리키는 commit 16진수 보기**

git rev-parse [이름]

- **Tag 정보 올리기**

git push -tags

- **Tag 정보 받기**

git tag -d [이름]

git fetch origin tag [이름]

git log --decorate : tag 정보를 볼 수 있다.

git alias

- `$ git config --global alias.co checkout`
 - `$ git config --global alias.br branch`
 - `$ git config --global alias.ci commit`
 - `$ git config --global alias.st status`
-

• Committing

```
git init  
git add .  
git commit
```

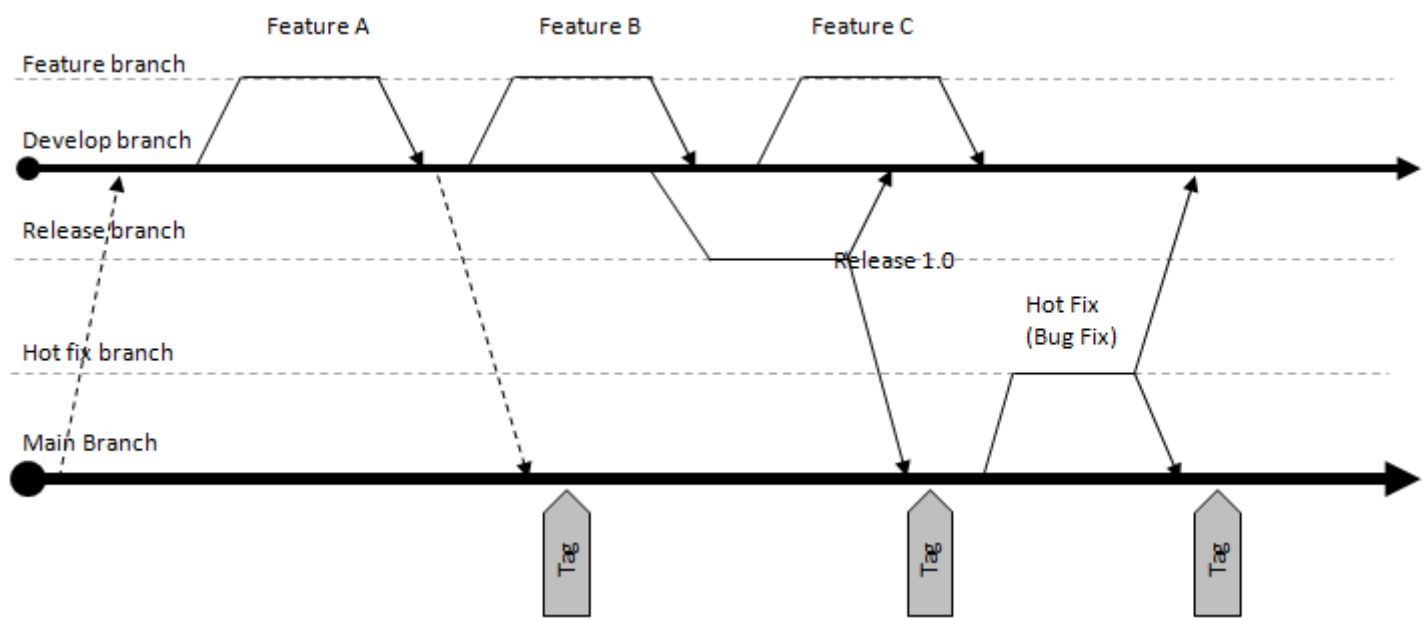
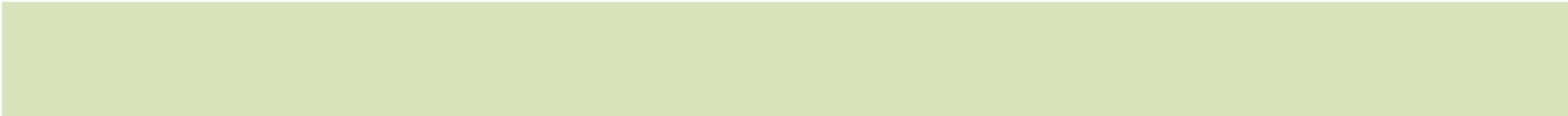
```
svnadmin create repo  
svn import file://repo
```

| | |
|-----------------|-------------------|
| svn diff less | git diff |
| patch -p0 | git apply |
| svn status | git status |
| svn revert | git checkout |
| svn add | git add |
| svn rm | git rm |
| svn mv | git mv |
| svn commit | git commit -a |
| svn log | git log |
| svn blame | git blame |
| svn cat url | git show rev:file |

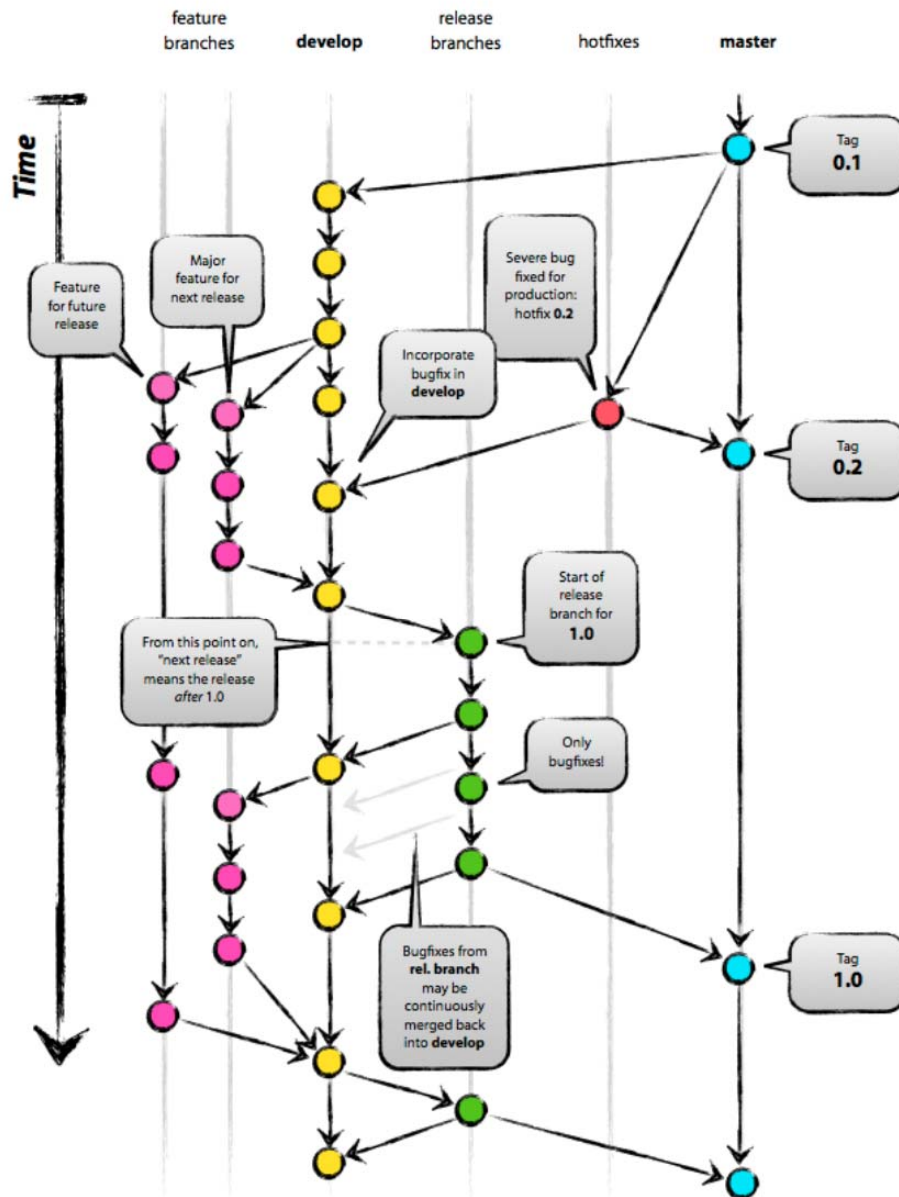


| | |
|--|-------------------------|
| svn diff -crev url | git show rev |
| svn copy http://example.com/svn/trunk http://example.com/svn/tags/name | git tag -a name |
| svn list http://example.com/svn/tags | git tag -l |
| svn log -limit 1 http://example.com/svn/tags/tag | git show tag |
| svn copy http://example.com/svn/trunk http://example.com/svn/branches/branch | git branch |
| git switch http://example.com/svn/branches/branch | git checkout branch |
| svn update -r rev | git checkout rev |
| svn update | git checkout prevbranch |
| svn merge -r 20:HEAD http://example.com/svn/branches/branch | git merge branch |
| svn merge -c rev url | git cherry-pick rev |
| svn update | git pull |

git-flow



git-flow



- Master, Develop
origin의 master 브랜치는 실제 최종 릴리즈된 소스코드만을 관리
개발중의 코드는 모두 develop 브랜치에 커밋. 즉 처음 master 브랜치에서 develop 브랜치를 만든 다음, 개발 중 계속 develop 브랜치에 커밋을 하고 최종 릴리즈가 된 다음, master 브랜치에 머지 하는 방식
- feature
항상 Develop으로부터 브랜치 생성
특정 새로운 기능을 구현할 때 만들고 끝나면 제거
- release
develo으로 브랜치 만들고 나중에 develo과 master 브랜치로 머지
Develop 브랜치에서 웬만한 기능 구현이 완료되었을 때 이 브랜치를 생성
버전명, 빌드 날짜 등을 수정하고, 테스트 중 오류 수정
- hotfix
현재 릴리즈 되어 있는 소스코드(master)의 오류를 긴급 수정할 때 사용

git-flow install

- **Linux**
 - Installing on Ubuntu or Debian
 - `$ apt-get install git-flow`
 - For Debian stable, one can either use the git flow installer, or the Debian package from unstable (it works just fine on stable too).
 - Installing on Archlinux
 - `$ yaourt -S gitflow-git`
 - Installing on Fedora
 - `$ yum install gitflow`
 - Ps.: Tested on Fedora 17, 18 and 19.
 - Other Linuxes
 - Under other Linuxes, the easiest way to install git-flow is using Rick Osborne's excellent git-flow installer, which can be run using the following command:
 - `$ wget --no-check-certificate -q -O - https://raw.githubusercontent.com/nvie/gitflow/develop/contrib/gitflow-installer.sh | sudo bash`
-

- **\$ git flow init**

- 일반적으로 기본적인 구현 진행은 **develop** 브랜치에서 진행한다.

- 하지만, 완전히 새로운 큰 기능 구현을 시작한다면, 아래와 같은 명령을 사용한다.

```
git flow feature
```

```
git flow feature start <name> [<base>]
```

```
git flow feature finish <name>
```

<base> 는 develop 브랜치의 특정 commit ID 를 뜻한다. feature 를 finish 하면 해당 브랜치가 삭제

- 만일 릴리즈 과정을 시작한다면, 아래의 명령을 사용한다.

```
git flow release
```

```
git flow release start <release> [<base>]
```

```
git flow release finish <release>
```

마찬가지로, <base> 는 develop 브랜치의 commit ID 를 말한다. release 가 finish 되면, master 브랜치에 머지가 된다.

- 릴리즈 후, 수정사항이 발생할 경우, hotfix 브랜치를 시작한다.

```
git flow hotfix
```

```
git flow hotfix start <release> [<base>]
```

```
git flow hotfix finish <release>
```

위에서 <base> 는 master 브랜치의 특정 commit ID 를 뜻한다.

git-flow

