한성대학교
HANSUNG UNIVERSITY

버전 관리 시스템과 GIT

목차

학습목표

- 버전 관리가 무엇인지 이해할 수 있다.
- 버전 관리 시스템의 종류에 대해 이해할 수 있다.
- GIT의 개념에 대하여 이해할 수 있다.
- GIT의 장점 및 특징에 대하여 이해할 수 있다.
- GIT의 로컬 저장소 활용을 위한 기본 명령어를 이해할 수 있다.
- GIT의 작업 흐름 분기를 위한 branch 명령어를 이해할 수 있다.

01 버전 관리 시스템

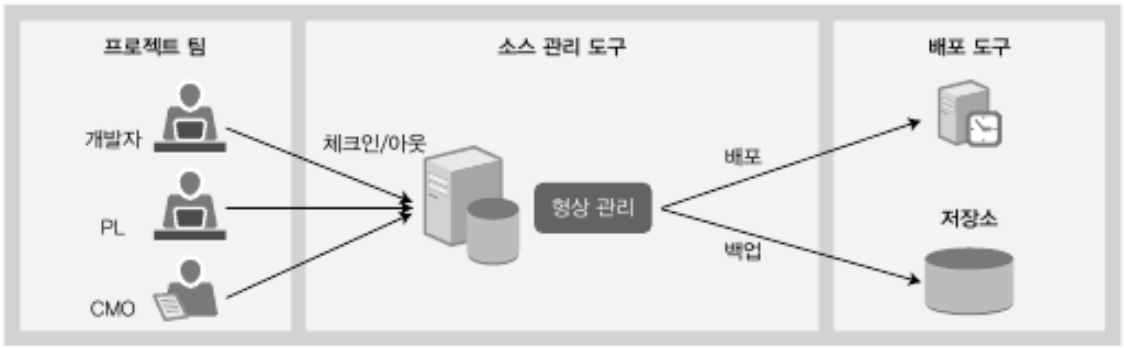
01. 버전 관리 시스템

- **버전: 컴퓨터 파일이나 프로그램의 변경 이력 혹은 수정 내용**
- **버전 관리의 예**
  - RPG 게임
    - 매 전투가 끝난 후 게임을 저장하는 행위
    - 다음 전투에서 전멸하는 경우 이전 전투 직후의 저장 파일을 활용
    - 같은 결과(전멸)이 나오지 않도록 준비를 다르게 해서 진행 가능
  - 리포트 작성
    - 컴퓨터 오류 때문에 파일을 잃어버리지 않도록 지속 저장
    - 날짜를 붙여서 복사본을 따로 저장해두기도 함
  - 태블릿으로 그림 그리기
    - 스케치, 자세한 묘사, 채색 과정이 진행될 때마다 날짜와 시간을 붙여 사본을 생성
    - 완성 후에도 피드백에 따라 내용의 지속적인 변경을 수행
  - 버전 관리를 위한 전문적인 툴 필요

01. 버전 관리 시스템

- **버전 관리 시스템 (Version Control System): 버전을 관리하는 시스템**
- **버전 관리 시스템의 특징**
  - 사본 생성, 보존, 복원을 한 번에 지원하는 시스템
  - 다른 사람들과 협업 시에 서로 간의 상태를 똑같이 유지하는 기능
  - 오류 복구
  - 소스 코드의 변경 사항 추적
  - 기록한 내용에 대한 조회 기능 제공

그림 1-2 버전 관리 시스템의 구조



02 버전 관리 시스템의 종류

02. 버전 관리 시스템의 종류

클라이언트 서버 모델

- 하나의 중앙 저장소를 공유한 후 각각의 클라이언트(개발자)는 저장소의 일부분만을 갖음
- 자신이 작업하는 부분만 로컬에 임시로 저장한 후 작업을 수행
- 중앙 저장소에서 프로젝트 관리의 모든 것을 처리
- 클라이언트에서 할 수 있는 것이 파일 수정과 서버로의 커밋 등 제한적임
- 서버가 고장나면 로컬의 파일 사본들만 남게 됨

그림 1-3 클라이언트-서버 모델

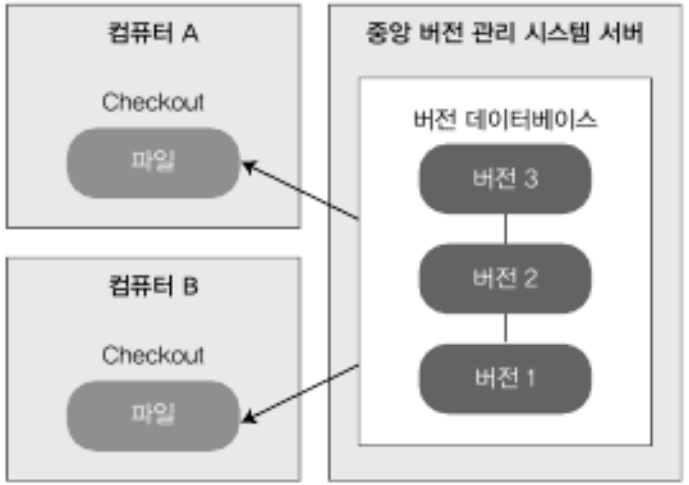


표 1-1 클라이언트-서버 모델의 버전 관리 시스템

무료/유료	버전 관리 시스템
무료 / 오픈 소스	CVS(1986, 1990 in C), CVSNT(1998), QVCS Enterprise(1998), Subversion(2000)
유료 / 상용	Software Change Manager(1970년대), Panvalet(1970년대), Endevor(1980년대), DSEE (1984), Synergy(1990), ClearCase(1992), CMVC(1994), Visual SourceSafe(1994), Perforce(1995), StarTeam(1995), Integrity(2001), Surround SCM(2002), AccuRev SCM(2002), SourceAnywhere(2003), SourceGear Vault(2003), Team Foundation Server(2005), Rational Team Concert(2008)

02. 버전 관리 시스템의 종류

분산 모델

- 프로젝트에 참여하는 모든 클라이언트가 전체 저장소에 대한 개별적인 로컬 저장소를 보유
- 클라이언트 각자가 전체 저장소의 사본을 로컬에 보유
- 저장소와의 모든 상호작용이 로컬 저장소에도 반영될 수 있음

그림 1-4 분산 모델

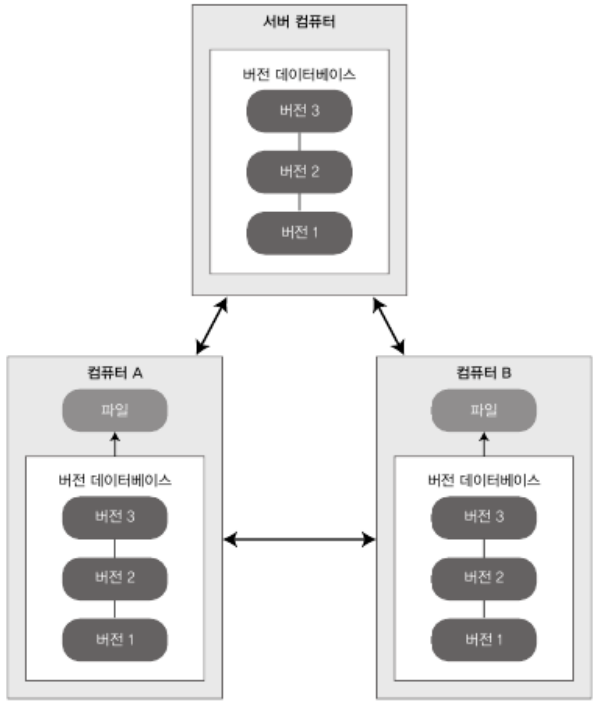


표 1-2 분산 모델의 버전 관리 시스템

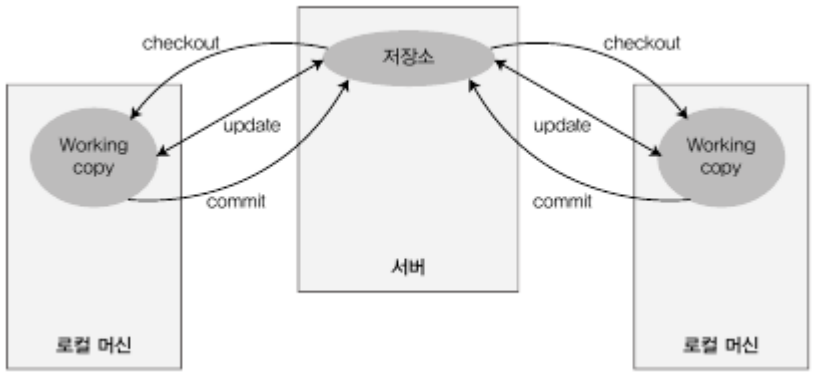
무료/유료	버전 관리 시스템
무료 / 오픈 소스	GNU arch(2001), Darcs(2002), DCVS(2002), ArX(2003), Monotone(2003), SVK(2003), Codeville(2005), Bazaar(2005), Git(2005), Mercurial(2005), Fossil(2007), Veracity (2010)
유료 / 상용	TeamWare(1990년대), Code Co-op(1997), BitKeeper(1998), Plastic SCM(2006)

02. 버전 관리 시스템의 종류

CVS (Concurrent Versions System)

- 클라이언트-서버 방식의 버전 관리 시스템
- 1986년 딕 그룬이 개발
- 서버의 저장소에 프로젝트의 온전한 원본을 보유
- 클라이언트는 서버에서 파일을 가져다가 로컬 저장소에서 변경한 뒤 변경한 내용을 서버에 반영
- 파일 각각의 버전을 관리하고 추적 가능
- 파일 이름의 변경이나 이동을 자동으로 추적하는 것이 어려움
- 원본은 오직 서버에만 존재하여 서버 사고시 프로젝트 복구가 어려울 수 있음

그림 1-6 CVS의 동작 흐름



02. 버전 관리 시스템의 종류

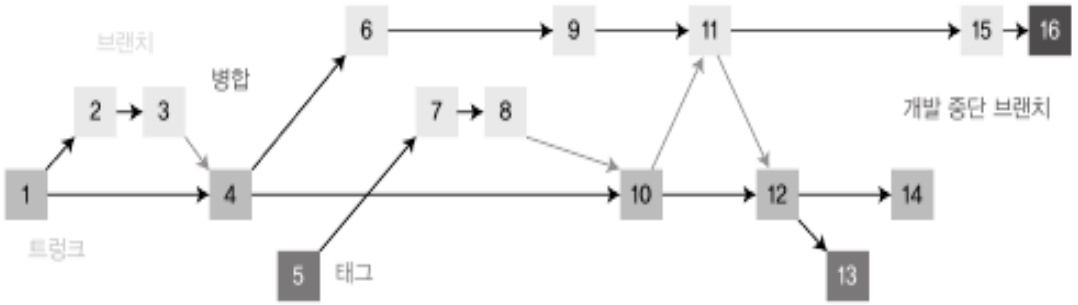
서브버전 (Subversion)

- 2000년 콜렙넷에서 개발
- 아파치의 최상위 프로젝트 중 하나로 유지 보수중
- CVS의 단점을 개선
  - 각각의 커밋이 원자적으로 다른 사용자의 커밋과 엉키지 않고, 커밋 실패 시 롤백이 가능
  - 파일 이름 변경, 복사, 이동, 삭제 등의 작업 내역을 유지하는 히스토리를 만듬
  - 소스파일 이외의 이진파일도 저장가능

02. 버전 관리 시스템의 종류

- **Trunk, Branch, Tag**
  - Trunk: 소스의 주 개발 작업을 진행하는 폴더
  - Branch: 소스의 현재 버전을 유지보수하고, 현재 버전을 기반으로 차기 버전을 개발할 경우 사용하는 폴더
  - Tag: 릴리즈된 소스를 관리하기 쉽게 따로 보관하는 폴더

그림 1-8 서브버전의 브랜칭 개념도[3]



01

2020 삼성 QLED TV, 라인업별 성능 비교!

※대표 모델 기준이며, 모델별 차이가 있을 수 있습니다.

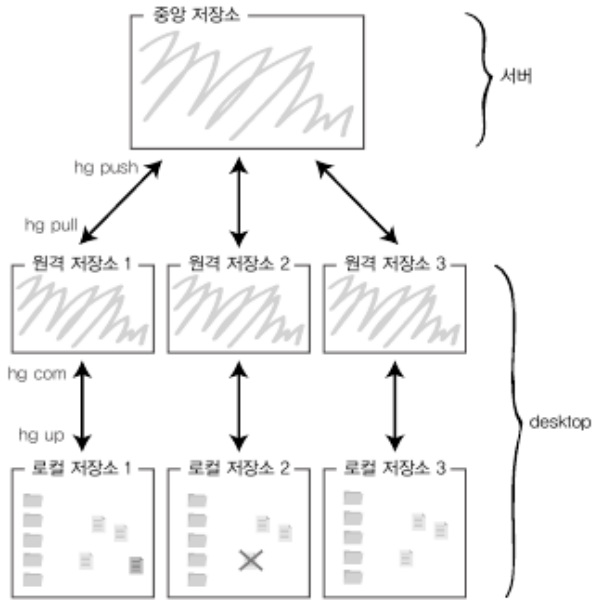
라인업	QT60A	QT70A	QT80A	QT90A	QT800A	QT900S
디자인	슬림 베젤	슬림 베젤	4 베젤리스	슬림 베젤	4 베젤리스	인피니티 스크린
화면 크기	43/50/55/65/**75"**	55/65/75/**85"**	55/65/75/**85"**	55/65/75/**85"**	65/75/**82"**	65/75/**85"**
해상도	4K UHD (3840 x 2160)				**8K UHD (7680 x 4320)**	
화질 엔진	퀀텀 프로세서 4K Lite	퀀텀 프로세서 4K			**퀀텀 프로세서 8K**	
업스케일링	4K 업스케일링	**4K AI 업스케일링**			**8K AI 업스케일링**	
로컬 디밍	엣지형 (로컬 디밍X)		다이렉트 풀어레이 12X	다이렉트 풀어레이 16X	다이렉트 풀어레이 24X	다이렉트 풀어레이 32X
			※엣지형 대비 LED 백라이트 개수가 12, 16, 24, 32배 많아 명암비 상승			
HDR	퀀텀 HDR	퀀텀 HDR	퀀텀 HDR 12X (퀀텀 HDR 1500)	퀀텀 HDR 16X (퀀텀 HDR 2000)	퀀텀 HDR 16X (퀀텀 HDR 2000)	퀀텀 HDR 32X (퀀텀 HDR 3000)
사운드	2.0채널 20W 출력	2.0채널 20W 출력	2.2.2채널 60W 출력	4.2.2채널 60W 출력	4.2.2채널 70W 출력	4.2.2채널 70W 출력

02. 버전 관리 시스템의 종류

머큐리얼

- 2005년 매트 맥컬이 개발한 분산 모델의 버전 관리 시스템
- 각각의 클라이언트가 전체 저장소를 가짐
- 버전 관리 시스템에 필요한 모든 기능을 통합해서 제공
- 명령어가 서브버전과 공통된 것이 많음 (단, 앞에 hg라는 접두어가 붙음)
- 프로젝트의 커밋 내역 변경 불가

그림 1-9 머큐리얼의 동작 흐름



03 GIT

03. GIT

GIT의 장점

- 리누스 토발즈가 리눅스 커널 버전을 관리하기 위해 최초로 개발
- 완벽한 분산 환경에서 빠르고 단순하게 다수의 브랜치 작업을 수행하는 것을 지원
- 리눅스 커널 같은 대형 프로젝트의 버전 관리 기능 제공
- 전 세계의 수 많은 사용자가 사용 중 (현시점 버전 관리 시스템의 표준)
- GIT을 사용한 저장소의 공유 사이트인 GitHub 웹 사이트의 존재
- 방대한 양의 사용자 수를 토대로 한 다양한 오픈소스 소프트웨어가 존재

03. GIT

GIT의 특징

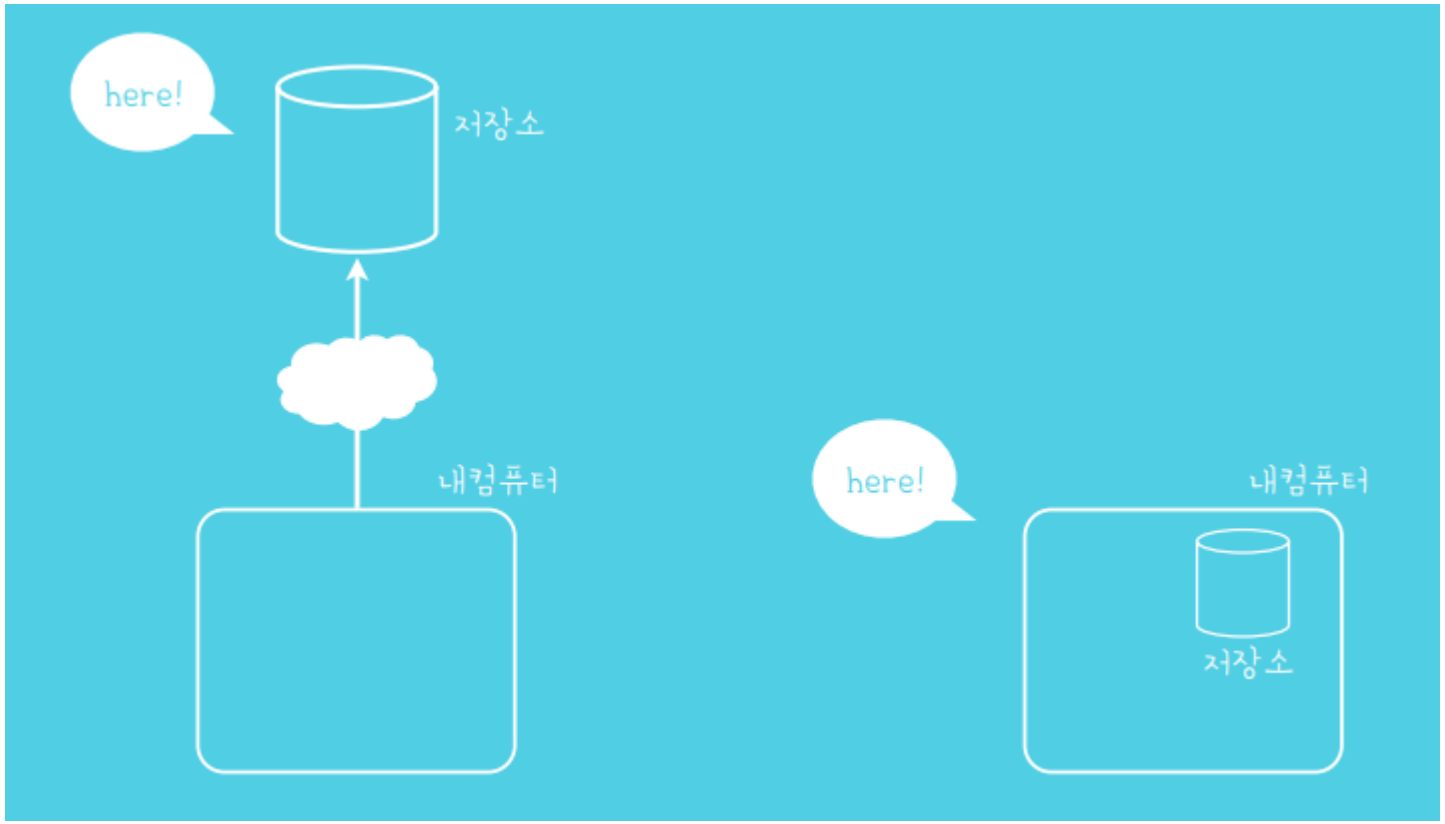
- Master 저장소 서버와 master 저장소의 완벽한 사본을 가지는 클라이언트 저장소로 구성
- 로컬 및 원격 저장소 생성
- 로컬 저장소에 파일 생성 및 추가
- 수정 내역을 로컬 저장소에 제출
- 파일 수정 내역 추적
- 원격 저장소에 제출된 수정 내역을 로컬 저장소에 적용
- Master에 영향을 끼치지 않는 브랜치 생성
- 브랜치 사이에 병합(Merge)
- 브랜치를 병합하는 도중의 충돌 감지

03. GIT

GIT과 SVN의 차이

SVN

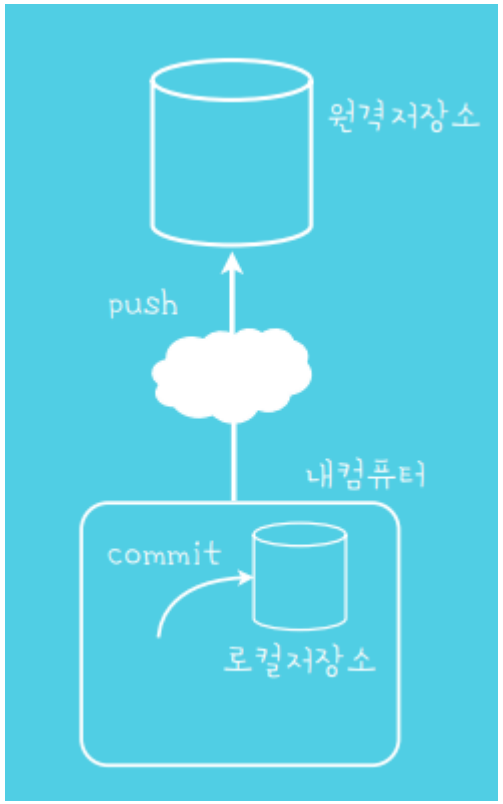
GIT

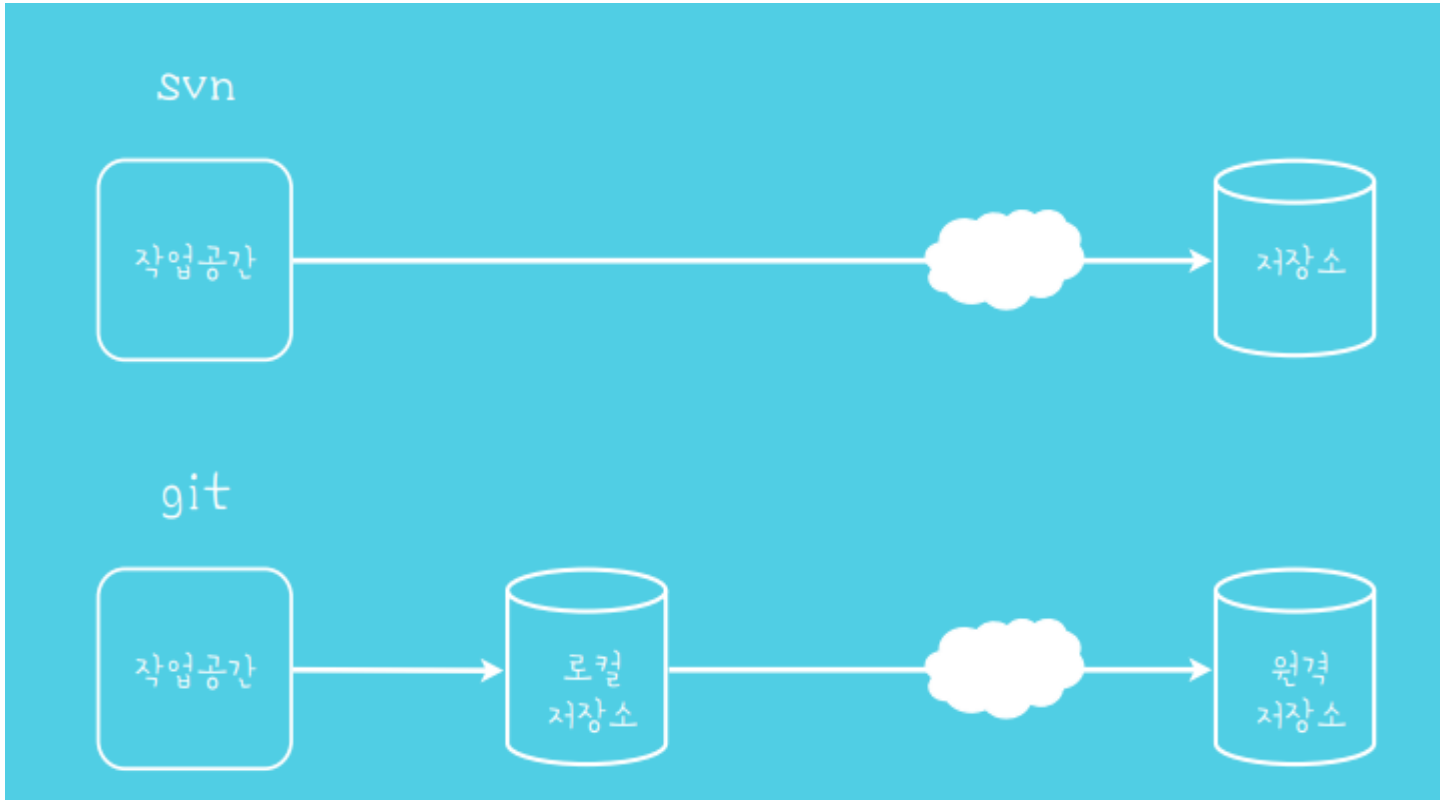


03. GIT

GIT과 SVN의 차이

- 내 컴퓨터의 저장소에 작업 내용을 커밋 (commit)
- 다른 사람과 공유를 위해 원격 저장소에 푸쉬 (Push)

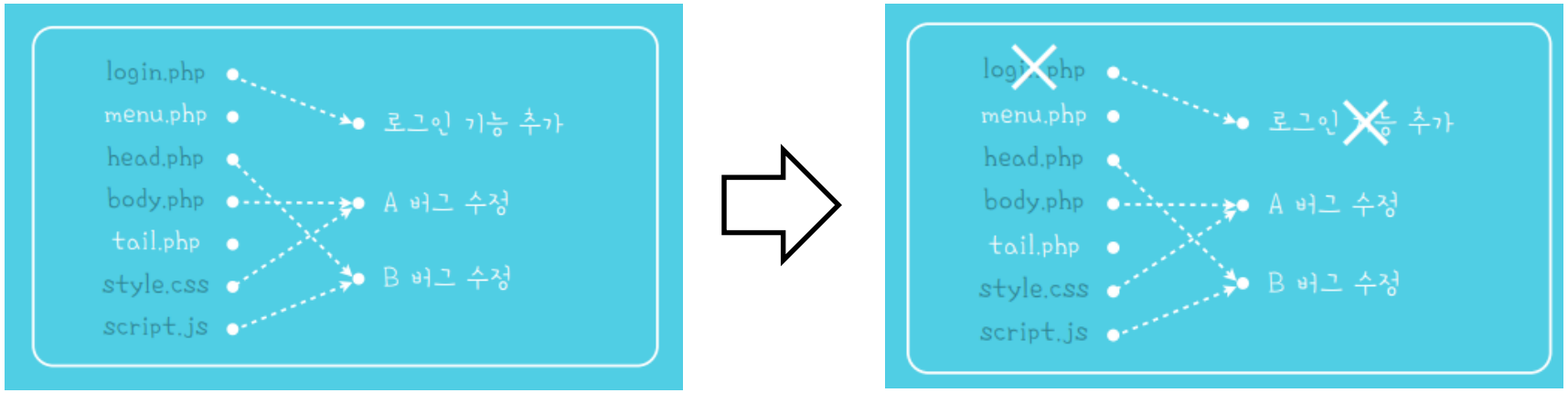


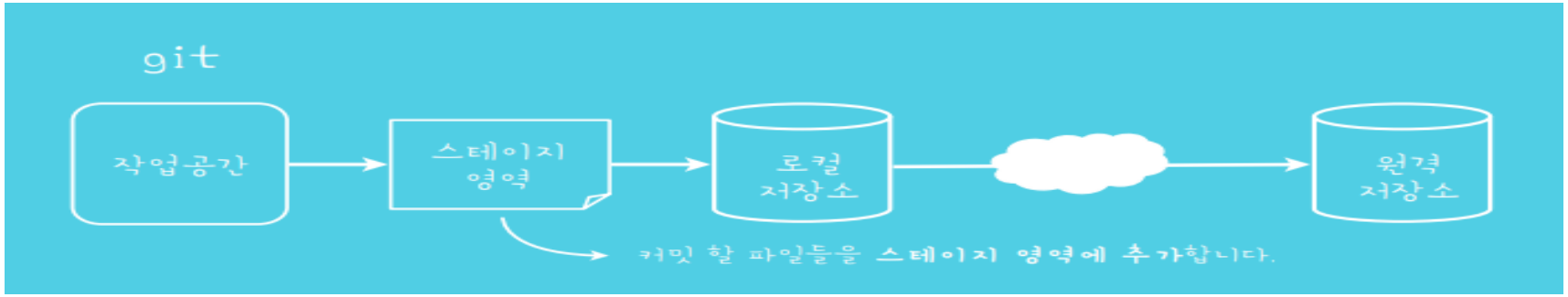


03. GIT

GIT과 SVN의 차이

- 스테이지 영역을 통해 커밋할 파일의 구분이 가능
- 로컬저장소에 커밋 전 인덱스를 통해 스테이징이 가능

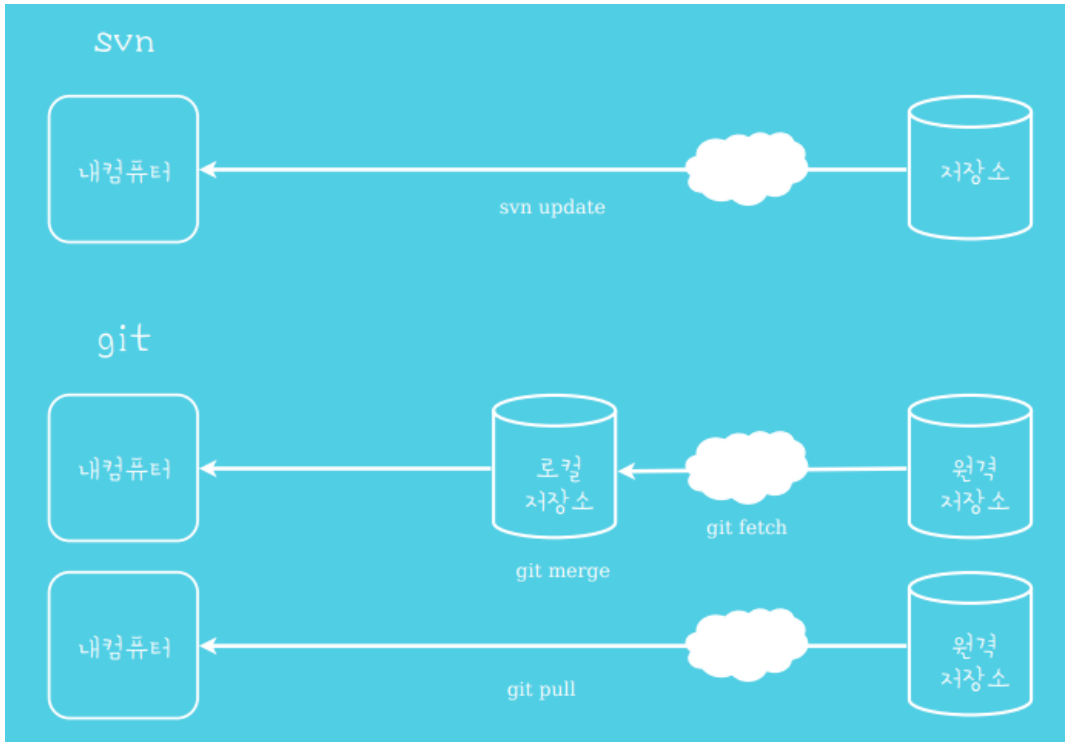




03. GIT

GIT과 SVN의 차이

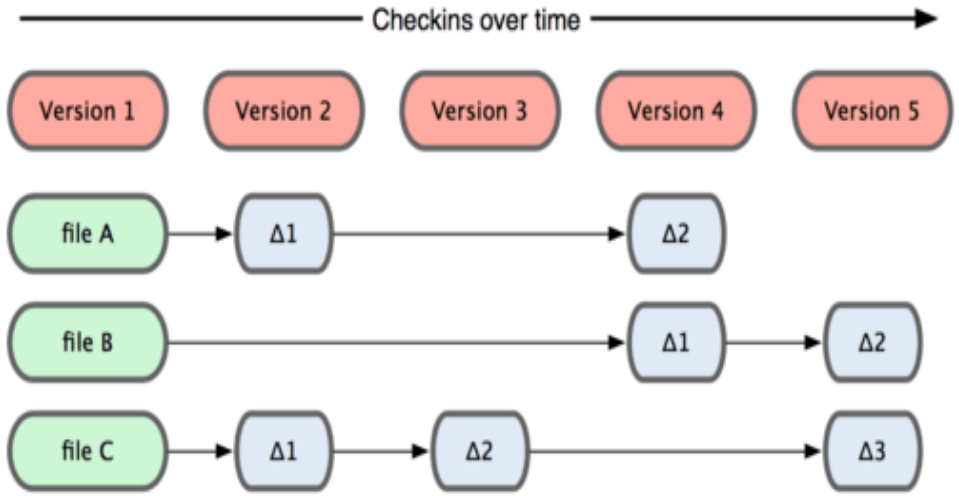
- 원격저장소에 push 전 다른 사용자가 먼저 원격저장소를 변경할 수 있음
- 다른 사람들이 작업 한 내용을 받아 데이터에 병합해야 함 (fetch+merge)
- Pull 명령어를 통해 한번에 수행이 가능

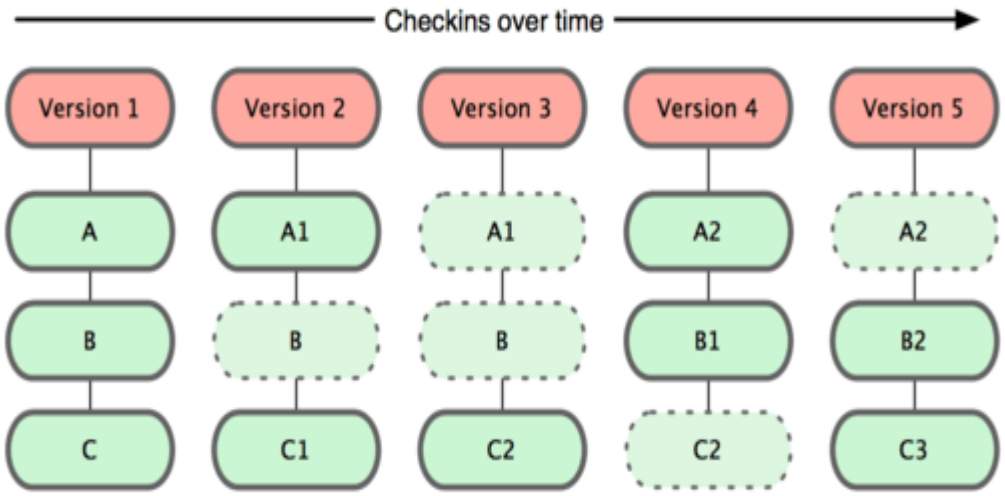


03. GIT

GIT과 SVN의 차이

- SVN의 경우 중앙 저장소에서 버전별로 델타(파일의 변화)를 관리
- 기초가 되는 파일 몇 모든 변경 내역을 서버로부터 내려받아야 함
- GIT의 경우 현재의 서버, 프로젝트 데이터 등을 사진 찍듯이 그대로 찍어서 남겨두는 스냅샷을 활용

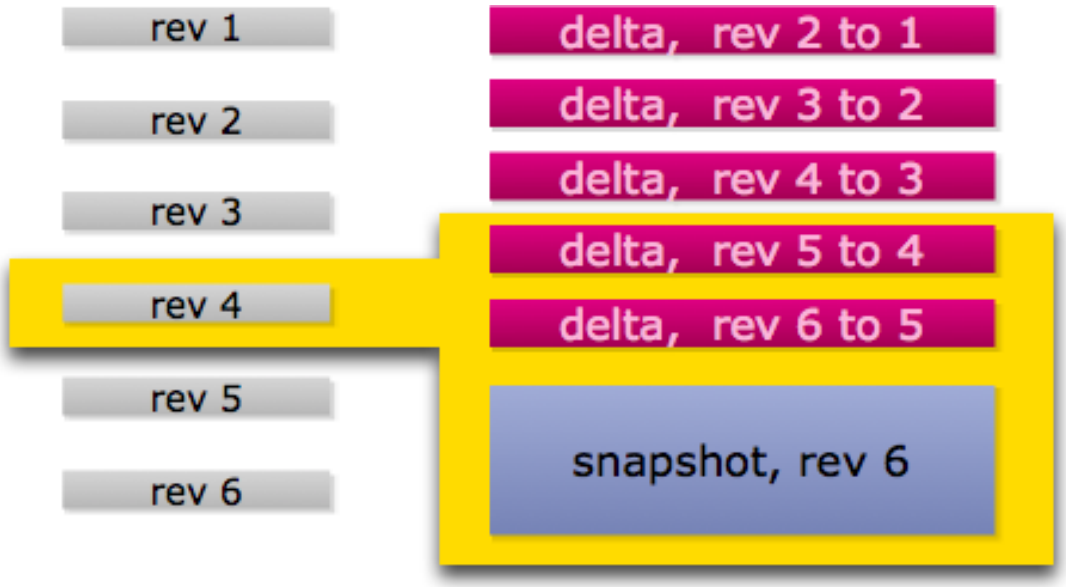




03. GIT

GIT과 SVN의 차이

- GIT은 마지막 커밋의 스냅샷만 저장
- 나머지 커밋의 경우 스냡샷과 스냅샷의 차이를 기록한 델타를 저장
- 이를 통해 저장소의 크기를 효율적으로 관리



03. GIT

GIT을 통한 협업의 장점

- 소스코드를 주고 받을 필요 없이, 병렬 개발이 가능
- 브랜치를 통한 개발 후, Master에 합치는 방식(Merge)을 통해 개발 수행
- 분산 버전 관리이기 때문에 인터넷이 없는 환경에서도 개발 가능
- 인터넷을 경유할 필요가 없어 커밋(commit)이 빠름
- 중앙 저장소가 없어져도, 클라이언트 저장소를 통해 원본 복구 가능
- 프로그램이나 패치를 배포하는 과정도 효율적 (Pull을 통한 업데이트, 패치파일 배포)

03. GIT

GIT 기본 흐름

- 저장소 생성은 최소의 한번만 수행

그림 3-1 기본 작업 과정

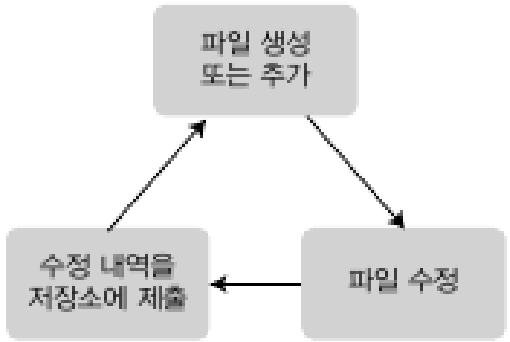


표 3-1 저장소 사용에 필요한 Git 기본 명령어

목표	명령어	설명
저장소 생성	`git init`	실행한 위치를 Git 저장소로 초기화합니다.
저장소에 파일 추가	`git add 파일이름`	해당 파일을 Git이 추적할 수 있게 저장소에 추가합니다.
저장소에 수정 내역 제출	`git commit`	변경된 파일을 저장소에 제출합니다.
저장소 상태 확인	`git status`	현재 저장소의 상태를 출력합니다.

03. GIT

GIT branch를 활용한 작업 흐름

- 안정화된 프로젝트에서 실험적인 기능을 추가하거나 수정할때 사용
- Branch, checkout, merge 등의 명령어를 추가로 활용

그림 3-2 브랜치 이동을 통해 변경된 작업 흐름

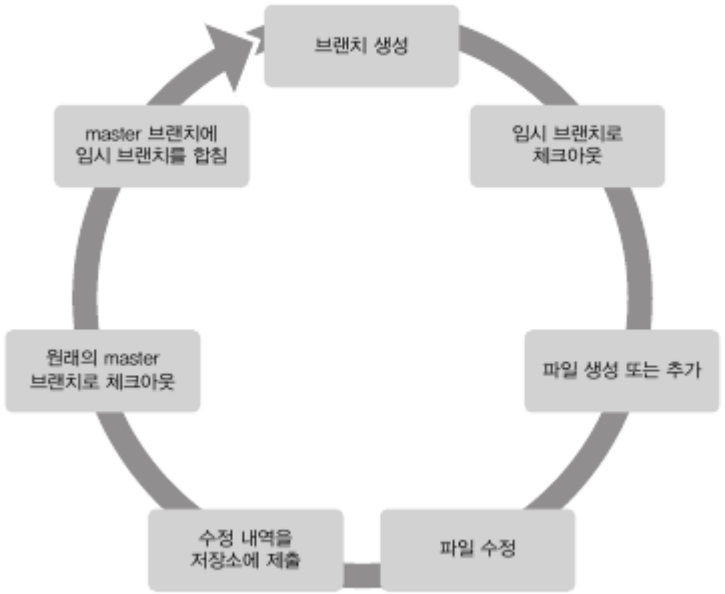


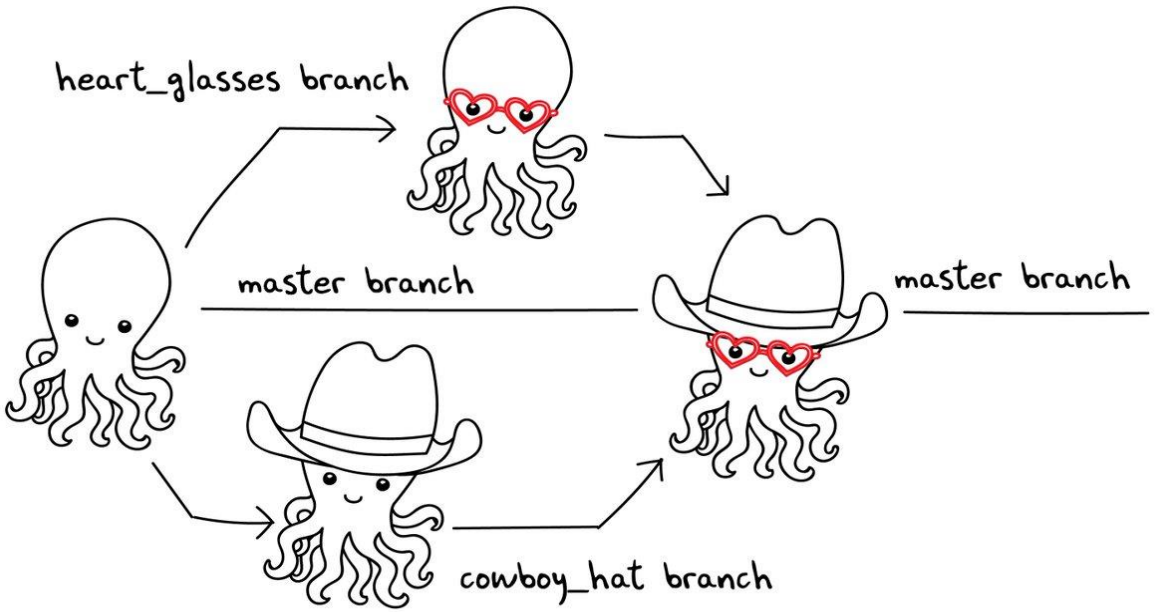
표 3-2 저장소 사용을 위한 branch 명령어

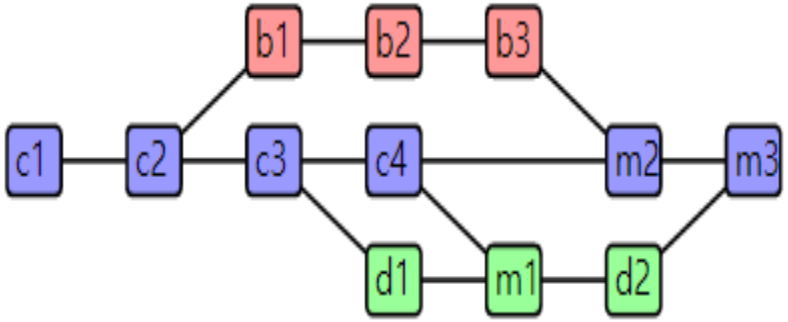
목표	명령어	설명
저장소에 브랜치 추가	`git branch 이름`	'이름'의 브랜치를 만듭니다.
작업 중인 브랜치 변경	`git checkout 브랜치이름`	현재 작업 중인 '브랜치이름'을 변경합니다.
브랜치 병합하기	`git merge 브랜치이름`	현재 작업 중인 브랜치에 '브랜치이름'의 브랜치를 끌어와 병합합니다.

03. GIT

GIT branch를 활용한 작업 흐름

- 안정화된 프로젝트에서 실험적인 기능을 추가하거나 수정할때 사용
- Branch, checkout, merge 등의 명령어를 추가로 활용

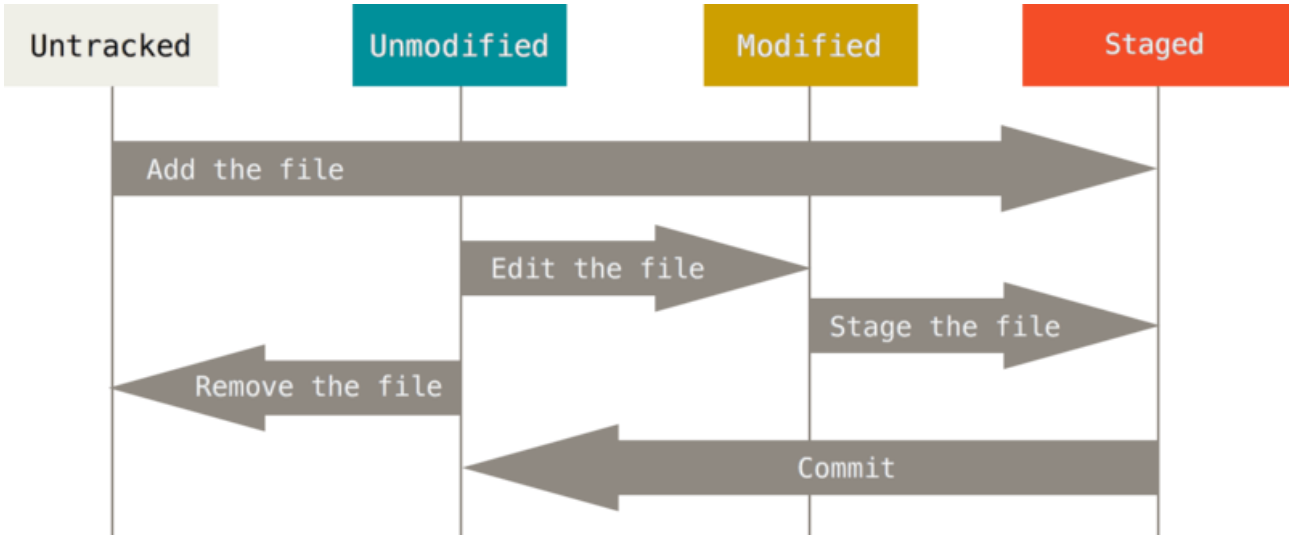




03. GIT

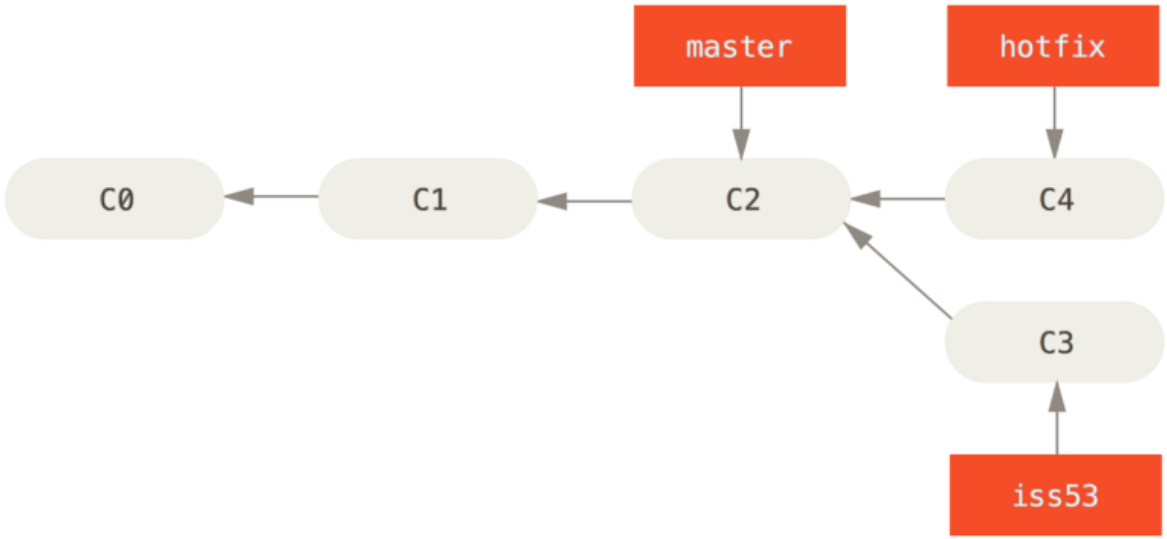
■ 파일 상태 라이프사이클

- Untracked: GIT 저장소에 추가되지 않은 상태. Git이 변경 이력을 관리하지 않으며 스냅샷에도 추가되지 않음
- Tracked: 파일이 Git 저장소에 추가되어서 시스템에 의해 관리되고 있는 상태. 스냅샷에 추가됨
- Unmodified: 수정되지 않은 상태. 마지막 커밋과 비교해서 변경된 부분이 없음
- Modified: 수정된 상태
- Staged: 수정된 파일을 커밋하기 위한 준비 상태. 커밋을 하게 되면 새로운 스냅샷에 생성되고 staged 상태의 파일은 다시 unmodifed로 돌아감



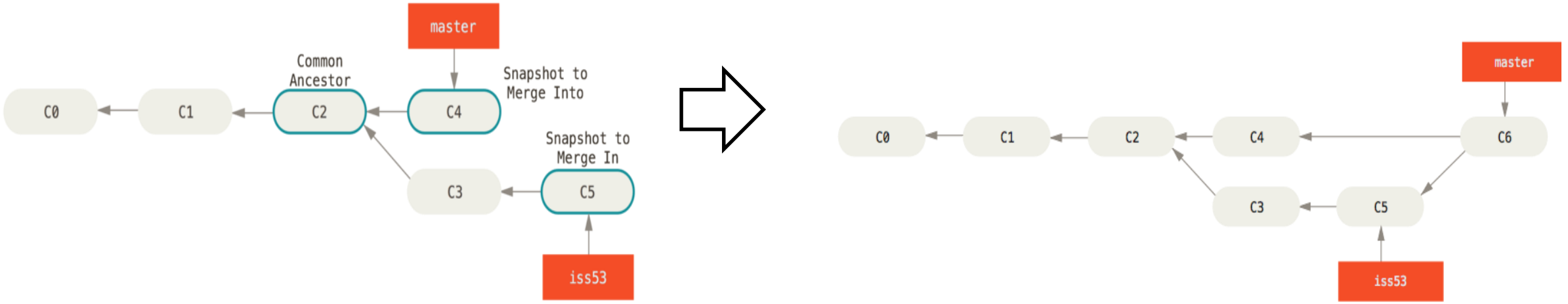
브랜치 병합

- 현재 브랜치가 가르키는 커밋이 병합할 브랜치가 가르키는 커밋의 조상인 경우
- Master는 hotfix의 부모이므로 c2와 c4 커밋을 서로 비교해서 병합을 진행



브랜치 병합

- 현재 브랜치가 가리키는 커밋이 병합할 커밋의 조상이 아닌 경우
  - 3-way merge: 공통 조상 커밋, 병합할 두 브랜치의 커밋을 비교



03. GIT

충돌

- 3-way merge시 서로 다른 브랜치에서 같은 부분을 수정한 경우 충돌이 발생
- 충돌 해결
  - 특정 브랜치의 코드를 사용
  - 코드를 직접 수정
  - 외부 병합 툴을 사용해서 해결

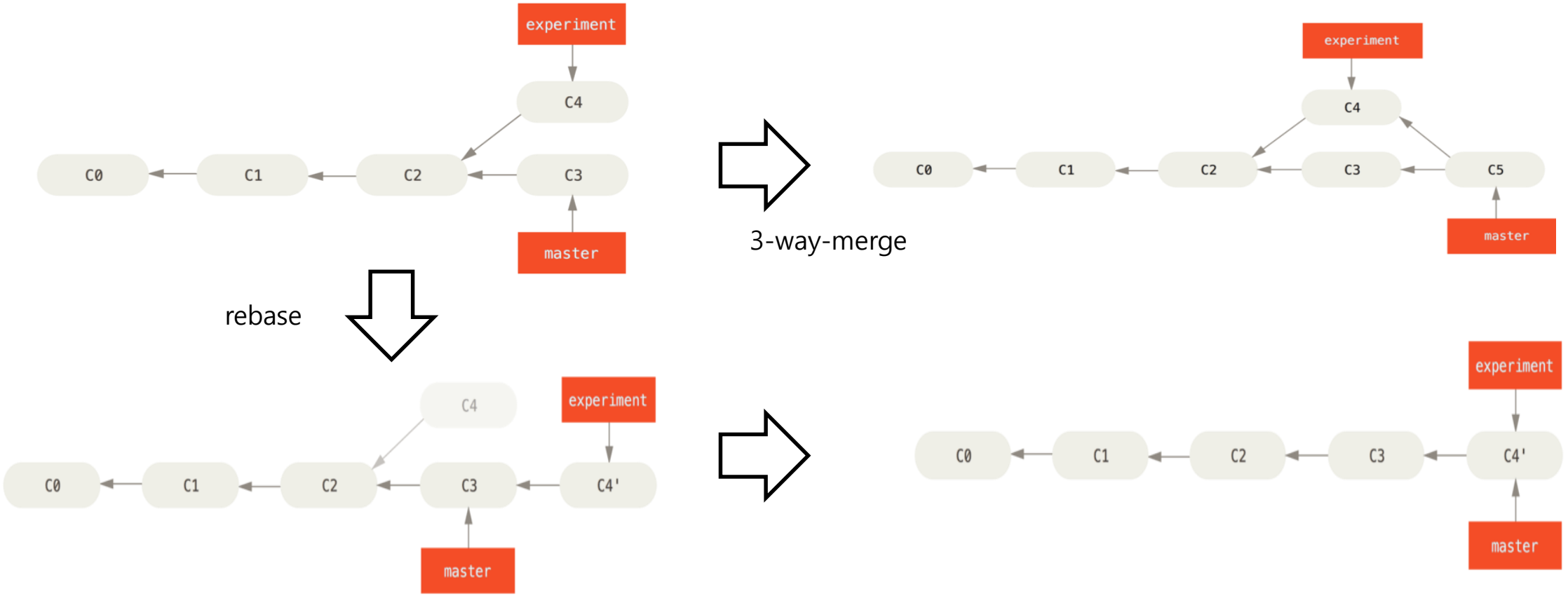
병합 시도시 체크아웃되어 있는 브랜치

병합할 브랜치에서 충돌한 부분

```
<<<<<<< HEAD:index.html
<div id="footer">contact : email.support@github.com</div>
=======
<div id="footer"> please contact us at support@github.com </div>
>>>>>>> iss53:index.html
```

재배치

- 브랜드를 병합하는 다른 방식
- 복수의 브랜치를 커밋 히스토리가 분기된 것처럼 보이지 않고 1개의 브랜치처럼 구성 가능



04 로컬 저장소 사용을 위한 GIT 기본

04. 로컬 저장소 사용을 위한 GIT 기본

■ 로컬 저장소 초기 생성 및 활용

- 저장소 생성은 최초의 한 번만 수행
- Git init
  - 저장소 생성
  - 실행한 위치를 GIT저장소로 추가
- Git add
  - 저장소에 파일 추가
  - 해당 파일을 GIT이 추적할 수 있게 저장소에 추가
- Git commit
  - 저장소에 수정 내역 제출
  - 변경되 파일을 저장소에 제출
- Git status
  - 저장소 상태 확인
  - 현재 저장소의 상태를 출력

로그를 입력

그림 3-1 기본 작업 과정

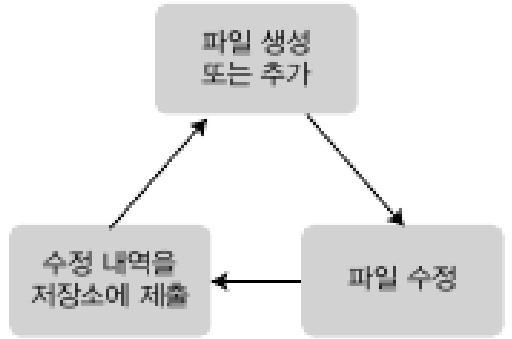


표 3-3 필수 UNIX 명령어

목적	명령	효과
디렉터리 생성	mkdir 디렉터리이름	'디렉터리이름'을 이름으로 갖는 디렉터리를 생성합니다.
파일 내용 출력	cat 파일이름	'파일이름'의 파일 내용을 화면에 출력합니다.
디렉터리 내용물 출력	ls	현재 디렉터리의 내용을 출력합니다.
디렉터리 이동	cd 디렉터리이름	현재 위치에서 접근할 수 있는 '디렉터리이름' 디렉터리로 이동합니다.

04. 로컬 저장소 사용을 위한 GIT 기본

■ 브랜치를 통한 작업 흐름 변경

- 안정화된 프로젝트에서 실험적으로 기능을 추가하거나 수정
- Git branch 이름
  - 저장소에 브랜치 추가
  - '이름'이라는 브랜치를 생성
- Git checkout 브랜치이름
  - 작업 중인 브랜치 변경
  - 현재 작업 중인 '브랜치이름'을 변경
- Git merge 브랜치이름
  - 브랜치 병합하기
  - 현재 작업 중인 브랜치에 '브랜치이름'의 브랜치를 끌어와 병합

그림 3-1 기본 작업 과정

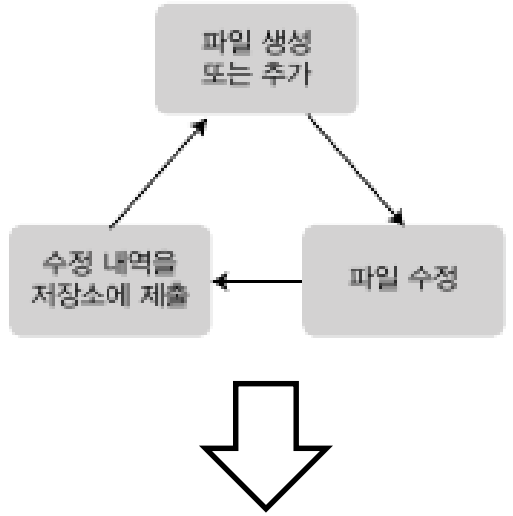
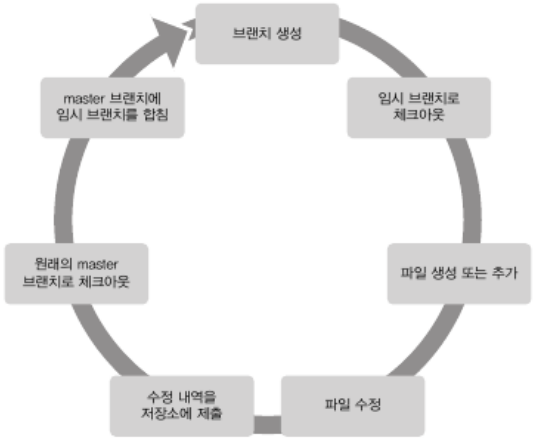


그림 3-2 브랜치 이동을 통해 변경된 작업 흐름



git log --graph
// --stat