Git

버전 관리

vcs

- 버전 관리 시스템
- 버전 관리 시스템은 파일 변화를 시간에 따라 기록했다가 나중에 특정 시점의 버전을 다시 꺼내올 수 있는 시스템이다.
- 각 파일을 이전 상태로 되돌릴 수 있음
- 프로젝트를 통째로 이전 상태로 되돌릴 수 있음
- 시간에 따라 수정 내용을 비교해 볼 수 있음
- 누가 문제를 일으켰는지 추적할 수 있음

DVCS(분산 버전 관리 시스템)

- git, mecurial, bazaar, darcs
- 저장소를 히스토리와 더불어 전부 복제
- 대부분의 dvcs 환경에서는 리모트 저장소가 존재

세 가지 상태

commited

- 데이터가 로컬 데이터베이스에 안전하게 저장된 상태

modified

- 수정한 파일을 아직 로컬 데이터베이스에 커밋하지 않은 상태

staged

- 현재 수정한 파일을 곧 커밋할 것이라고 표시한 상태

사용법

git 저장소 만들기

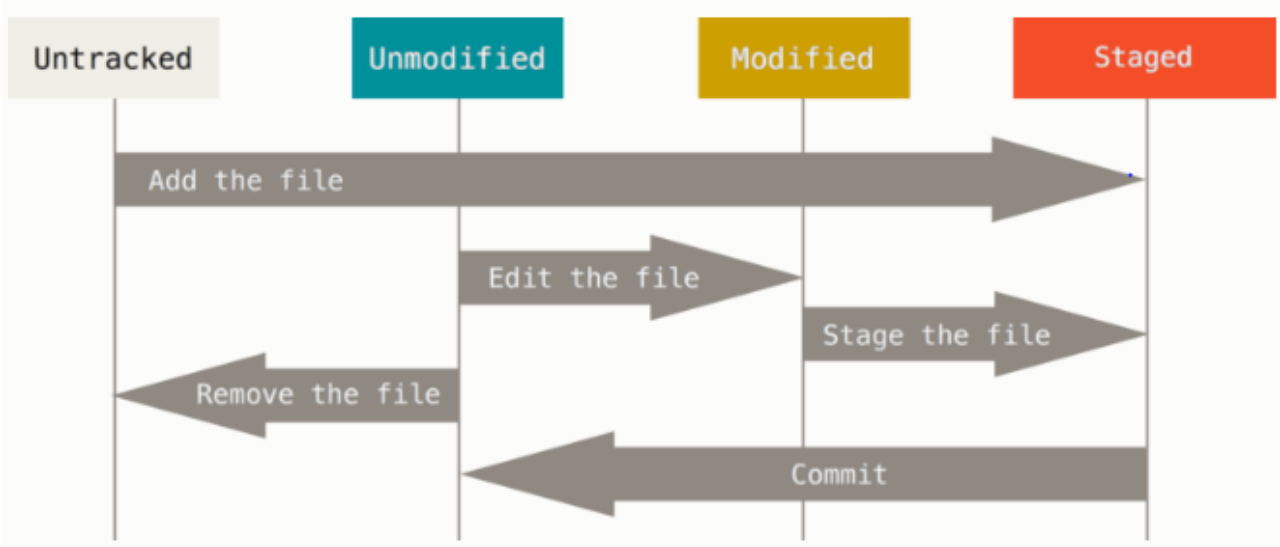
- 아직 버전관리를 하지 않는 로컬 디렉토리를 git 저장소 적용

```
git init
git add .
git commit -m "  "
```

- 기존 저장소를 clone

```
git clone https://github.com/libgit2/libgit2
```

commit 사이클



파일의 상태 확인

```
git status
```

- 현재 작업 중인 브랜치를 알려줌

스테이징

```
git add .
git add 파일명
```

파일 무시하기

- .gitignore

파일 변경 내용 보기

```
git diff
```

- unstaged 상태인 것들만 보여줌

변경 사항 커밋하기

```
git commit
git commit -m "sjakl;fasjd" // inline
```

파일 삭제

```
git rm filename
```

파일 이름 변경

```
git mv file_from file_to
```

커밋 히스토리 조회

```
git log
git log -p
git log -p -2
```

옵션	설명
-p	각 커밋에 적용된 패치를 보여준다.
--stat	각 커밋에서 수정된 파일의 통계정보를 보여준다.
--shortstat	--stat 명령의 결과 중에서 수정한 파일, 추가된 라인, 삭제된 라인만 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 weeks ago" 처럼 상대적인 형식으로 보여준다.
--graph	브랜치와 머지 히스토리 정보까지 아스키 그래프로 보여준다.
--pretty	지정한 형식으로 보여준다. 이 옵션에는 oneline, short, full, fuller, format이 있다. format은 원하는 형식으로 출력하고자 할 때 사용한다.
--oneline	--pretty=oneline --abbrev-commit 두 옵션을 함께 사용한 것과 같다.

되돌리기

- 이전 커밋 수정

```
git commit --amend
```

- 파일 상태 unstage로 변경

```
git reset HEAD filename
```

- modified 파일 되돌리기

```
git checkout -- filename
```

리모트 저장소

- 인터넷이나 네트워크 어딘가에 있는 저장소
- 다른 사람들과 함께 일한다는 것은 리모트 저장소를 관리하면서 데이터를 psuh하고 pull하는 것

리모트 저장소 확인

```
git remote
```

리모트 저장소 추가

- url 대신 name을 쓸 수 있음

```
git remote add name url
```

리모트 저장소 데이터 가져오기

```
git fetch name
```

- 로컬에는 없지만, 리모트 저장소에는 있는 데이터를 모두 가져옴
- 저장소를 clone 하면 자동으로 리모트 저장소를 origin이라는 이름으로 추가한다

```
git pull
```

- 리모트 저장소에서 데이터를 가져와 merge까지 해준다

리모트 저장소에 **push**

- 프로젝트를 공유하고 싶을 때 upstream 저장소에 push 할 수 있다.
- master 브랜치를 origin 서버에 push

```
git push origin master
```

리모트 저장소 살펴보기

```
git remote show origin
```

리모트 저장소 이름 변경, 삭제

```
git remote rename name_from name_to
```

```
git remote remove name
```

브랜치

- 독립적으로 개발하는 것
- 브랜치를 만들어 작업하고, 나중에 merge

브랜치 생성

```
git branch name
```

- 새로 만든 브랜치도 지금 작업하고 있던 마지막 커밋을 가리킴

HEAD

- 지금 작업하는 로컬 브랜치를 가리킨다

브랜치 이동

```
git checkout name
```

- 브랜치를 이동하면 워킹 디렉토리의 파일이 변경된다

브랜치 merge

```
git checkout -b name
```

동일 기능

```
git branch name
git checkout name
```

- 브랜치를 이동하려면 아직 커밋하지 않은 파일이 checkout 할 브랜치와 충돌 나면 브랜치를 변경할 수 없다.

- 브랜치를 변경할 때는 워킹 디렉토리를 정리하는 것이 좋음

- hotfix 브랜치와 master 브랜치를 merge

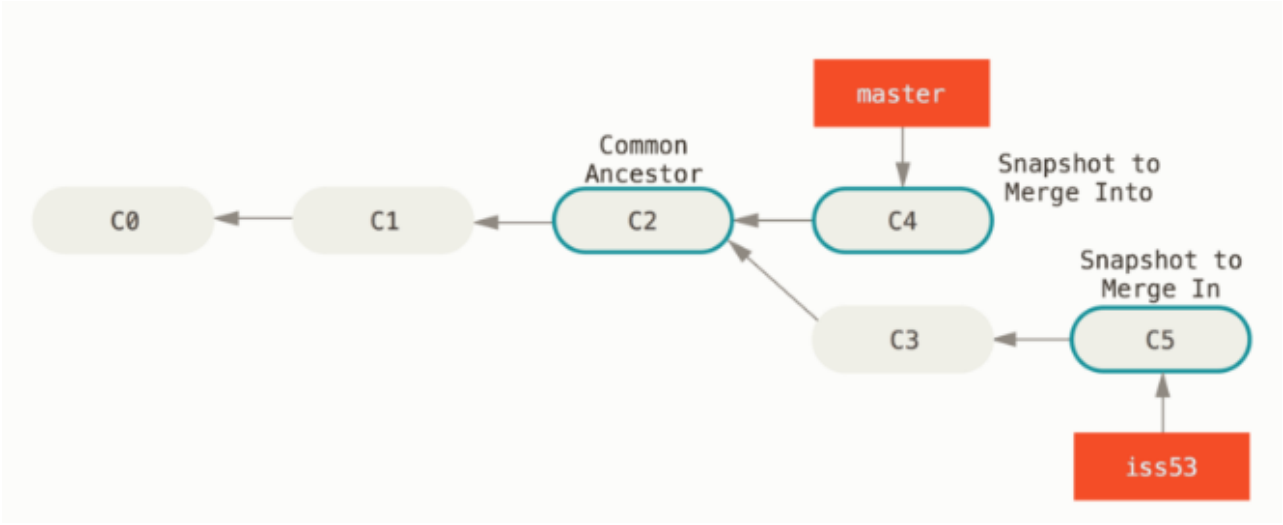
```
git checkout master
git merge hotfix
```

- Fast forward
  - merge 과정 없이 그저 최신 커밋으로 이동하는 것

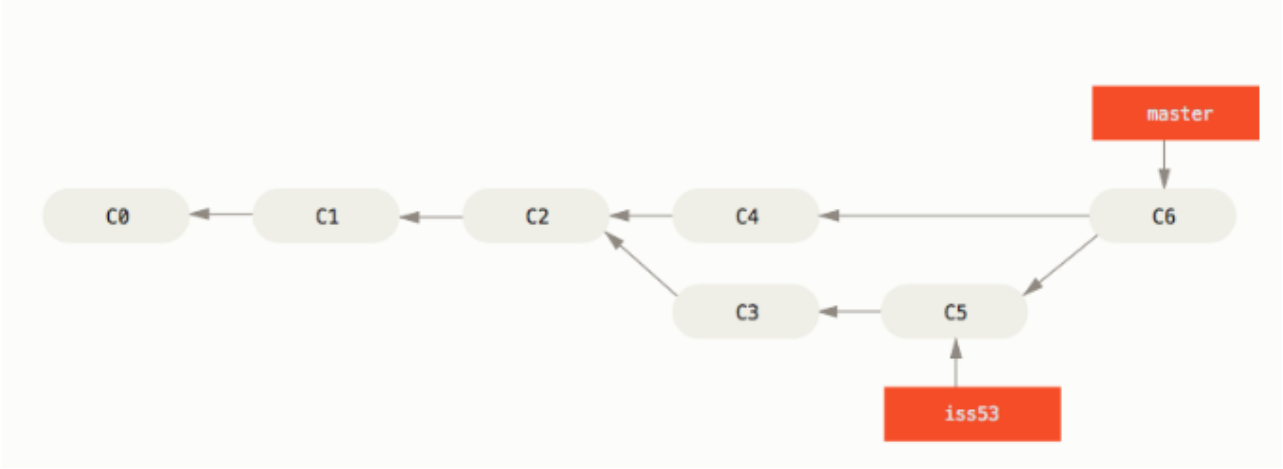
브랜치 삭제

```
git branch -d hotfix
```

- merge 전



- merge 후



충돌

- merge하는 두 브랜치에서 같은 파일의 한 부분을 동시에 수정하고 merge하면 git은 해당 부분을 merge하지 못함
- 충돌이 일어났을 때 git이 어떤 파일을 merge 할 수 없었는지 살펴보려면 git status를 사용한다.

- git은 충돌이 난 부분을 표준 형식에 따라 표시해줌, 그러면 개발자는 해당 부분을 수동으로 해결한다

- 충돌 났을 때 수정하는 명령어

```
git mergetool
```

브랜치 관리

```
git branch
```

- 브랜치의 목록을 보여준다

브랜치 워크플로

브랜치를 만들고 Merge 하는 것을 어디에 써먹어야 할까. 이 절에서는 Git 브랜치가 유용한 몇 가지 워크플로를 살펴본다. 여기서 설명하는 워크플로를 개발에 적용하면 도움이 될 것이다.

Long-Running 브랜치

Git은 꼼꼼하게 3-way Merge를 사용하기 때문에 장기간에 걸쳐서 한 브랜치를 다른 브랜치와 여러 번 Merge 하는 것이 쉬운 편이다. 그래서 개발 과정에서 필요한 용도에 따라 브랜치를 만들어 두고 계속 사용할 수 있다. 그리고 정기적으로 브랜치를 다른 브랜치로 Merge 한다.

이런 접근법에 따라서 Git 개발자가 많이 선호하는 워크플로가 하나 있다. 배포했거나 배포할 코드만 `master` 브랜치에 Merge 해서 안정 버전의 코드만 `master` 브랜치에 둔다. 개발을 진행하고 안정화하는 브랜치는 `develop` 이나 `next` 라는 이름으로 추가로 만들어 사용한다. 이 브랜치는 언젠가 안정 상태가 되겠지만, 항상 안정 상태를 유지해야 하는 것이 아니다. 테스트를 거쳐서 안정적이라고 판단되면 `master` 브랜치에 Merge 한다. 토픽 브랜치(앞서 살펴본 `iss53` 브랜치 같은 짧은 호흡 브랜치)에도 적용할 수 있는데, 해당 토픽을 처리하고 테스트해서 버그도 없고 안정적이면 그때 Merge 한다.

사실 우리가 얘기하는 것은 커밋을 가리키는 포인터에 대한 얘기다. 커밋 포인터를 만들고 수정하고 분리하고 합치는지에 대한 것이다. 개발 브랜치는 공격적으로 히스토리를 만들어 나아가고 안정 브랜치는 이미 만든 히스토리를 뒤따르며 나아간다.

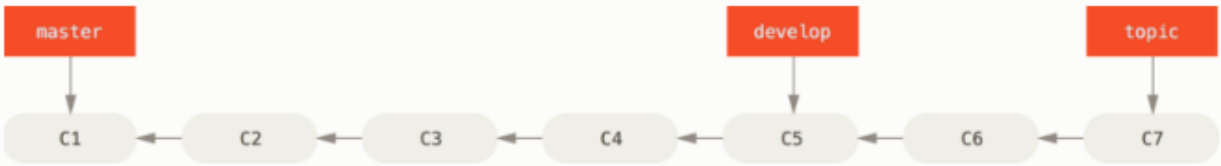
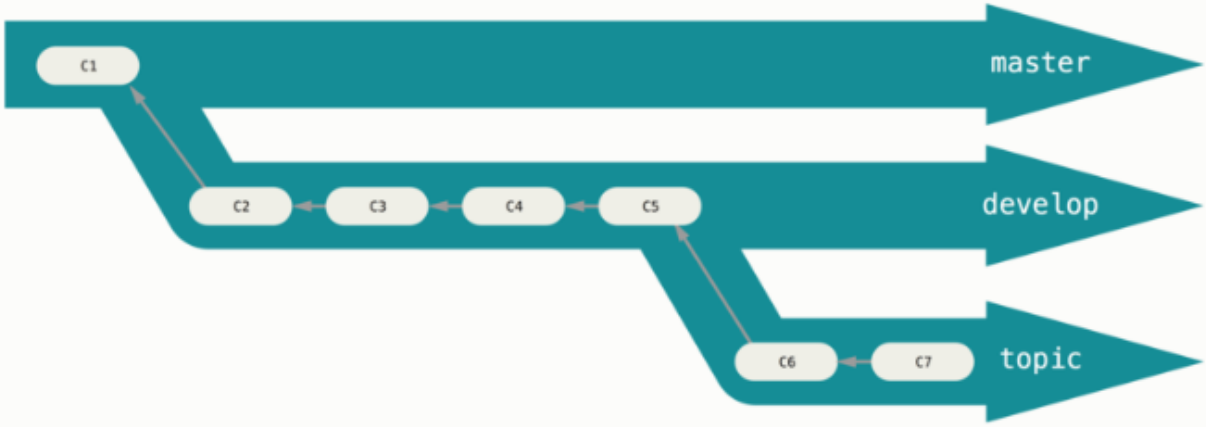


Figure 26. 안정적인 브랜치일수록 커밋 히스토리가 뒤쳐짐
실험실에서 충분히 테스트하고 실전에 배치하는 과정으로 보면 이해하기 쉽다



리모트 브랜치

- 리모트 트래킹 브랜치 - <remote>/<branch> 형식으로 되어 있음

- 예를 들어 리모트 저장소 origin의 master 브랜치를 보고 싶다면 origin/master라는 이름으로 확인

- 리모트 서버로부터 저장소 정보를 동기화하려면

```
git fetch origin
```

  명령을 사용한다
- 명령을 실행하면 우선 "origin" 서버의 주소 정보를 찾아서, 현재 로컬의 저장소가 갖고 있지 않은 새로운 정보가 있으면 모두 내려받고, 받은 데이터를 로컬 저장소에 업데이트하고 나서 origin/master 포인터의 위치를 최신 커밋으로 이동시킨다.

push하기

- 로컬의 브랜치를 서버로 전송하려면 리모트 저장소에 push 해야한다.

새로 받은 브랜치의 내용을 merge

```
git checkout -b branch origin/branch
```

- 그러면 origin/branch에서 시작하고 수정할 수 있는 branch라는 로컬 브랜치가 만들어진다

- 트래킹 브랜치는 리모트 브랜치와 직접적인 연결고리가 있는 로컬 브랜치이다.

```
git pull
```

- 트래킹 브랜치에서 git pull을 하면 리모트 저장소로부터 데이터를 내려받아 연결된 리모트 브랜치와 자동으로 merge한다.

- 서버로부터 저장소를 clone 하면 git은 자동으로 master 브랜치를 origin/master 브랜치의 트래킹 브랜치로 만든다.

```
git checkout -b <branch> <remote>/<branch>
```

- 트래킹 브랜치 생성

pull == fetch + merge

리모트 브랜치 삭제

```
git push origin --delete <branch>
```

- 위 명령을 실행하면 서버에서 브랜치 하나가 사라진다