

git basic




About..

컴퓨터소프트웨어공학과
김 원 일



- git book
 - <https://git-scm.com/book/ko/v2>

 **git** --distributed-even-if-your-workflow-isnt

Search entire site...

About
Documentation

- Reference
- Book**
- Videos
- External Links


Downloads
Community

This book is available in [English](#).
Full translation available in

- [azərbaycan dili,](#)
- [български език,](#)
- [Deutsch,](#)
- [Español,](#)
- [Français,](#)
- [Ελληνικά,](#)
- [日本語,](#)
- [한국어,](#)
- [Nederlands,](#)


Book


The entire Pro Git book, written by Scott Chacon and Ben Straub and published by Apress, is available here. All content is licensed under the [Creative Commons Attribution Non Commercial Share Alike 3.0 license](#). Print versions of the book are available on [Amazon.com](#).




2nd Edition (2014)

Download Ebook

 pdf

 epub

 mobi

1. 시작하기

- 1.1 버전 관리란?
- 1.2 짧게 보는 Git의 역사
- 1.3 Git 기초
- 1.4 CLI
- 1.5 Git 설치
- 1.6 Git 최초 설정
- 1.7 도움말 보기
- 1.8 요약

2. Git의 기초

- 2.1 Git 저장소 만들기
- 2.2 수정하고 저장소에 저장하기
- 2.3 커밋 히스토리 관리하기



일반적인 파일 버전 관리



• 일반적인 로컬 파일 버전 관리

- 디렉터리에 **파일을 복사하여 관리**하는 방법을 사용
- 디렉터리명에 **날짜와 시간**을 포함하여 관리하는 것이 일반적

• 문제점

- 디렉터리 삭제나 잘못된 파일 수정 및 복사 등으로 관리가 어려움
- 해당 디렉터리 생성 시점에 필요한/알아야 하는 **정보를 별도로 관리**해야 함
 - 날짜와 시간만으로는 해당 시점에 관련된 정보를 모두 확인하기 어려움

이름	수정한 날짜	유형
 2021_04_11	2021-04-26 오후 5:12	파일 폴더
 2021_04_12	2021-04-26 오후 5:12	파일 폴더
 2021_04_12_1	2021-04-26 오후 5:12	파일 폴더
 2021_04_12_2	2021-04-26 오후 5:12	파일 폴더
 2021_04_13	2021-04-26 오후 5:12	파일 폴더
 2021_04_14	2021-04-26 오후 5:12	파일 폴더



버전 관리



- VCS(**Version Control System**)
 - **파일 변화를 버전에 따라 관리할 수 있는 시스템**
 - **파일 역사**(History)를 관리할 수 있는 시스템을 통칭
 - 다양한 형태의 파일의 변화를 저장하고 관리 가능
 - 단위 파일 뿐만 아니라 프로젝트 전체를 관리할 수 있음
 - **다양한 형태의 프로젝트 변화를 확인할 수 있음**
 - **파일의 이전 상태** 확인
 - **프로젝트의 이전 상태** 확인
 - **수정 내용 비교**
 - 문제를 발생시킨 **사용자**에 대한 **정보 확인**
 - 파일의 생성과 변화 전체를 확인 가능
 - 이슈 발생 시점과 관련 정보 확인 가능
 - 파일 정보를 **로컬 데이터베이스**에 기록하는 형태
 - 파일 변경 정보를 관리할 수 있도록 인터페이스를 제공
 - 파일 변경에 대한 정보를 **패치**(Patch) 형태로 관리
 - 패치 적용으로 특정 시점으로 파일을 되돌릴 수 있도록 관리 인터페이스 제공



중앙집중식 버전 관리



- CVCS(**Central VCS**)

- 서버를 이용하여 버전을 중앙에서 관리하는 시스템
- CVS, Subversion, Perforce와 같은 시스템
- 파일에 대한 모든 정보를 서버에서 관리
- 클라이언트(개발자)는 변경 정보를 서버에서 받아서 적용
- 관리자에 의한 중앙 집중식 관리가 가능
- 모든 클라이언트가 관리된 동일한 파일을 전달 받을 수 있음
- 문제점
 - 서버에 문제가 발생할 경우, 해결할 수 있는 방법이 존재하지 않음
 - 개발 시스템 전체가 정지되므로 개발 업무 자체가 중단되는 사태가 발생
 - 서버 디스크에 문제가 발생한 경우, 파일에 대한 전체 History 정보를 잃을 수 있음



분산 버전 관리



- DVCS(**Distributed VCS**)
 - 파일 History와 **파일 변경의 모든 정보를 분산하여 저장**하는 구조
 - 관리 서버의 모든 정보를 **클라이언트도 동일하게 보관**
 - 모든 정보가 클라이언트로 복제되어 저장
 - 서버에 문제가 생겨도 가장 최신 버전을 갖는 개발자 정보로 복원이 가능
 - 완벽하게 복원하는 것은 어려우나, 주요 History는 복원
- git
 - **Linux Kernel 소스 관리**를 위해 개발됨
 - 1991~2002년 동안 Linux Kernel은 Patch 파일과 단순 압축 파일로 관리
 - 2002년 BitKeeper라는 상용 DVCS 프로그램을 이용하여 관리를 시작
 - 2005년 BitKeeper의 상용화로 소스 관리 도구의 필요성이 각인됨
 - Linux Torvalds가 직접 개발에 참여
 - 빠른 속도와 단순한 구조로 설계
 - 비선형적인 개발 지원
 - 완벽한 분산 데이터 저장
 - 대형 프로젝트에서도 문제 없도록 다양한 전략을 제시



기존 파일 버전 관리 - 1



• 파일 차이를 이용한 버전 관리

- 기존 코드 관리 프로그램들은 **patch 형식으로 정보를 관리**
- 텍스트 패치 형식은 **변화된 부분만을 별도 보관**하는 형태
- 개방형 OS에서는 diff 명령으로 간단히 patch 파일 생성이 가능
 - 2개의 파일을 비교하여 다른 점을 확인하는 프로그램

```
unangel@unangel: ~/work$ more test_a.txt
#include <stdio.h>
#include <stdlib.h>

int main( int argc, char **argv )
{
    printf( "Hello World\n" );
    return 0;
}
```

```
unangel@unangel: ~/work$ more test_b.txt
#include <stdio.h>

void main( void )
{
    printf( "hello world\n" );
    return 0;
}
```

```
unangel@unangel: ~/work$ diff test_a.txt test_b.txt
2d1
< #include <stdlib.h>
4c3
< int main( int argc, char **argv )
---
> void main( void )
6c5
<     printf( "Hello World\n" );
---
>     printf( "hello world\n" );
unangel@unangel: ~/work$ █
```

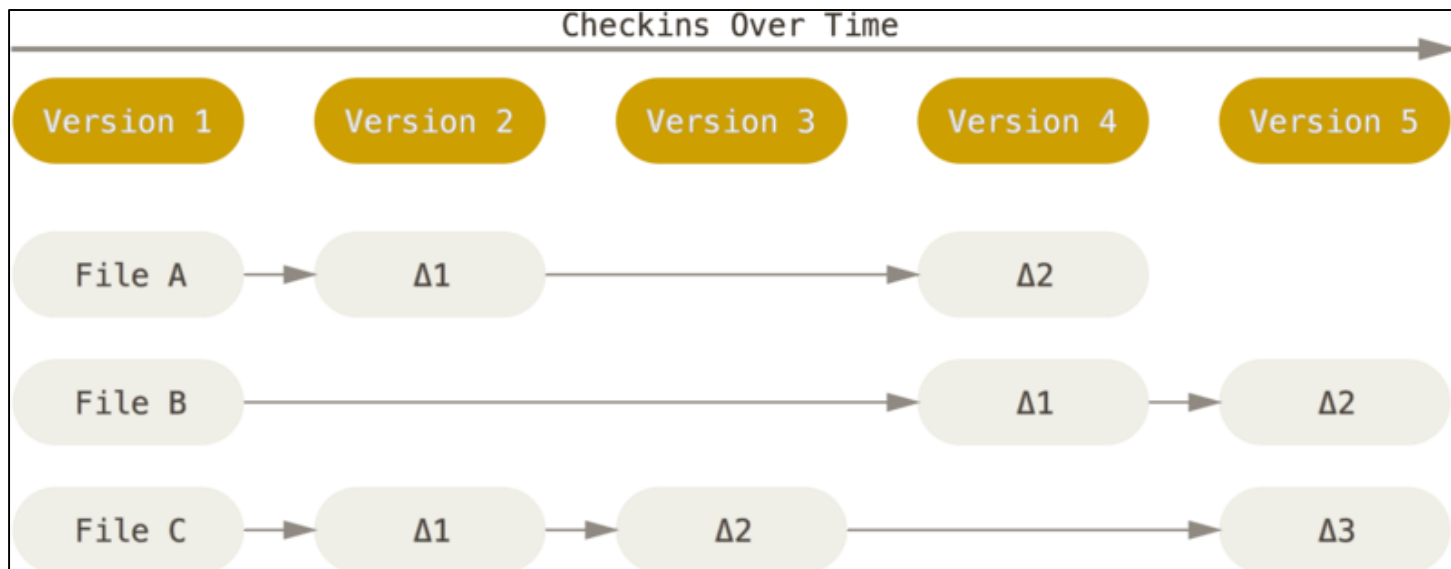


기존 파일 버전 관리 - 2



• 파일 목록의 관리

- 파일 변경을 patch 형태로 저장하여 사용하는 형태
- 파일 변화를 시간 순으로 관리하면서 파일 집합을 관리
- 델타 기반 버전 관리 시스템의 사용
 - version 4를 가져올(Checkout) 경우
 - "File A"에 델타1 $\Delta 1$ 을 적용하고, $\Delta 2$ 를 적용하여 파일을 가져오는 형식
 - 원본 파일을 항상 보관한 상태에서 시작해야 함



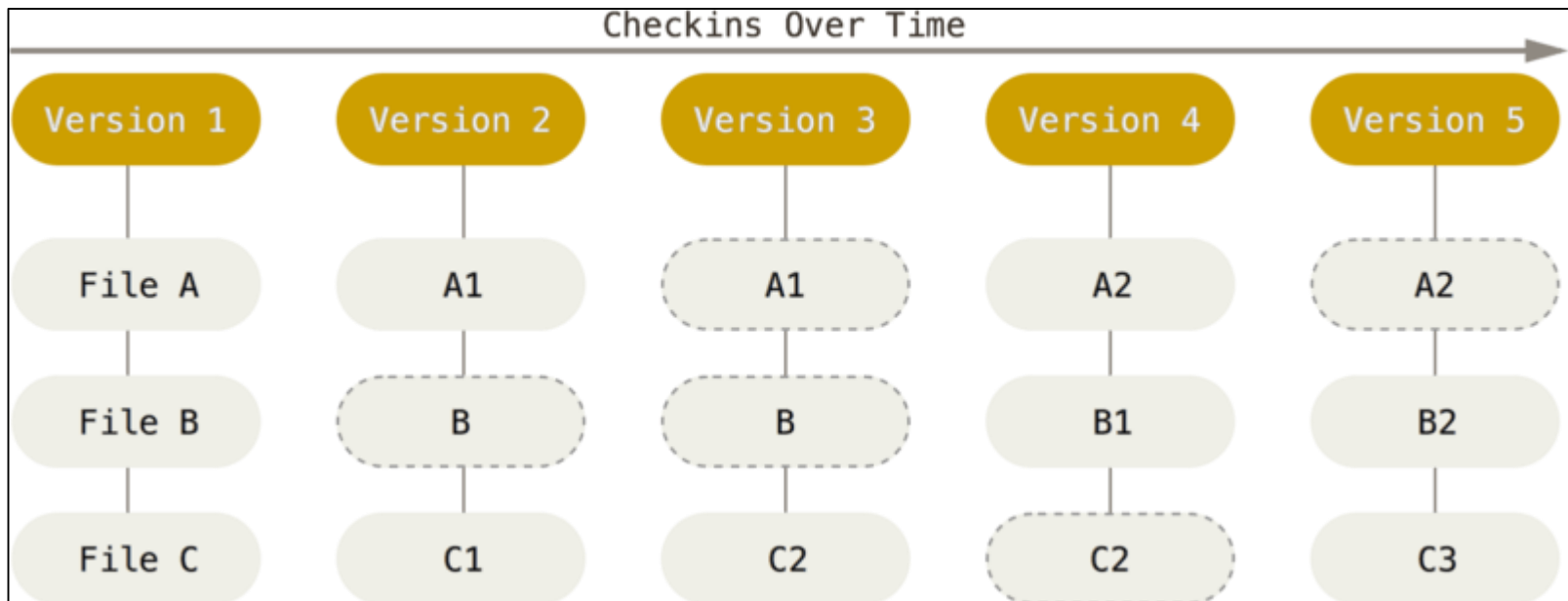


git을 이용한 파일 버전 관리 - 1



• 스냅샷을 이용한 버전 관리

- 스냅샷을 이용한 파일 시스템은 **변경 전 파일에 대한 정보만 링크**로 사용
- 스냅샷 기반 버전 관리 시스템의 사용
 - Version 4를 가져올(Checkout) 경우
 - A2, B1, C2 파일을 그대로 가져오기만 하면 문제 해결
 - C2 파일은 링크로 구성되고, 변경이 발생하면 C3로 저장하면 완료





git을 이용한 파일 버전 관리 - 2



- 저장소의 모든 정보를 모두가 가짐

- **git clone**을 통한 복제는 원본을 그대로 복제하여 로컬에 저장
 - 서버에 문제가 발생해도 모든 복제 본에 파일 이력과 변경 정보가 보관되는 형태
 - 하나의 복제 본만 존재해도 서버를 다시 구축할 수 있음
 - 파일 이력에 대한 데이터베이스 정보를 모두가 동일하게 보관
 - 서버에 업로드가 된 경우에 한함
- git은 **로컬 기반으로 데이터를 관리**하는 것이 기본 설정
 - 로컬에 모두 복제되므로, **이전 파일 정보도 로컬에서 즉시 확인 가능**
 - CVS와 같은 시스템은 이전 파일 정보 확인을 위해 서버에 반드시 접속해야 함
- **원격지 접근은 필요한 경우에만 접근**
 - **서버 업데이트**를 제외하고, 모든 작업을 로컬에서 수행할 수 있음
 - 협업 정보의 교환 전에는 굳이 서버에 업로드하지 않아도 문제 없음
- **무결성 지원**
 - 스냅샷을 작성할 때, 모든 파일에 대한 **체크섬**을 구하여 데이터를 관리
 - 체크섬은 SHA-1 해시를 이용하여 생성 (40자 길이의 16진수 문자열)

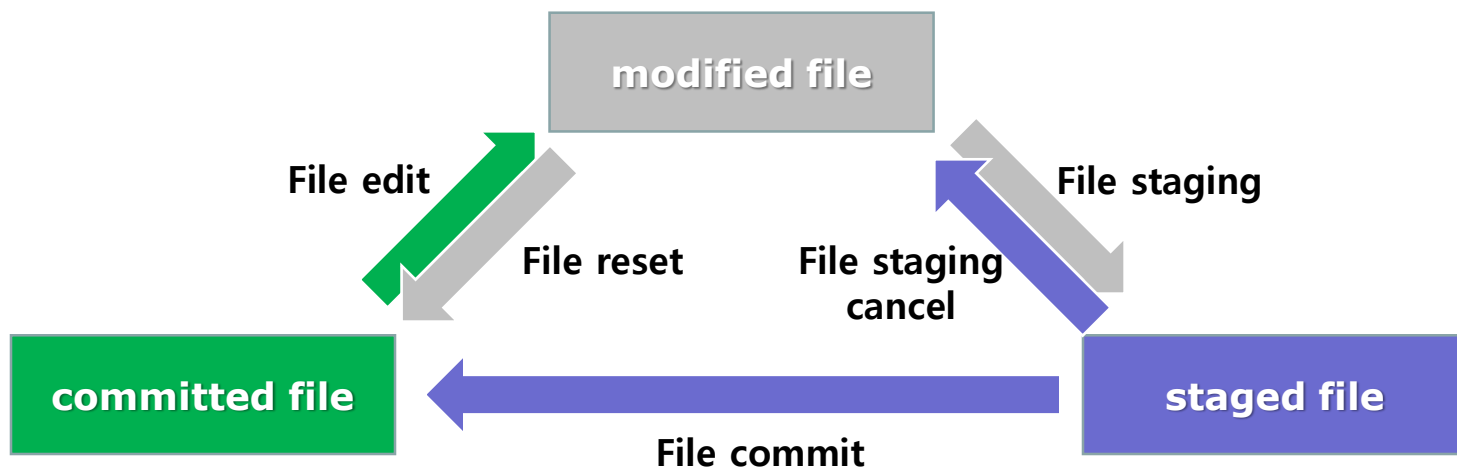


git을 이용한 파일 버전 관리 - 3



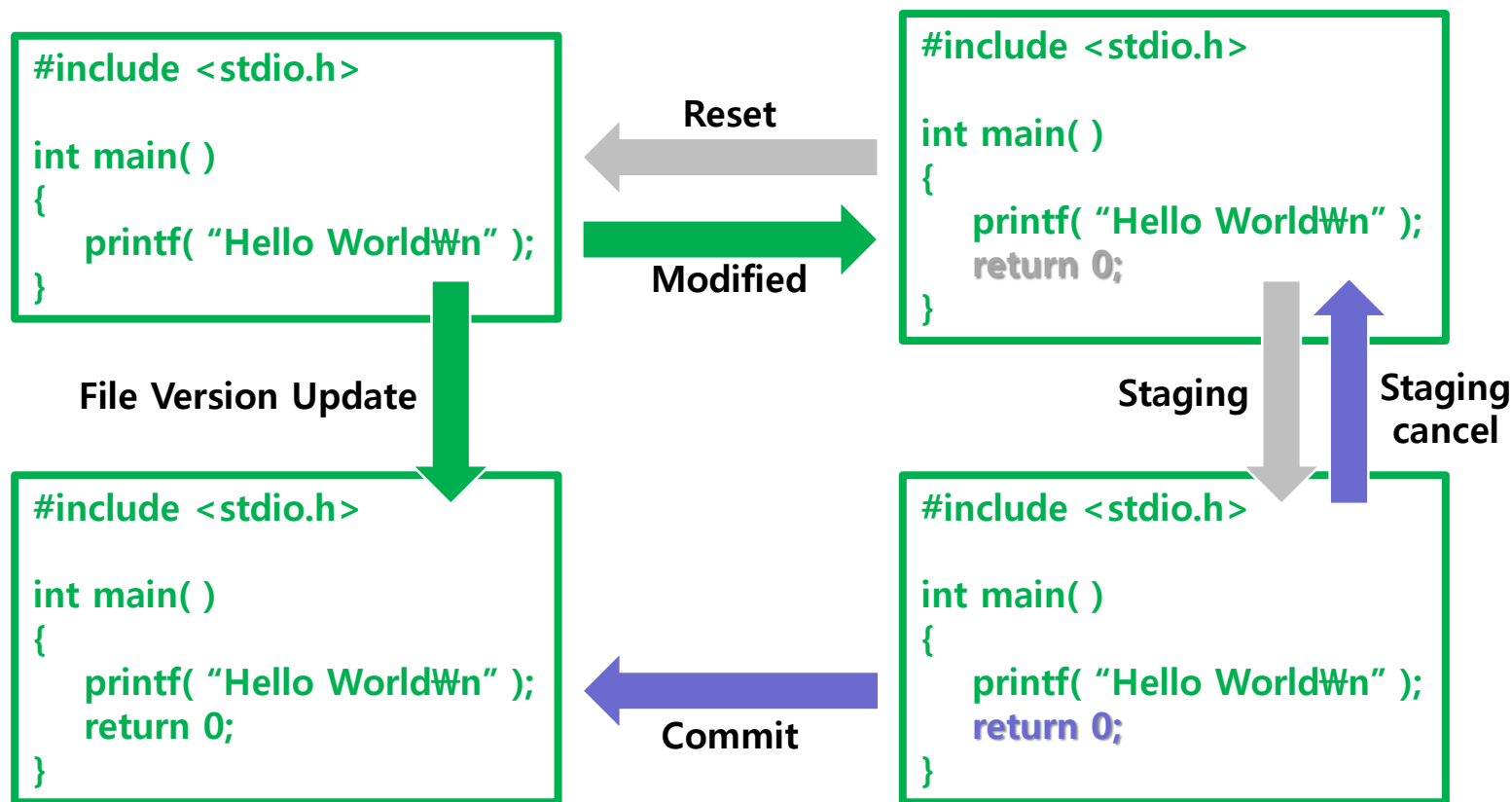
• 파일 상태 관리 - 1

- **committed** : git에서 파일을 안전하게 관리 및 추적하고 있는 상태
 - 수정 발생 전으로 언제든지 복원이 가능
- **modified** : 수정이 발생하였으나, 수정 내용은 아직 관리되지 않는 상태
 - 수정 내용이 데이터베이스에 적용되지 않았음
- **staged** : 수정 파일이 곧 commit될 예정인 상태
 - 데이터베이스에 바로 적용이 가능한 상태로, 추가적인 파일 수정이 불가능한 상태
 - 취소를 통해 파일을 다시 수정할 수 있음





• 파일 상태 관리 - 2





git 파일 버전 관리 명령어



- **자신의 git 저장소를 생성**

- ycs-학번 계정으로 저장소를 생성
- 저장소를 로컬로 복제하기 : **git clone**
- 복제된 저장소에 파일을 수정하기
- 수정된 파일(modified) 정보 확인 : **git status**
- 수정된 파일을 staging에 추가하기 : **git add**
- staging 파일을 수정 상태로 변경하기 : **git reset HEAD**
- 다시 staging 상태로 변경하기 : **git add**
- committed 상태를 로컬 데이터베이스에 기록하기 : **git commit**
- 변경된 로컬 데이터베이스를 서버로 업로드하기 : **git push**

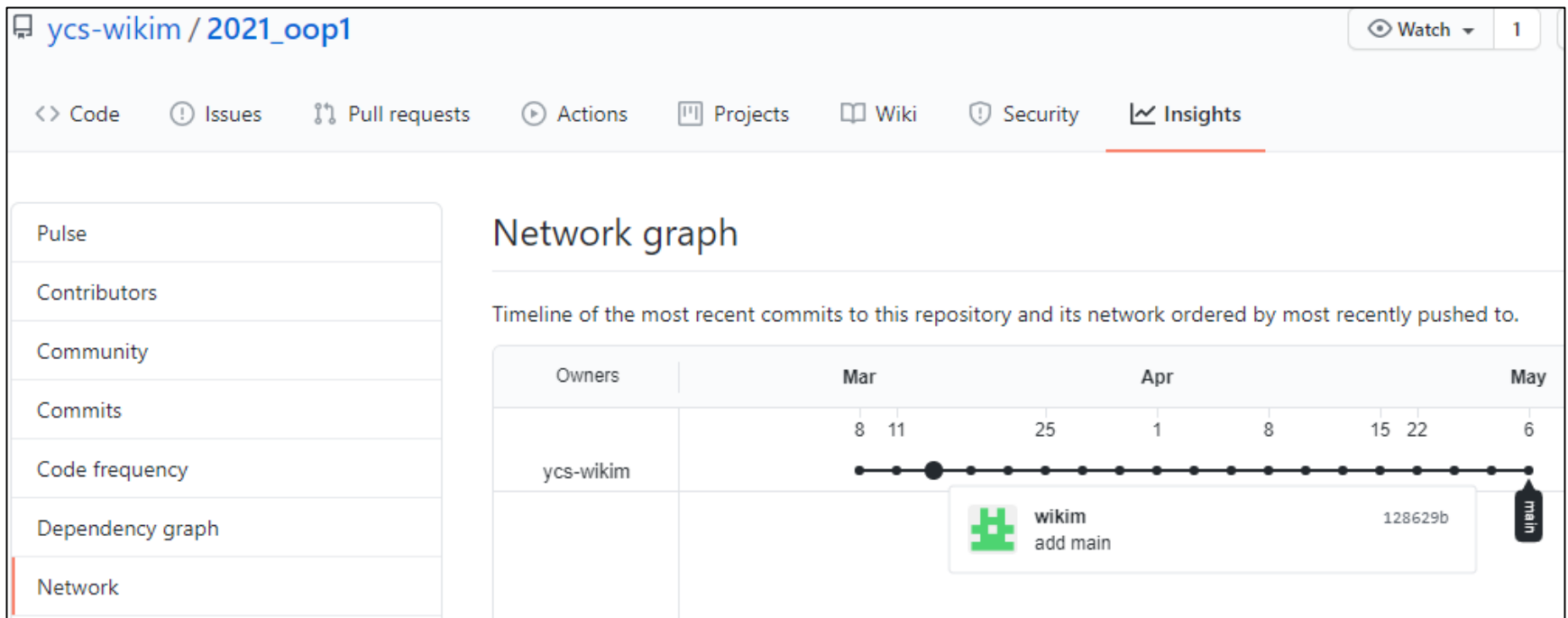


git 버전 정보 확인



• 저장소 업데이트

- 로컬에서 commit한 내용을 서버로 전달할 때 업데이트 발생
- 발생한 업데이트는 날짜/사용자 정보와 체크섬 정보가 같이 전달
- 저장소의 "Insights" → "Network" 메뉴에서 상태 확인 가능
- 파일 이력에 대한 정보를 전체적으로 확인할 수 있음



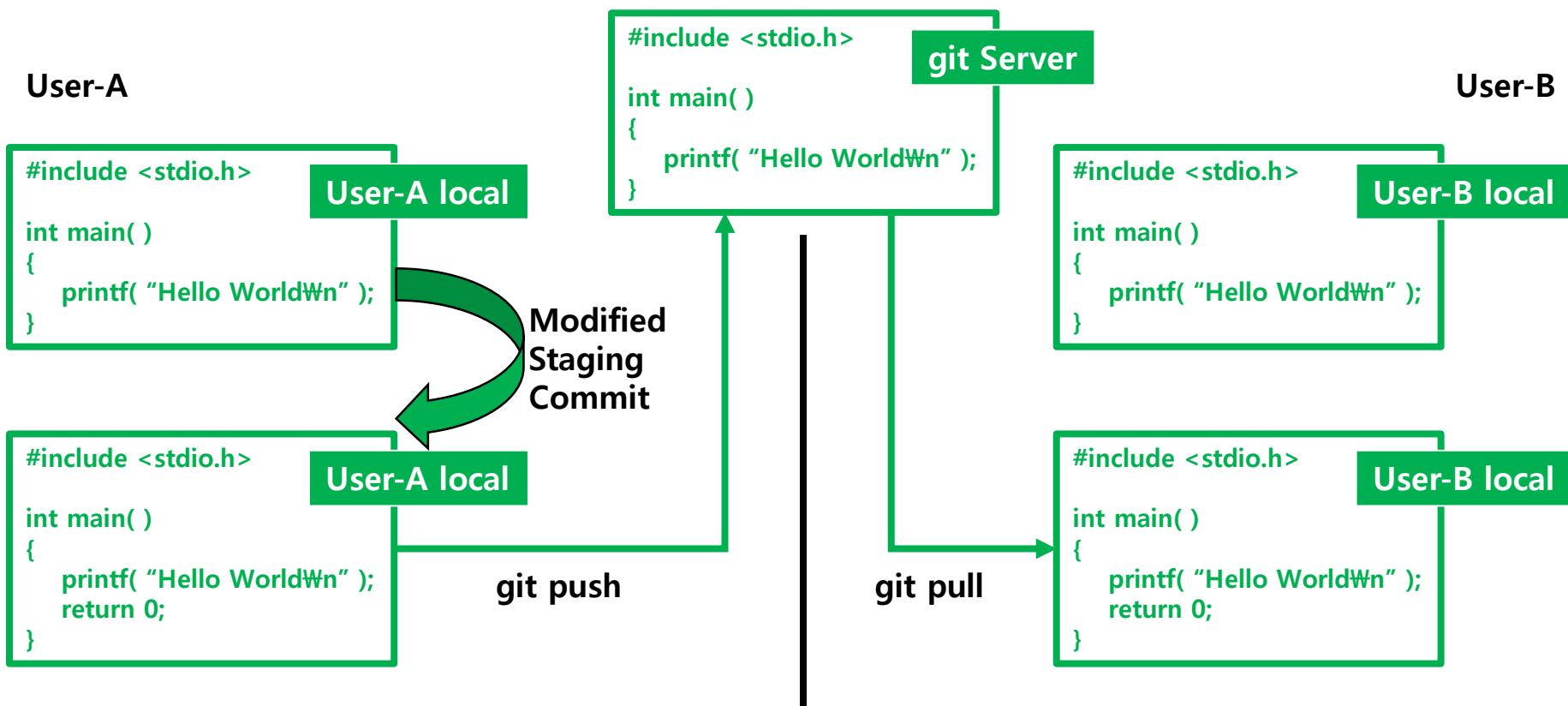


협업을 위한 git 사용 - 1



• 협업 시 주의해야 할 점

- 모든 정보가 **로컬**에 저장되므로, 업로드 전에는 변경 사항을 알 수 없음
- 서버에 업로드하면, 협업자들은 모두 정보를 **갱신**해야 함
- 동일한 파일을 다수 사용자가 수정할 경우, 충돌이 발생





협업을 위한 git 사용 - 2



- 시스템 및 소스 파일 설계 우선

- 시스템에 대한 분석과 기능 구현을 먼저 **설계**하고 개발을 시작
- 소스 코드 개발에 대한 **분업**을 먼저 진행하는 것이 좋음
- 수정할 파일과 수정하지 않을 파일을 구분하여 작업
- 동일 파일을 생성하거나 수정할 경우 문제가 발생 함
- **지정된 파일 이외에는 수정하지 않는 것이 기본**

- 브랜치(branch) 활용

- 브랜치란 **원래 코드**와 관계 없이 독립 개발을 지원하는 논리적인 개념
- 특정 버전 상태에서 논리적으로 분리된 파일들을 별도로 사용
- A 브랜치에서 브랜치를 생성한 시점의 A 파일들을 별도로 사용 가능
- 병합 후 업로드할 경우, 브랜치는 서버에 업로드하지 않아도 문제 없음
- 필요한 경우에는 서버에 브랜치를 업로드해야 함

◆ 브렌치의 활용 - 1

• 브렌치는 일종의 게임 세이브와 동일

- 게임 세이브로 **현재 상태를 저장**할 수 있음
- 저장 상태 후 플레이 도중 이전 상태로 되돌리고 싶을 경우 세이브를 로드
- 이때, 현재 상태 저장은 사용자의 선택 등을 저장하지 않음
- 원하는 시점을 로드하여 확인이나 수정 등이 가능



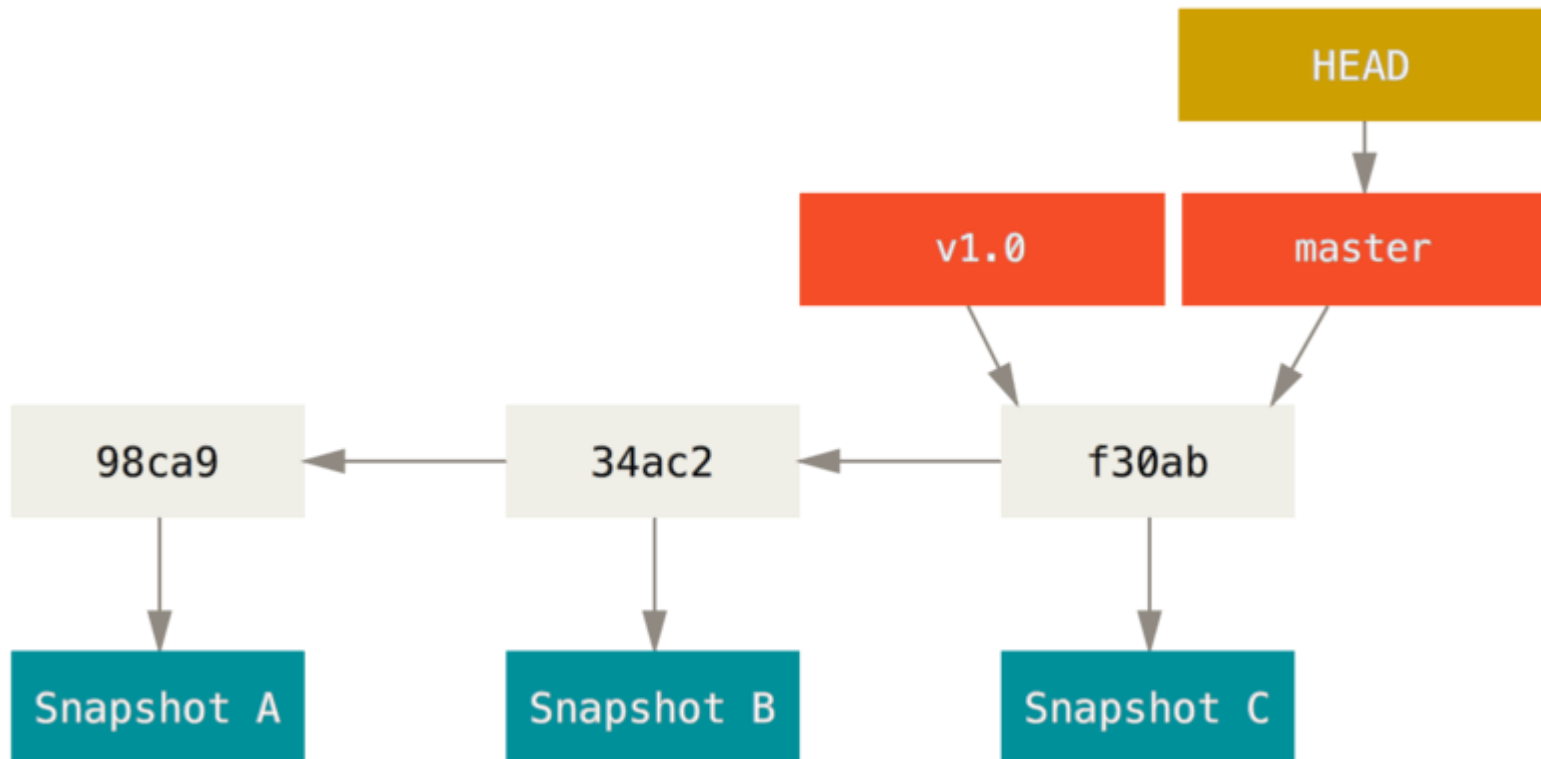


브랜치의 활용 - 2



- 브랜치를 통한 파일 별도 보관

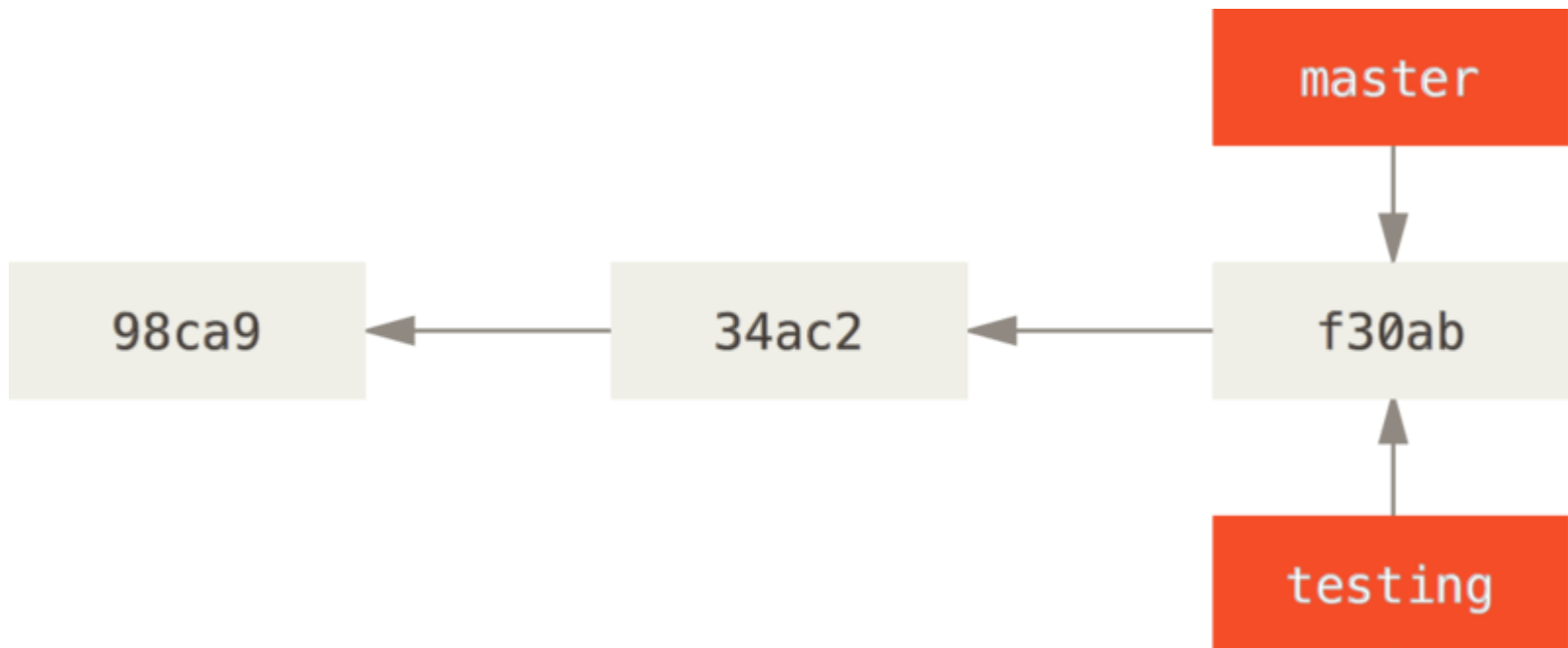
- 저장소의 기본 브랜치 : **master** 또는 **main**
- 현재 저장소의 상태





• 브랜치의 생성과 이동

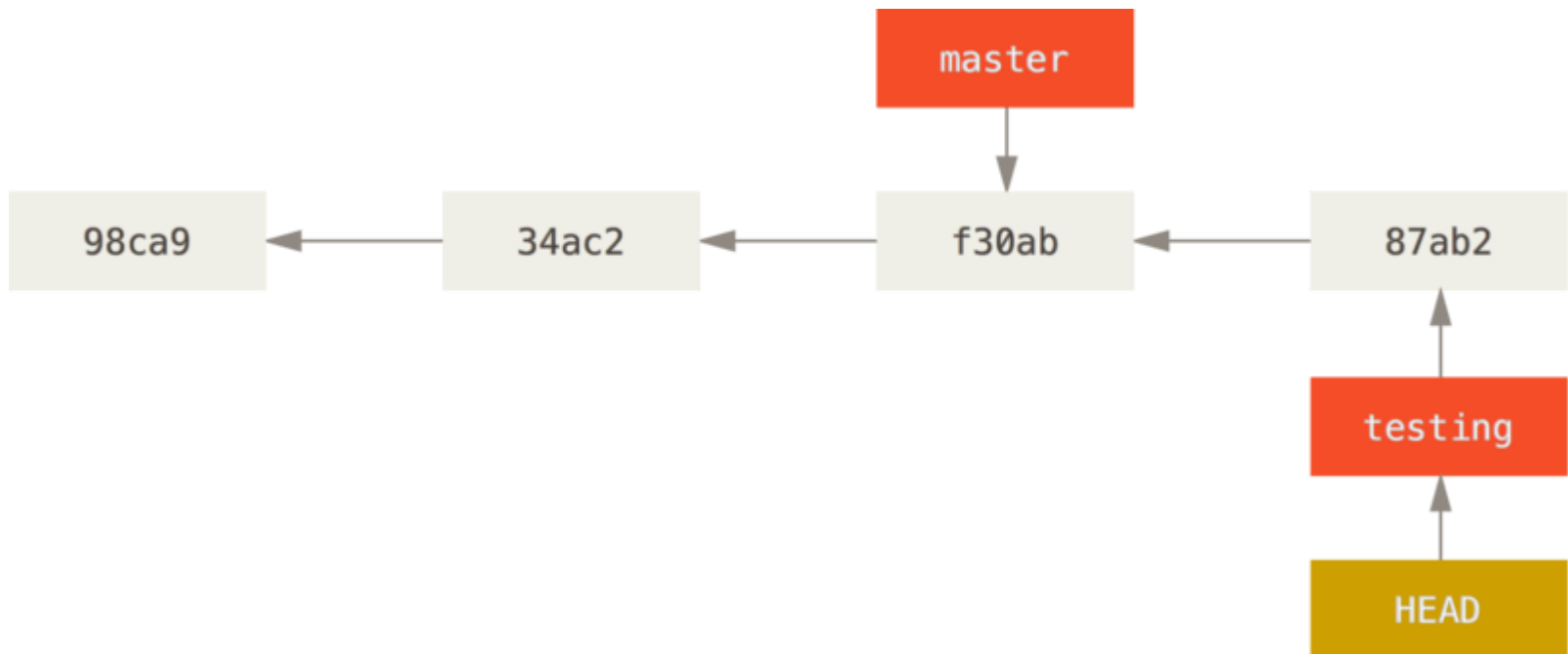
- 저장소의 브랜치 생성: **git branch** "브랜치이름"
- 현재 상태에서 논리적인 개별 상태를 생성
- "git branch testing" 명령을 수행하면 "testing" 브랜치가 생성됨
- "**git checkout** testing" 명령으로 해당 브랜치로 이동
- "git branch -b testing" 명령으로 한번에 수행 가능





• 브랜치에서 파일 수정

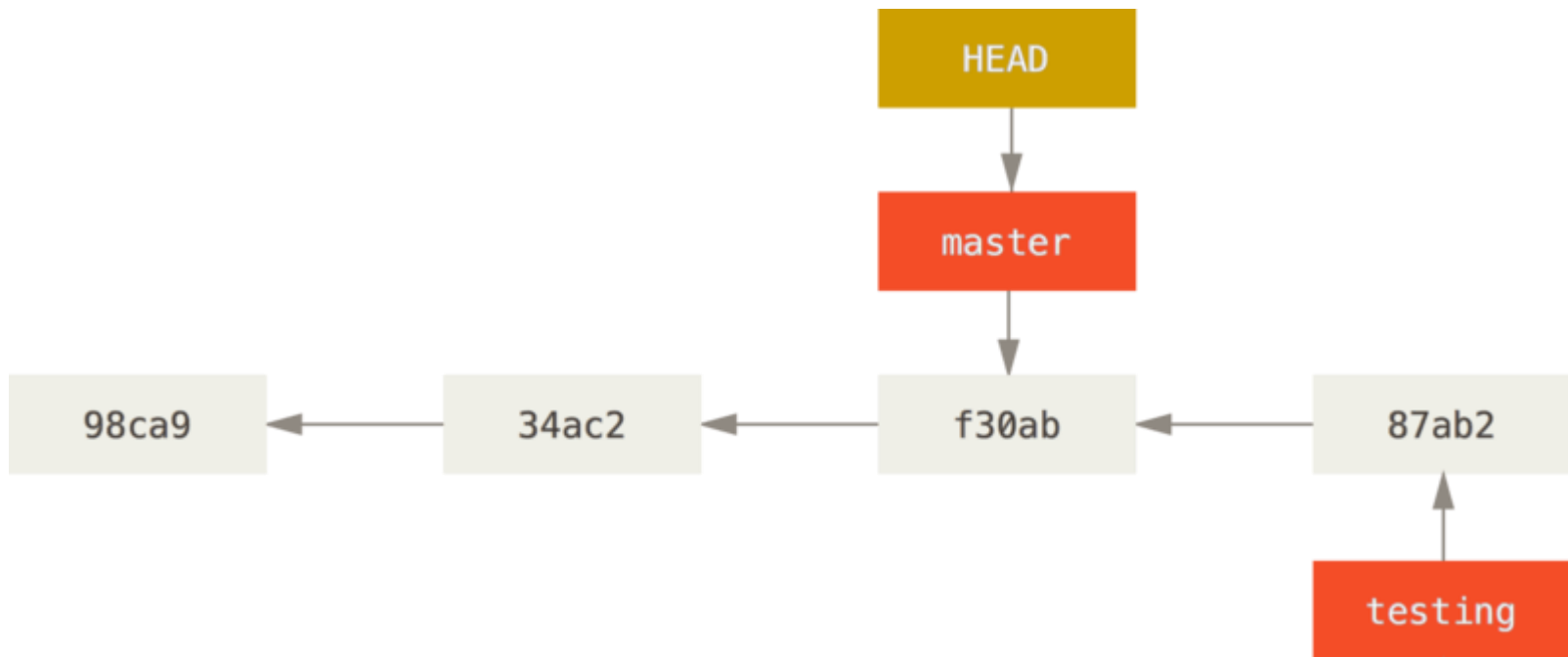
- 생성된 브랜치에서 파일을 수정하여 commit
- 해당 브랜치에 commit된 내용이 추가로 저장
- 현재 작업 위치에 HEAD가 항상 위치





• 브랜치 이동

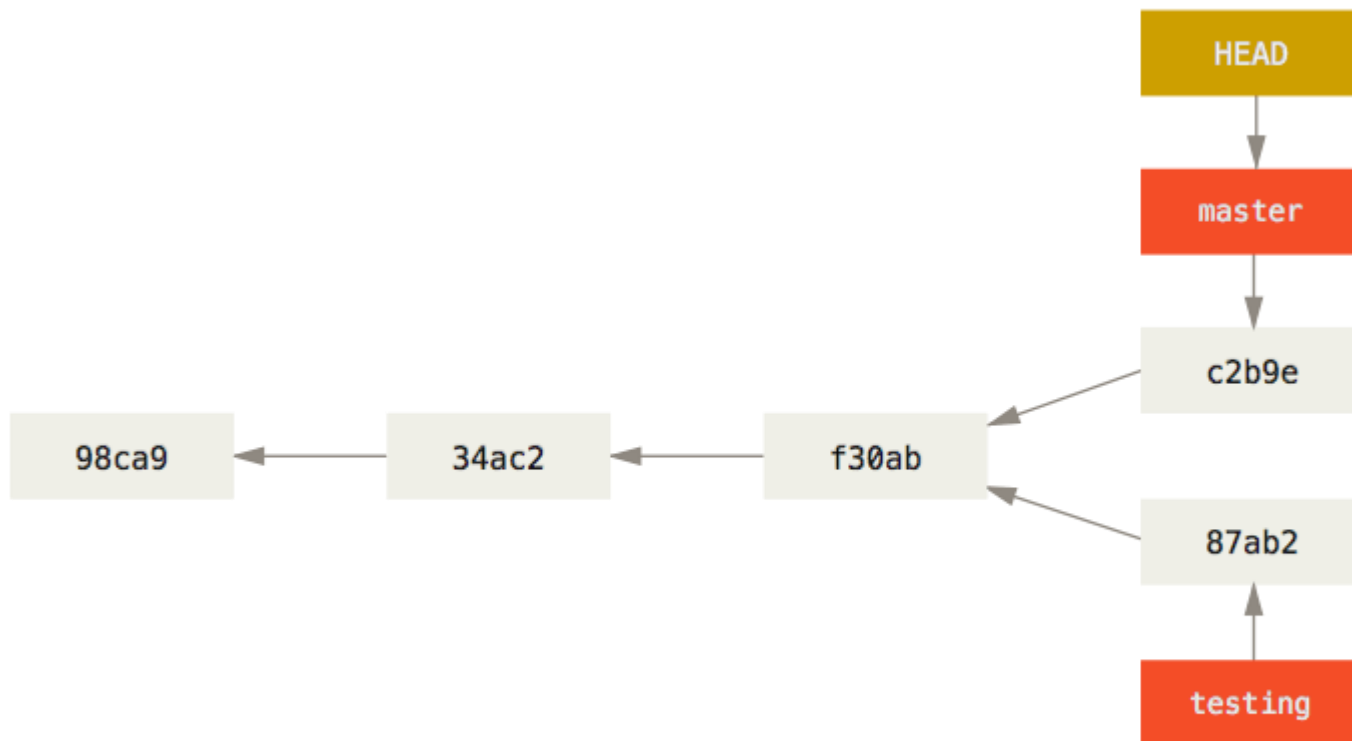
- 현재 브랜치에 모든 내용이 commit/staging된 상태에서만 이동 가능
- untracked 또는 modified 파일이 있는 경우는 이동 불가능
- "git branch 브랜치이름" 명령으로 브랜치 이동 가능
- "git checkout master"로 master 브랜치로 이동 가능
- "**git branch**" 명령은 현재 로컬의 브랜치 목록을 출력





• 브랜치의 파일 comit

- 이동한 master 브랜치에서 파일을 수정하고 commit
- master 브랜치 만의 파일 버전이 별도로 생성되어 업데이트
- 브랜치 이동 시, 각 브랜치 만의 수정 내역을 확인할 수 있음

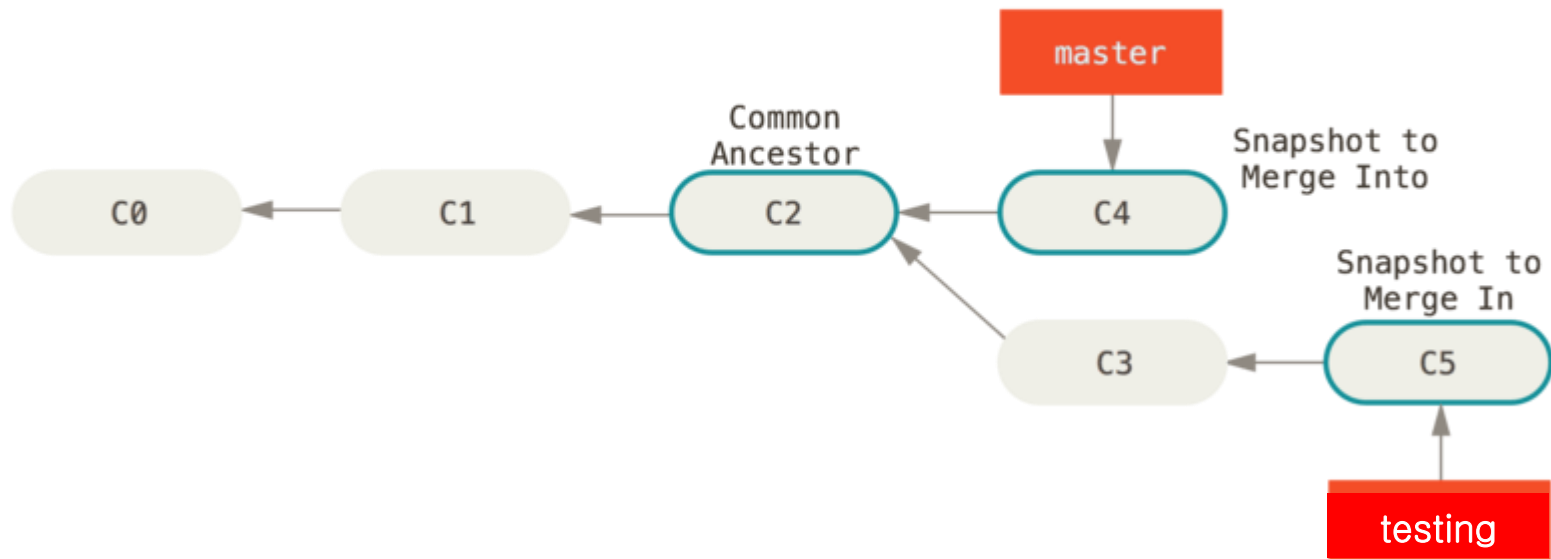


❖ 브랜치의 활용 - 7



• 브랜치 병합 - 1

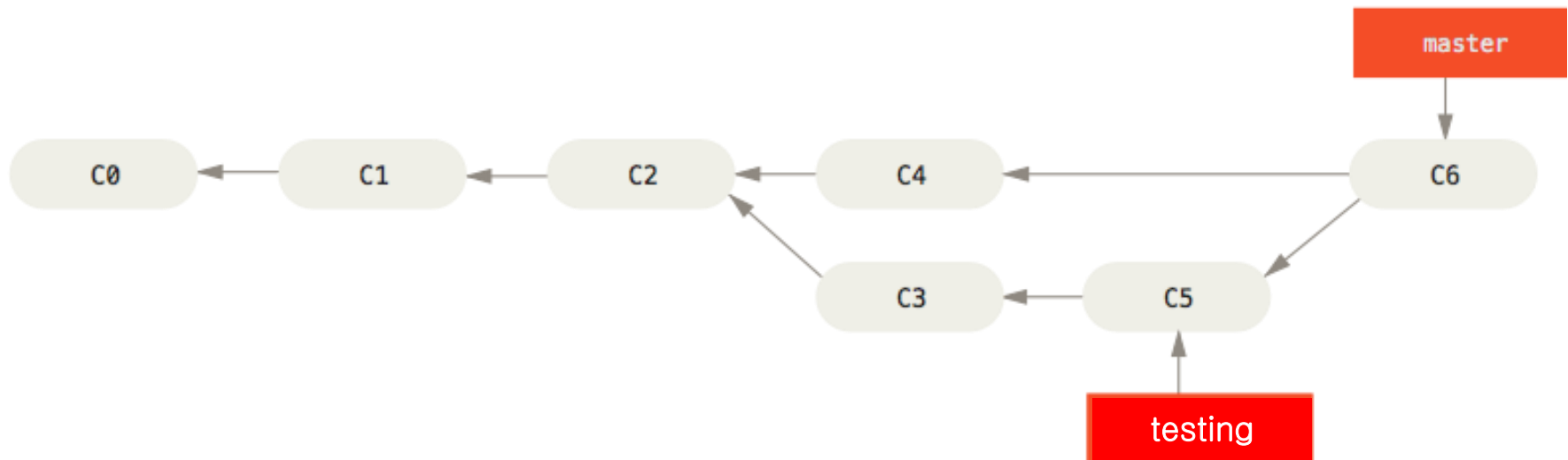
- 브랜치를 생성한 목표를 달성한 경우 병합을 수행
- 병합은 2개의 다른 파일 흐름을 합치는 과정
- "**git merge** 병합할브랜치명" 명령으로 병합
- 현재 브랜치에 다른 브랜치를 합치는 형태로 동작
- **master에 testing 브랜치를 병합하려면 master 브랜치에서 수행해야 함**





• 브랜치 병합 - 2

- "git checkout master"로 master 브랜치로 이동
- "git merge testing"으로 브랜치를 병합
- 병합되면 testing 브랜치 삭제가 가능해짐
- master 브랜치의 **C6** 스냅샷에는 testing에서 수행된 모든 작업이 포함

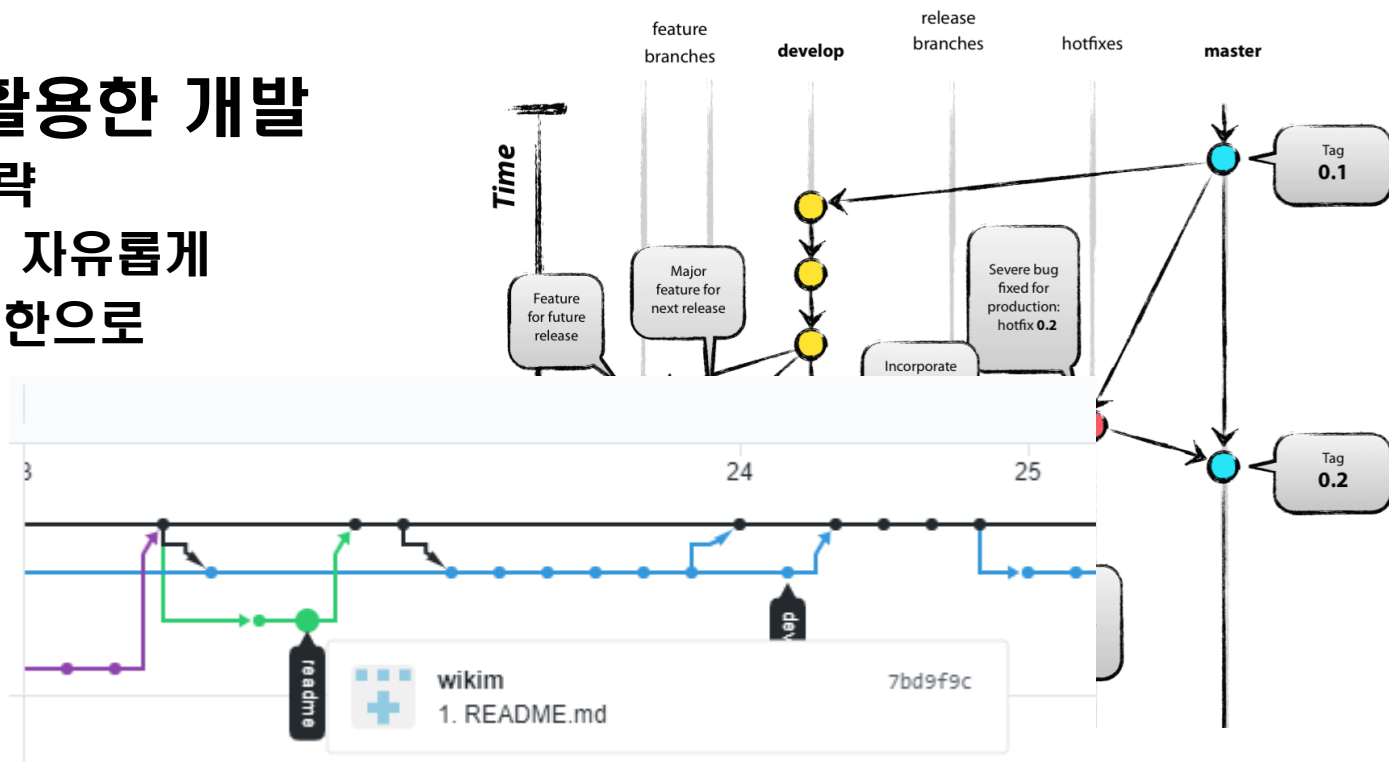




브랜치의 활용 - 9

● 브랜치를 활용한 개발

- git flow 전략
- 버전 관리는 자유롭게
- 충돌을 최소한으로



Pull Request 할때, 하나의 클래스를 몇명에서 수정하게되면 겹치게될텐데 그런경우엔 어떻게 하시나요?

좋아요 · 답글 달기 · 3년

저희는 되도록 코드 충돌이 발생하지 않도록 작업을 나누어서 진행을 합니다.

그래서 하나의 클래스를 여러 명에서 건들지 않도록 하고 있습니다.

작업을 나눌 때 같은 코드를 여러 명이 건드려야 한다면 작업자들끼리 이야기를 한 후 한 명이 먼저 작업을 처리 합니다.

그렇게 하더라도 간혹 코드 충돌이 발생하게 되는데요. 대부분 이 사실은 오전 티타임을 할 때나 코드리뷰를 할 때 알게됩니다.

코드 충돌 해결은 전적으로 뒤에 코드 병합을 하는 사람이 책임을 지게됩니다.

코드 충돌이 작으면 스스로 처리하지만, 코드 충돌 범위가 크면 작업 코드가 겹친 개발자와 함께 충돌 해결을 하고 있습니다.

좋아요 · 답글 달기 · 8 · 3년