NHN Academy

Git & GitHub

목차

NHN Academy

- 버전 관리 시스템
- Git 기초
- 기초 설정
- Git 기초 사용법
- Git/GitHub/Dooray!Project를 이용한 프로젝트 관리

NHN Academy

버전 관리 시스템

버전 관리 시스템 [Look]

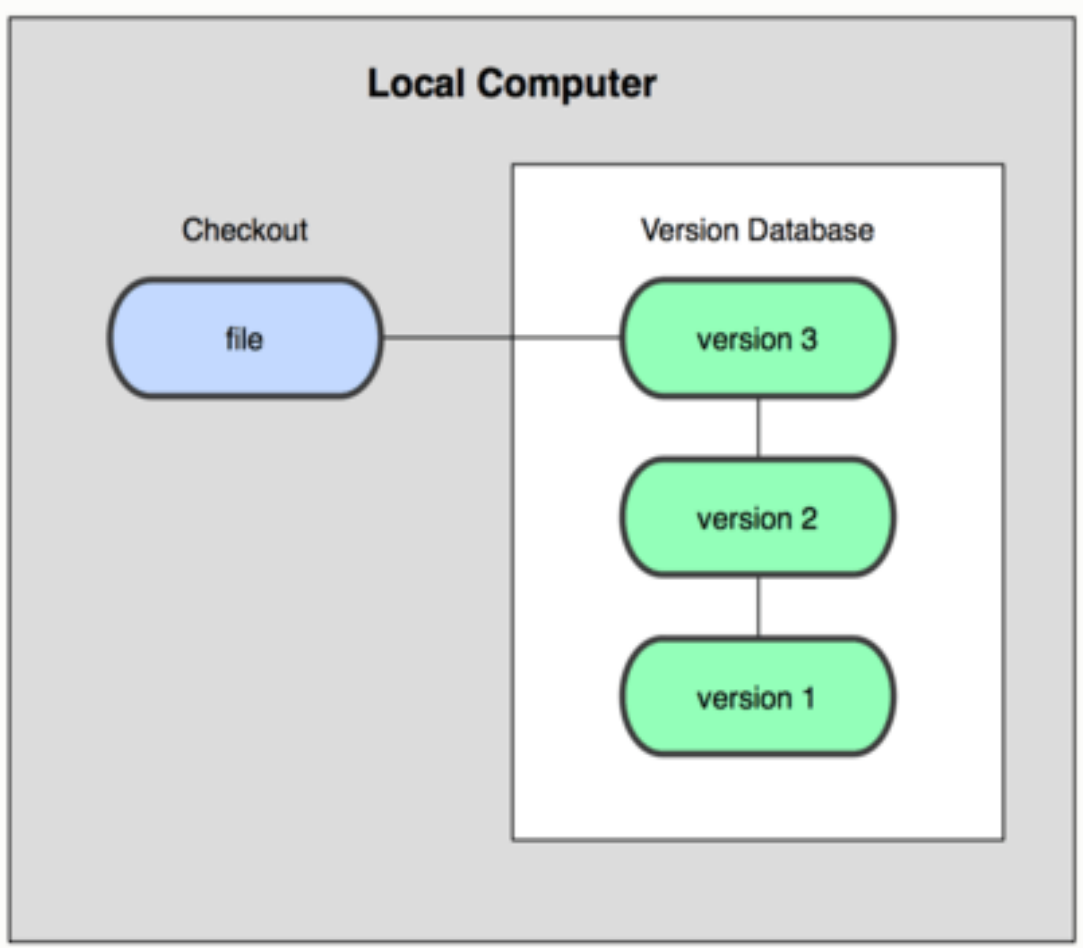
NHN Academy

· Version Control System; VCS
· 파일의 변화를 시간에 따라 기록하여 과거 특정 시점의 버전으로 다시 불러올 수 있는 시스템
· 개별 파일 혹은 프로젝트 전체를 이전 상태로 되돌리기
· 시간에 따른 변경 사항을 검토
· 문제가 되는 부분을 누가 마지막으로 수정하였는지 찾기
· 파일을 잃어버리거나 무언가 잘못되어도 대개 쉽게 복구 가능

로컬 버전 관리 시스템 [Look]

NHN Academy

· 대부분의 사람들이 버전 관리를 위해 쓰는 방법은 파일을 다른 디렉토리, 다른 이름으로 복사하는 것
· 이 방법은 간단하지만 실수가 발생하기 쉬움
  · 파일을 덮어쓰거나 의도하지 않은 위치로 복사
· 이 문제를 해결하기 위하여 오래전 프로그래머들은 간단한 데이터베이스에 파일의 변경 사항을 기록하는 로컬 버전 관리 시스템을 만듦
  ex) RCS



중앙집중식 버전 관리 시스템 [Look]

NHN Academy

- 여러 개발자들과 함께 작업하려면?
- 중앙집중식 버전 관리 시스템(Centralized Version Control System; CVCS) 탄생

  ex) CVS, Subversion, Perforce
- 파일을 저장하는 하나의 서버
- 중앙 서버에서 파일을 가져오는 다수의 클라이언트

중앙집중식 버전 관리 시스템 [Look]

장점

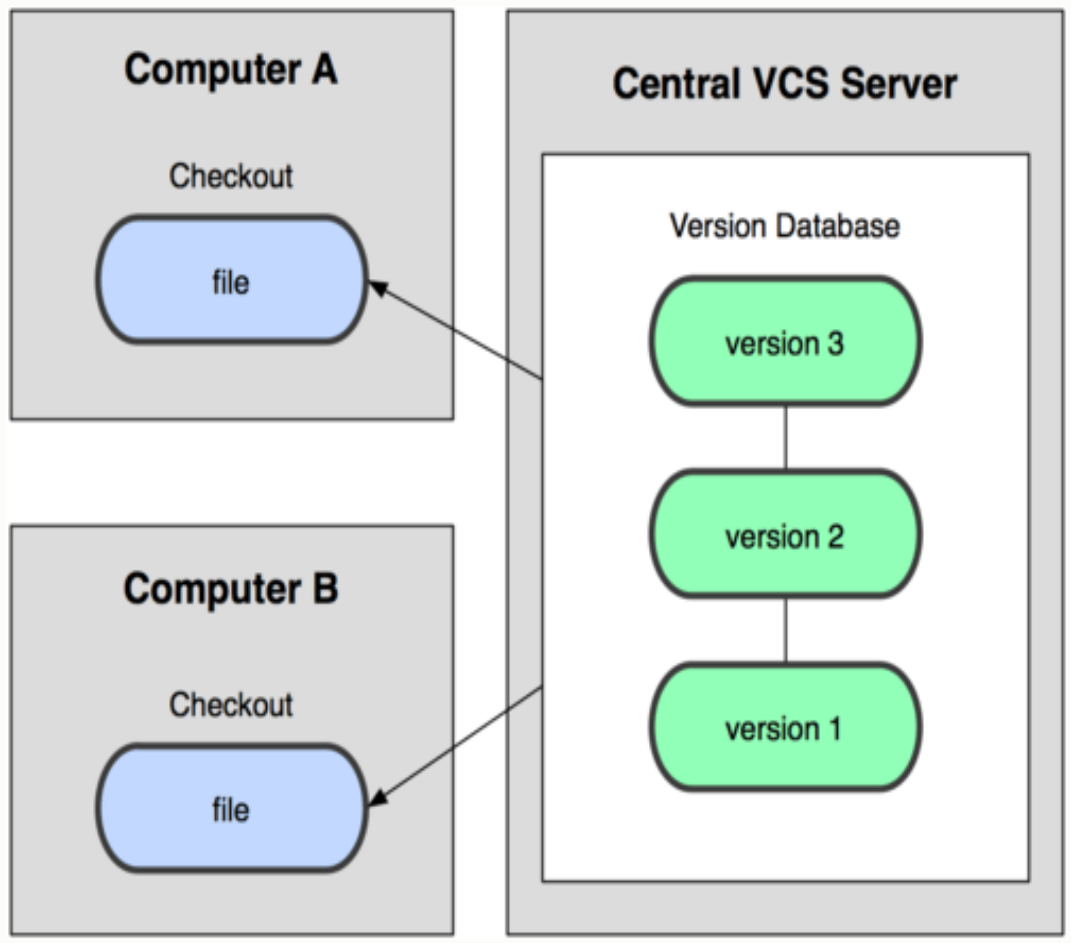
· 누구나 다른 사람이 무엇을 하고 있는지 알 수 있음

· CVCS를 관리하는 것은 수많은 클라이언트의 로컬 데이터베이스를 관리하는 것보다 쉬움

단점

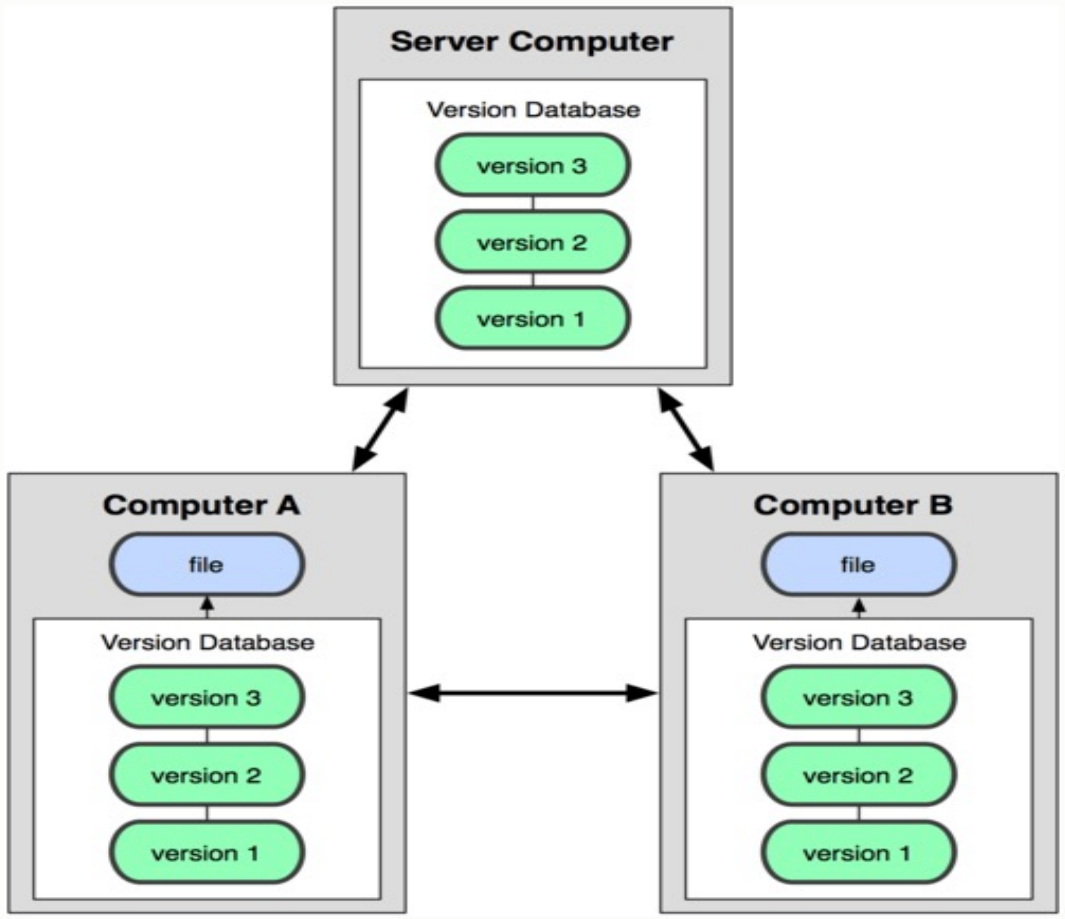
· 중앙 서버가 잘못되면 모든 것이 잘못된다

· 서버가 다운될 경우 다시 복구할 때까지 다른 사람과 협업도 진행 중이던 작업을 버전 관리하는 것도 불가능



분산 버전 관리 시스템 [Look]

- Distributed Version Control System; DVCS
  - Git, Mercurial, Bazaar, Darcs
- 클라이언트가 저장소를 통째로 복사
- 서버에 문제가 생겨도 어느 클라이언트는 복제된 저장소를 다시 서버로 복사하면 서버가 복구됨



NHN Academy

Git 기초

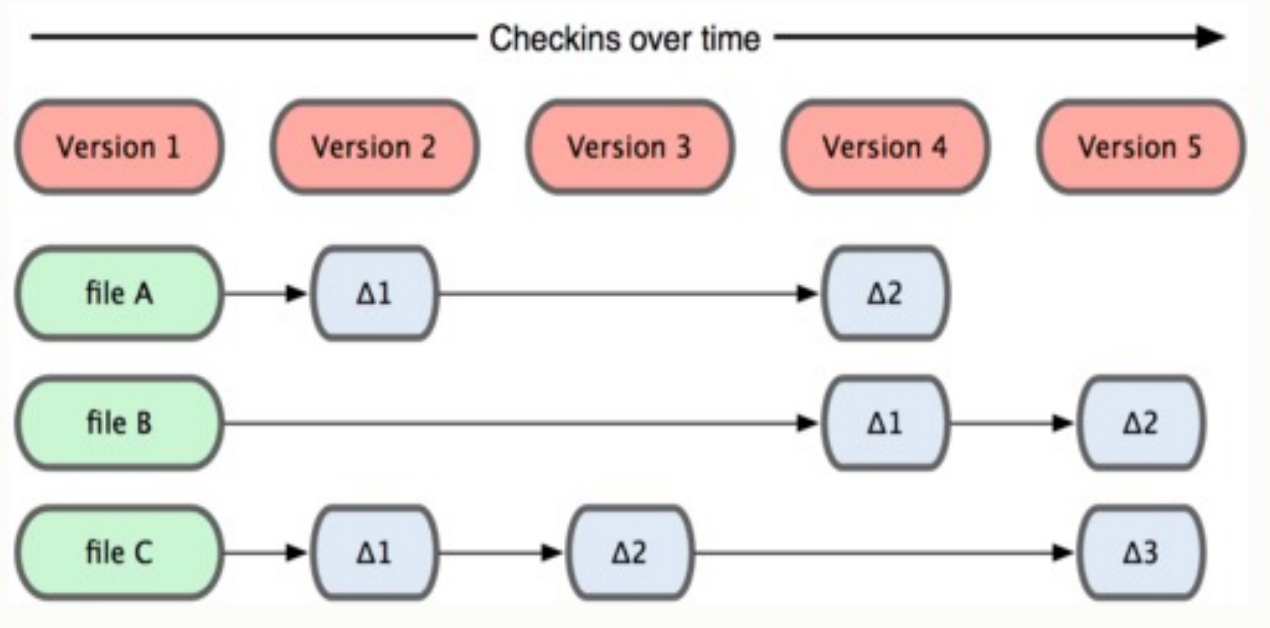
Git 역사 [Look]

- 리눅스 커널은 패치 파일와 단순 압축 파일로 관리
- 2002년 BitKeeper라는 상용 DVCS를 사용
- 2005년에 BitKeeper의 무료 사용이 제고됨
- 리눅스 개발 커뮤니티가 자체 도구를 만드는 계기
- 목표
  - 빠른 속도
  - 단순한 구조
  - 비선형적인 개발 (수천 개의 동시 다발적인 브랜치)
  - 완벽한 분산
  - 리눅스 커널 같은 대형 프로젝트에도 유용할 것(속도나 데이터 크기 면에서)
- 2005년 Git 탄생

델타가 아니라 스냅샷 [Look]

Subversion이나 비슷한 VCS

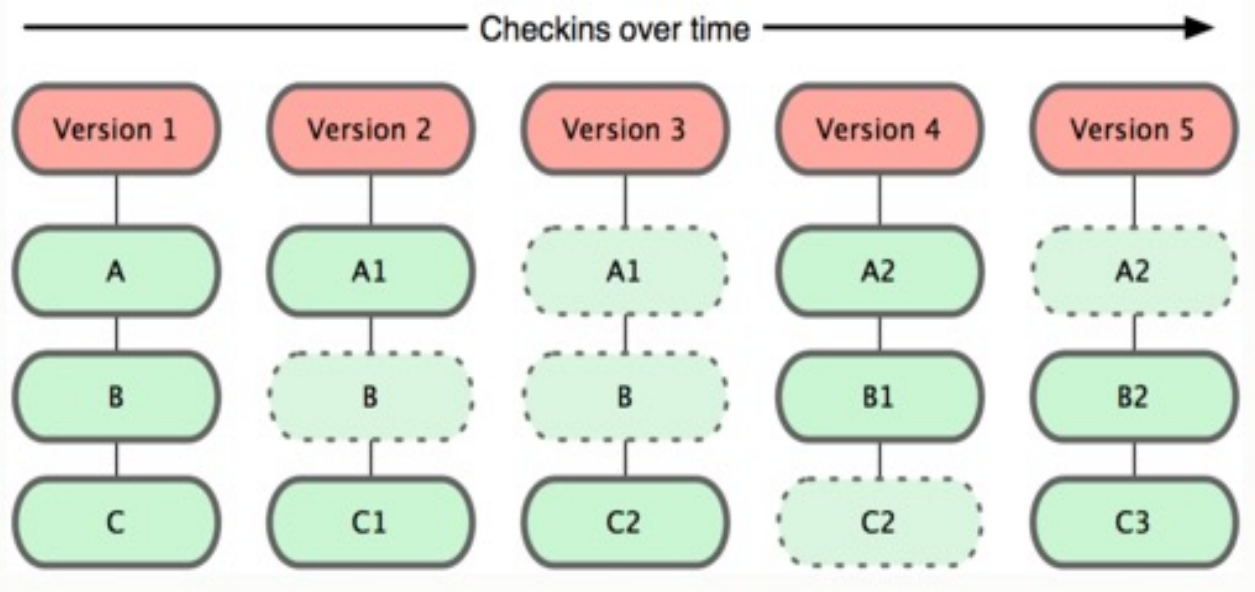
· 각 파일의 변화를 시간순으로 관리



델타가 아니라 스냅샷 [Look]

Git

- Git의 데이터는 파일의 스냅샷
- 파일이 달라지지 않으면 이전 버전의 링크만 저장



거의 모든 명령을 로컬에서 실행 [Look]

NHN Academy

- 거의 모든 명령이 로컬 파일과 데이터만 사용
- 프로젝트의 모든 히스토리가 로컬 디스크에 있기 때문에 모든 명령을 순식간에 실행

  ex) 프로젝트의 히스토리를 서버 없이 조회 -> 빠른 조회 가능
- 비행기나 기차 등에서 작업하고 네트워크에 접속하고 있지 않아도 커밋 가능

Git의 무결성 [Look]

NHN Academy

· 모든 데이터를 저장하기 전 체크섬(해시)을 구하고 그 체크섬으로 데이터를 관리

· SHA-1 해시 사용

· 40자 길이의 16진수 문자열

ex) 24b9da6552252987aa493b52f8696cd6d3b00373

· 파일을 이름으로 저장하지 않고 해당 파일의 해시로 저장

세 가지 상태 [Look]

Committed

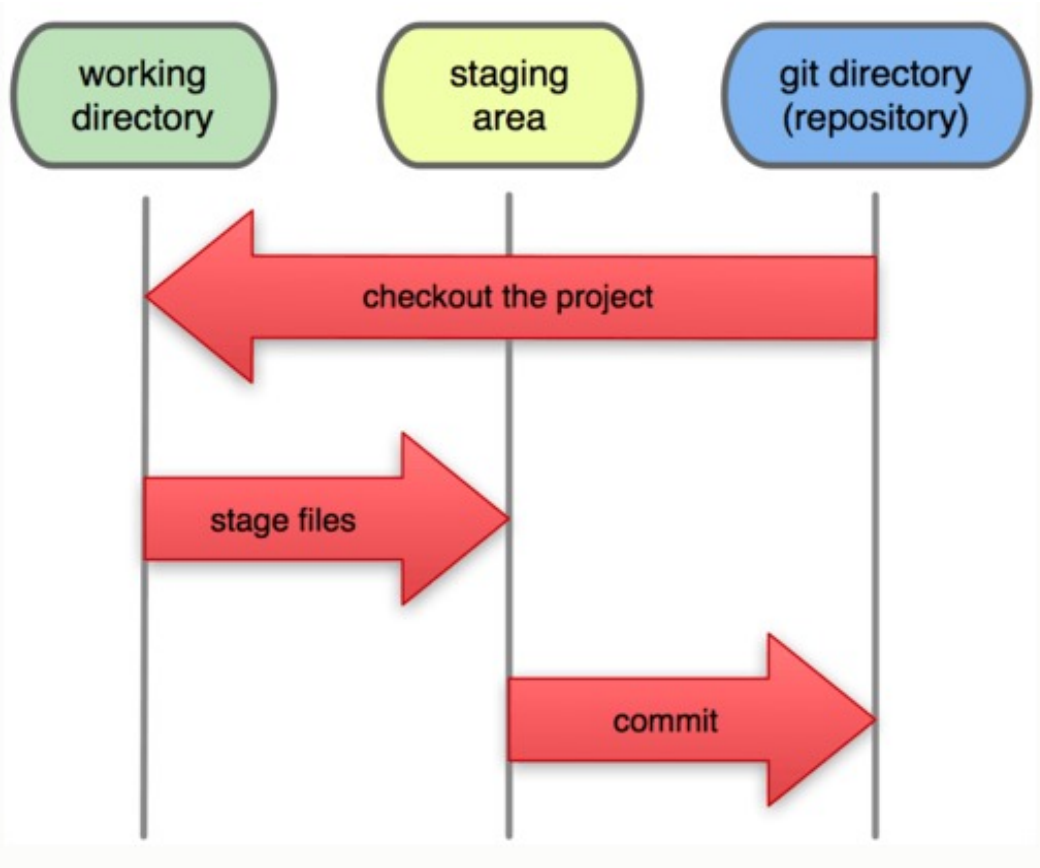
· 데이터가 로컬 데이터베이스에 안전하게 저장 됨

Modifed

· 수정한 파일을 아직 로컬 데이터베이스에
커밋하지 않음

Staged

· 수정한 파일을 곧 커밋할 것이라고 표시



세 가지 상태 [Look]

NHN Academy

Git으로 하는 기본적인 일

- 워킹 디렉토리에서 파일을 수정
- Staging Area에 파일을 Stage해서 커밋할 스냅샷을 만듦
- Staging Area에 있는 파일들을 커밋해서 Git 디렉토리에 영구적인 스냅샷으로 저장

NHN Academy

Git기초 사용법

파일 수정과 상태 변경 [Look]

NHN Academy

·워킹 디렉토리의 파일은 Tracked와 Untracked로 나뉨

· Tracked

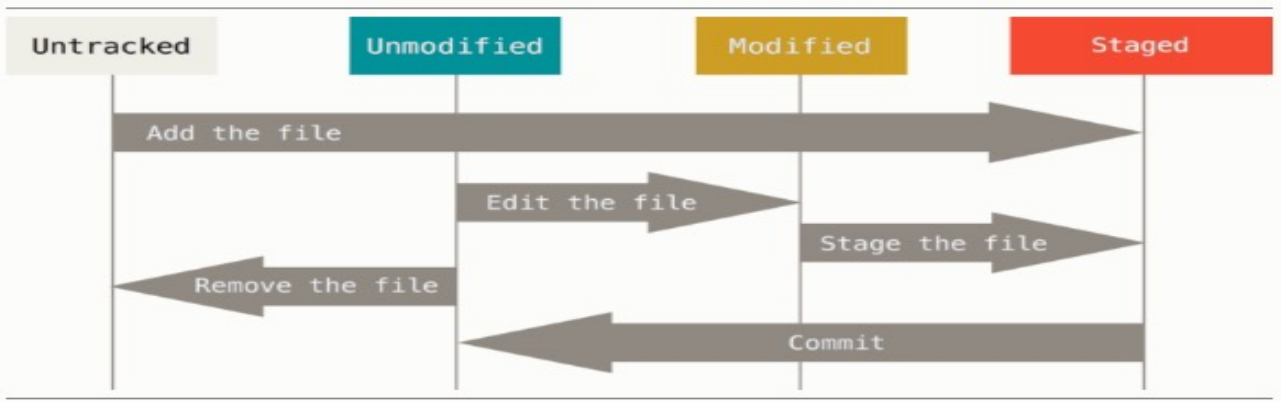
  · 이미 스냅샷에 포함

  · Unmodified

  · Modified

  · Staged

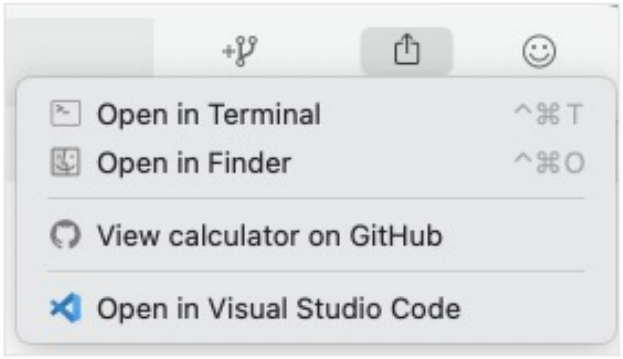
· Untracked



파일 수정과 상태 변경 [Look]

파일 수정

· Open in 기능을 이용하여 저장소가 위치한 곳으로 쉽게 이동 가능합니다.



· README.md 파일을 수정해 봅시다

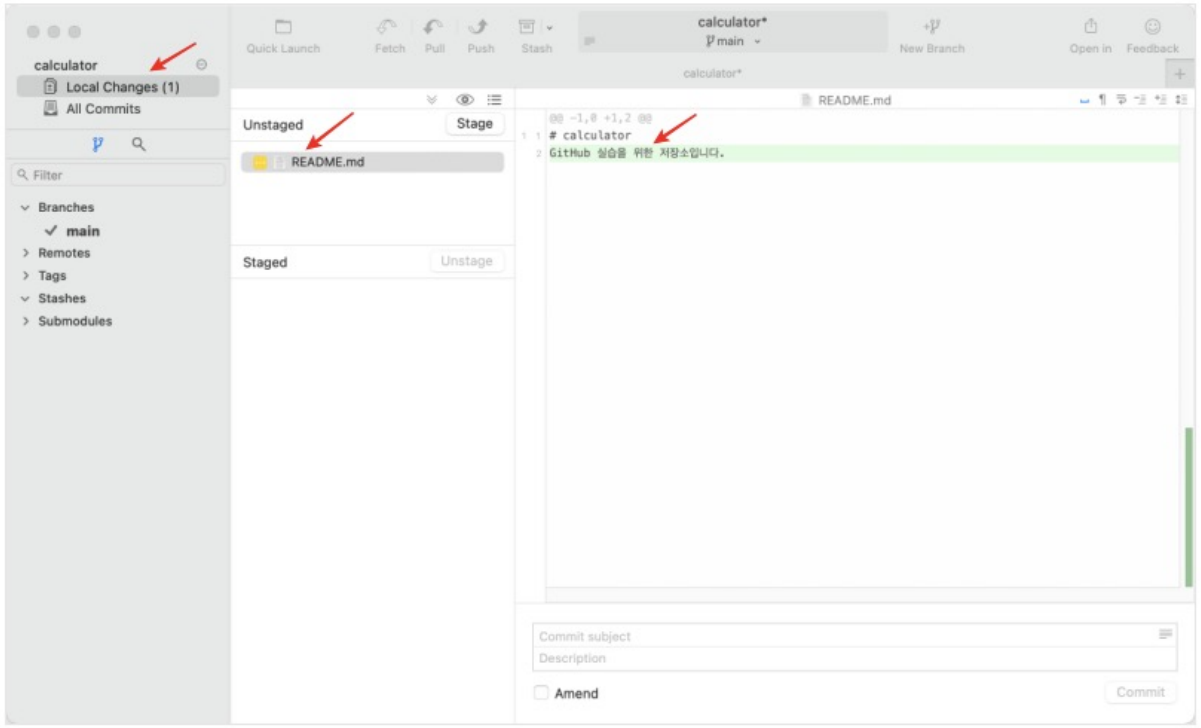
· 파일 내용을 변경하였습니다.

```
1  # calculator
2  GitHub 실습을 위한 저장소입니다.
```

파일 수정과 상태 변경 [Look]

NHN Academy

· Fork의 Local Changes 항목을 보면 README.md 파일이 Unstaged에 추가되었고 오른쪽에서는 Diff를 확인 할 수 있습니다.

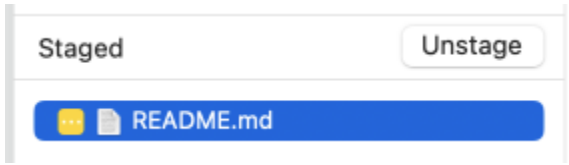


파일 수정과 상태 변경 [Look]

Stage 하기

· Unstaged에 있는 README.md 파일을 선택한 후 Stage를 클릭하면 Staged로 추가됩니다.

· 이제 커밋을 위한 준비가 완료되었습니다.



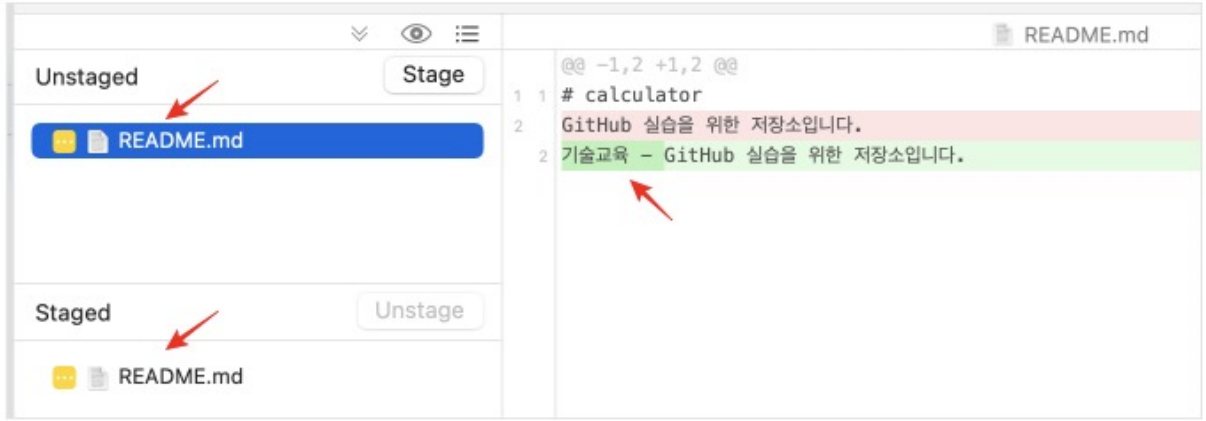
파일 수정과 상태 변경 [Look]

Stage 파일의 수정

· 그런데 아차! '기술교육'을 잊었네요. 파일을 다시 수정하였습니다.

```
1  # calculator
2  기술교육 - GitHub 실습을 위한 저장소입니다.
```

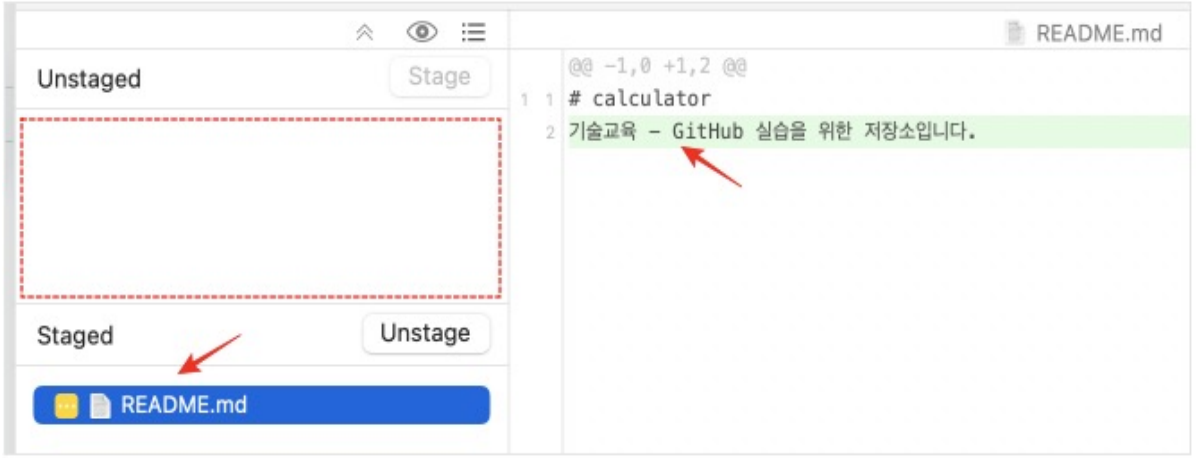
· README.md가 Unstaged/Staged 양쪽에 모두 있습니다.



파일 수정과 상태 변경 [Look]

NHN Academy

· Staged 파일을 수정하면 Staged 상태에서 수정된 Unstated 상태가 됩니다.
· 커밋을 위해서는 새로운 변경사항을 다시 Staged 상태로 변경하여야 합니다.



파일 수정과 상태 변경 [D.I.Y]

NHN Academy

README.md 파일을 수정하여 커밋 준비를 하세요.

- 한번 수정한 후 Unstage 상태가 되는 것을 확인하세요.
- Stage 상태로 만드세요.
- 다시 수정하여 Stage/Unstage 상태의 차이를 확인하세요.
- 최종적으로 Stage 상태로 만들어 커밋 준비를 하세요.

브랜치 [Look]

NHN Academy

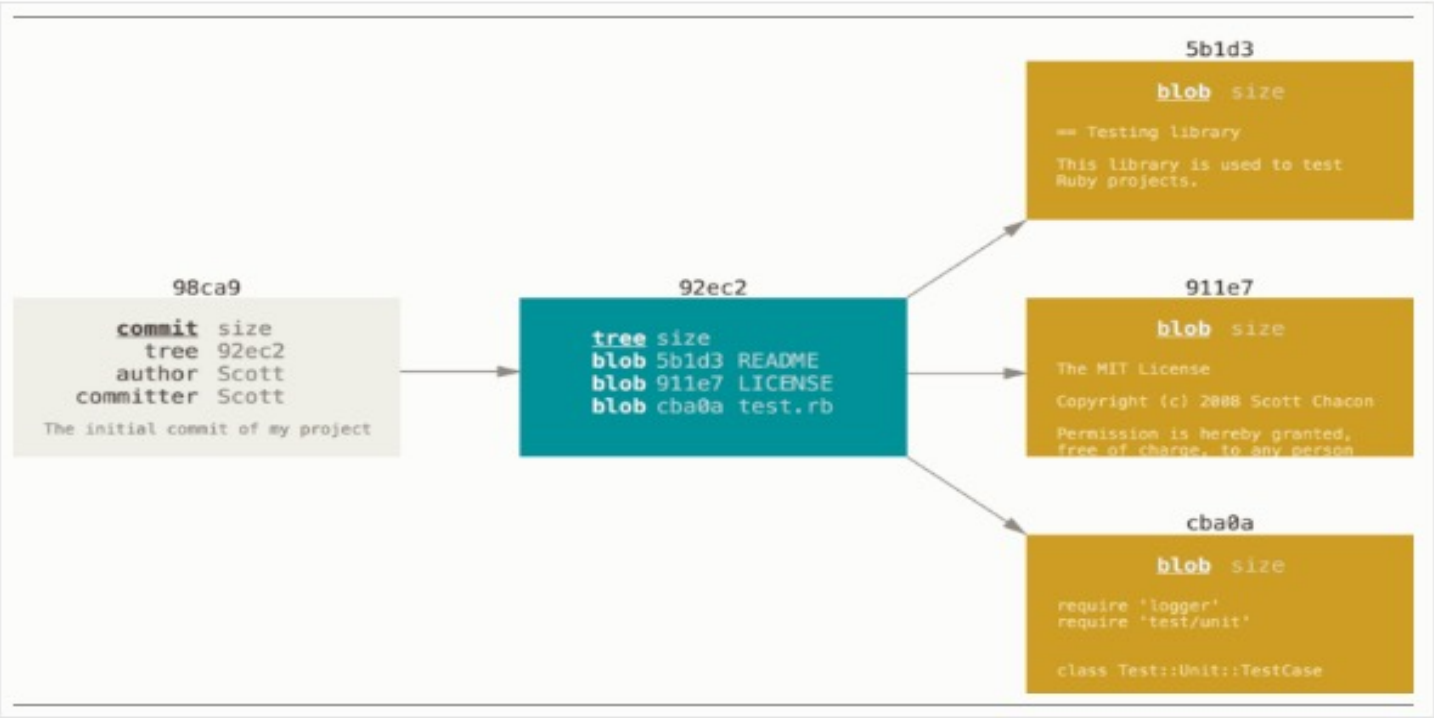
- 브랜치를 왜 만들고 사용해야 하는지 알아 봅시다.
- 실제 개발 과정에서 겪을 만한 예제
    1. 작업 중인 웹사이트가 있다.
    2. 새로운 이슈를 처리할 새 브랜치를 하나 생성한다.
    3. 새로 만든 브랜치에서 작업을 진행한다.
- 중요한 문제가 생겨서 해결하는 hotfix를 만들어야 한다면
    1. 새로운 이슈를 처리하기 전의 운영 브랜치로 이동한다.
    2. hotfix 브랜치를 새로 생성한다.
    3. 수정한 hotfix 테스트를 마치고 운영 브랜치로 머지한다.
    4. 다시 작업하던 브랜치로 옮겨가서 하던 일을 진행한다.
- 브랜치란 가상의 작업 공간

Git이 데이터를 저장하는 방법 [Look]

NHN Academy

· Git이 브랜치를 어떻게 다루는지 알려면 Git이 데이터를 어떻게 저장하는지 살펴봐야 합니다.

· 커밋을 하면 Git은 커밋 개체를 생성

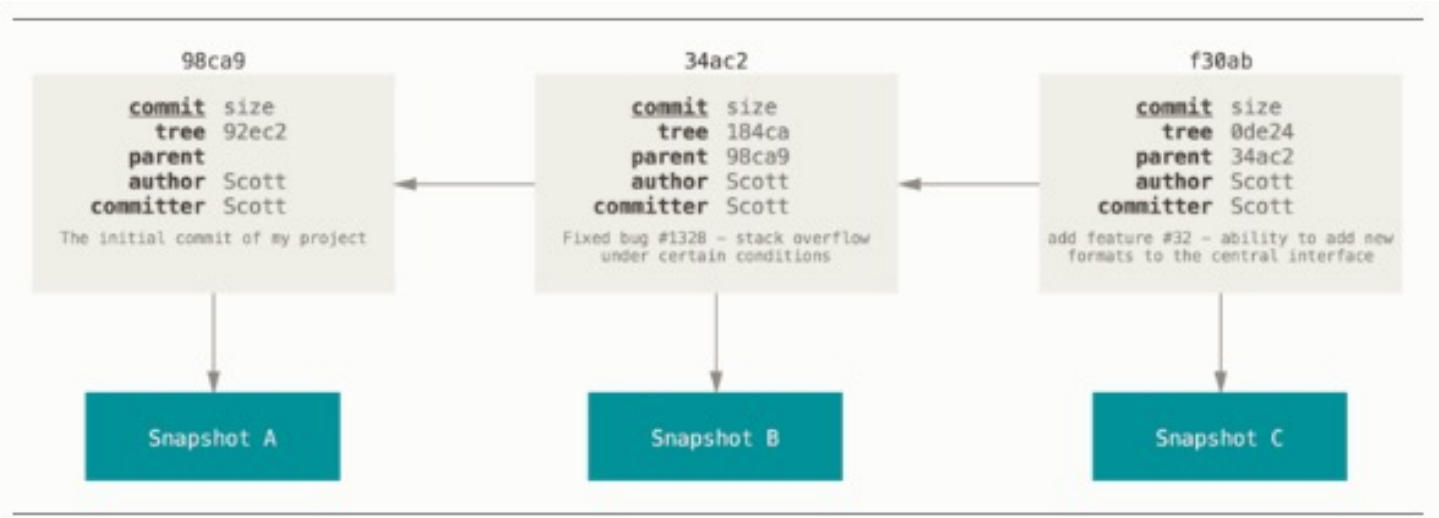


- 데이터의 스냇샷에 대한 포인터
- 메타 데이터: 저자, 커밋 메시지 등
- 이전 커밋에 대한 포인터

Git이 데이터를 저장하는 방법 [Look]

NHN Academy

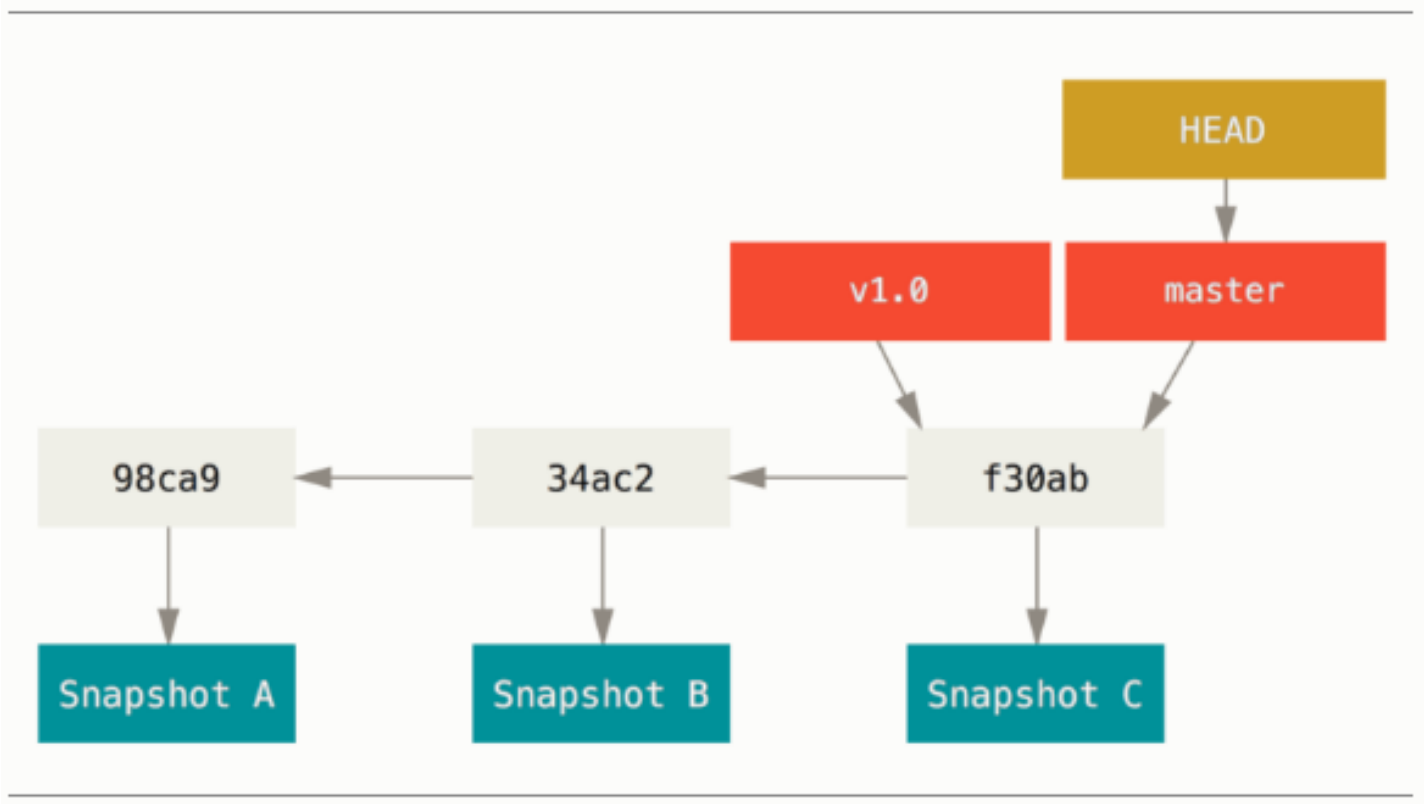
· 다시 파일을 수정하고 커밋하면 이전 커밋이 무엇인지도 저장



Git이 데이터를 저장하는 방법 [Look]

NHN Academy

· Git의 브랜치는 커밋을 가리키는 포인터

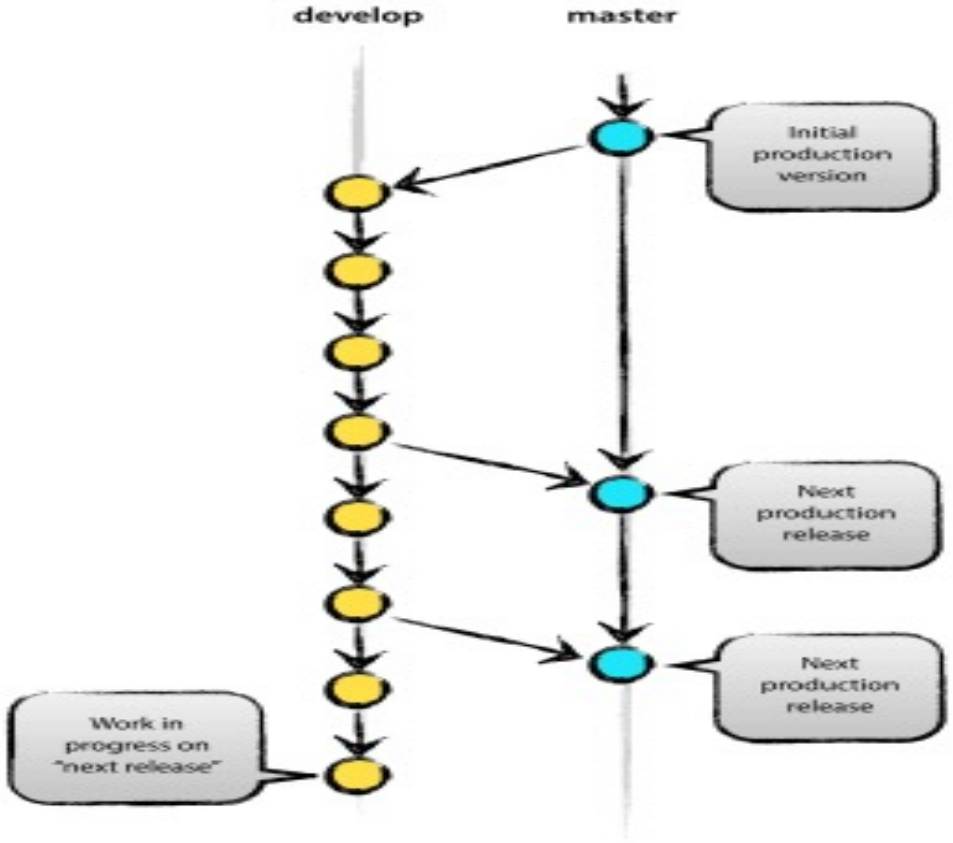


Git-flow [Look]

· 브랜치 관리 모델 중 하나로 Vincent Driessen이 주장

메인 브랜치

· 아래 두 브랜치는 항상 존재하는 메인 브랜치입니다.
  · master(main)
    · master(main) 브랜치는 배포된 소스가 있습니다.
  · develop
    · 다음 배포를 위한 소스가 있습니다.
    · 개발이 완료되면 일련의 과정을 통해 master(main)로 머지합니다.



Git-flow [Look]

NHN Academy

서포팅 브랜치

· master과 develop 외에 팀 멤버들이 병렬로 일할 수 있도록 도와주는 브랜치가 있습니다.

· 메인 브랜치와는 다르게 필요할 때 생성하였다가 삭제합니다.

1. feature 브랜치
2. release 브랜치
3. hotfix 브랜치

Git-flow [Look]

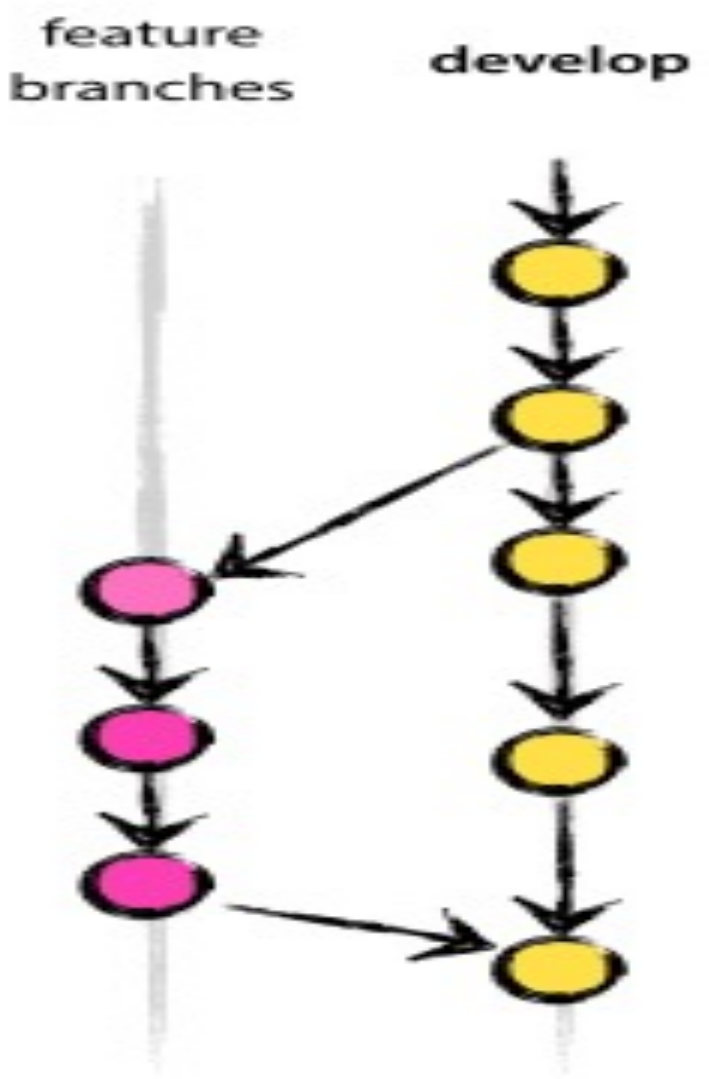
1. feature 브랜치

· 브랜치 생성: develop으로 부터

· 머지: develop으로

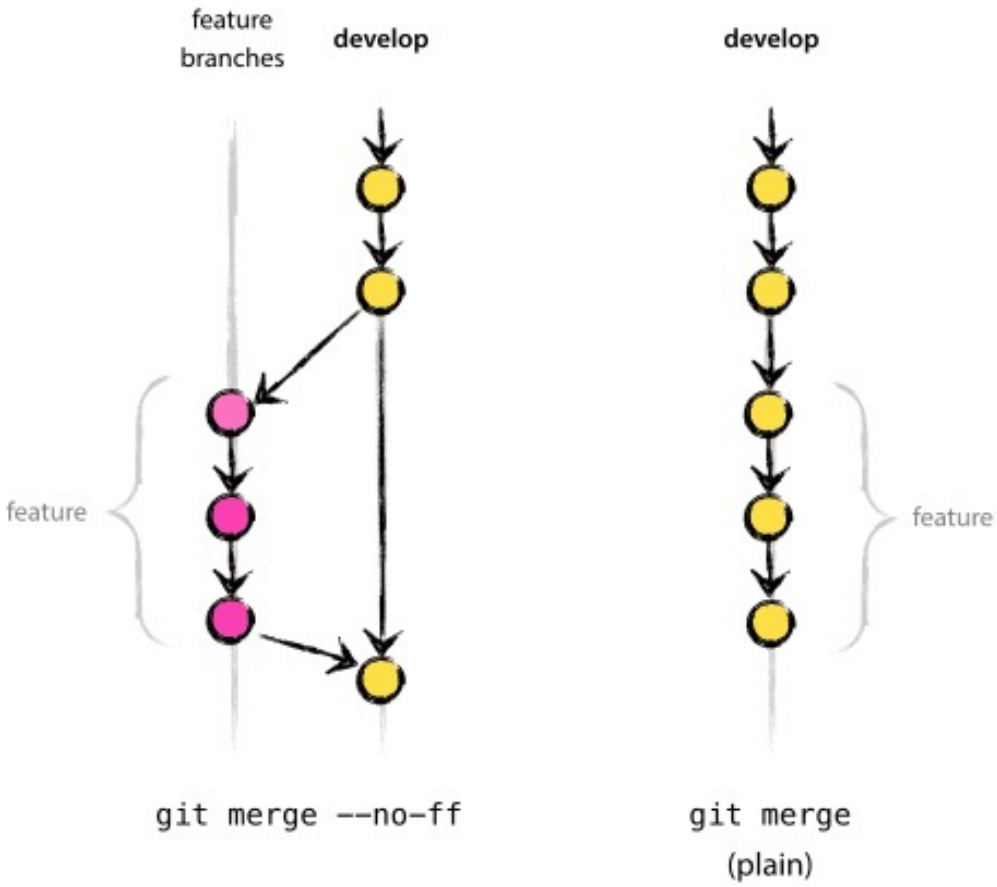
· prefix: feature/

· feature 브랜치는 특정 기능 하나에 대하여 develop으로 부터 생성하여 개발하며 개발이 완료되면 다시 develop 브랜치로 머지합니다.



Git-flow [Look]

· 이때 각 기능 별로 개발한 커밋을 구별할 수 있도록 fast-forward를 사용하지 않습니다.



Git-flow [Look]

2. release 브랜치

· 브랜치 생성: develop으로 부터

· 머지: develop과 master로

· prefix: release/

· release 브랜치는 배포를 위한 준비를 할 수 있는 브랜치입니다.

· release 브랜치는 develop 브랜치에 다음 배포를 위한
기능의 개발이 모두 완료되어 머지된 후 develop으로부터 생성합니다.

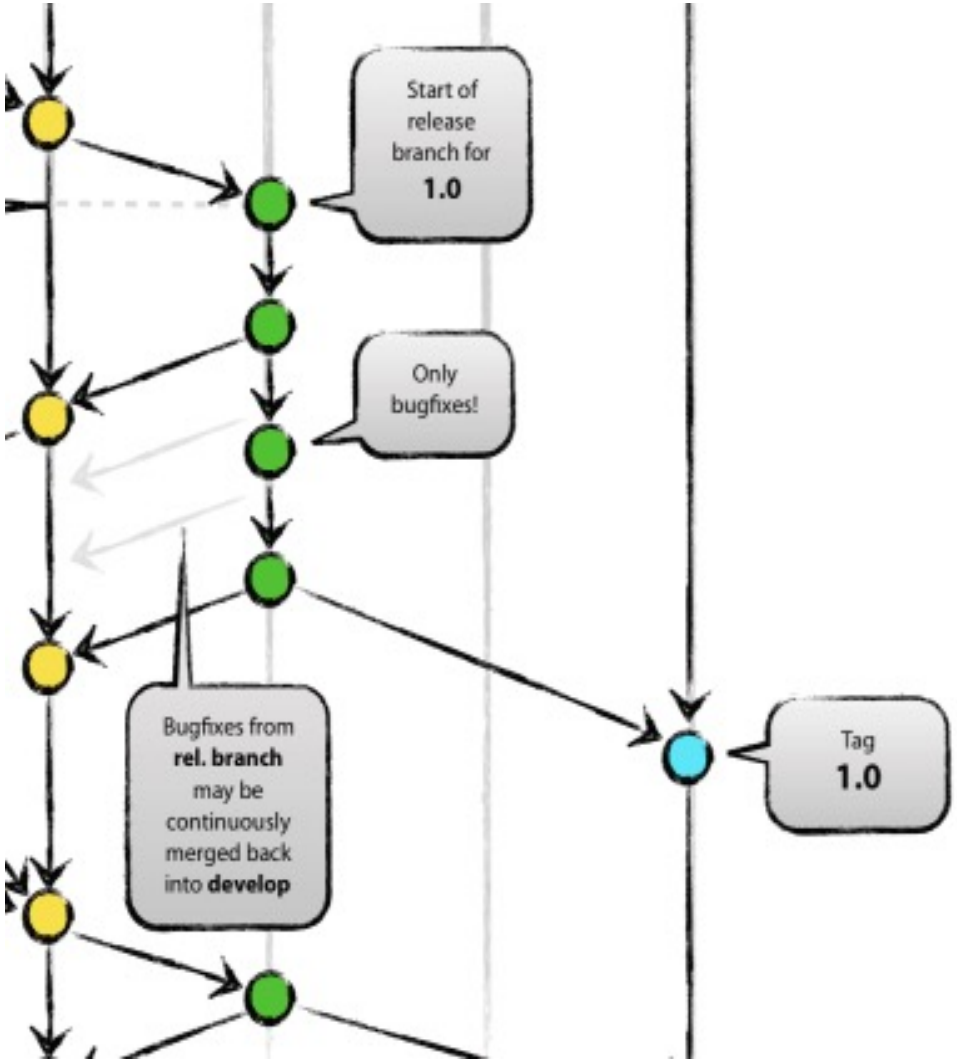
· release 브랜치에서는 각종 메타 데이터 (버전 명 등)을 변경하거나
작은 버그를 수정합니다.

· 배포 준비가 완료되면 release 브랜치를 master와 develop에 각각 머지합니다.

· master에는 버전 태그를 붙입니다.

· release 브랜치를 따로 가져가기 때문에 develop 브랜치에서는 바로 다음
배포를 위한 기능 기능을 시작할 수 있습니다.

· 그리고 release 브랜치를 다시 develop으로 머지하기 때문에 release 브랜치의
변경 사항이 develop에 반영됩니다.



Git-flow [Look]

3. hotfix 브랜치

· 브랜치 생성: master로 부터

· 머지: develop과 master로

· prefix: hotfix/

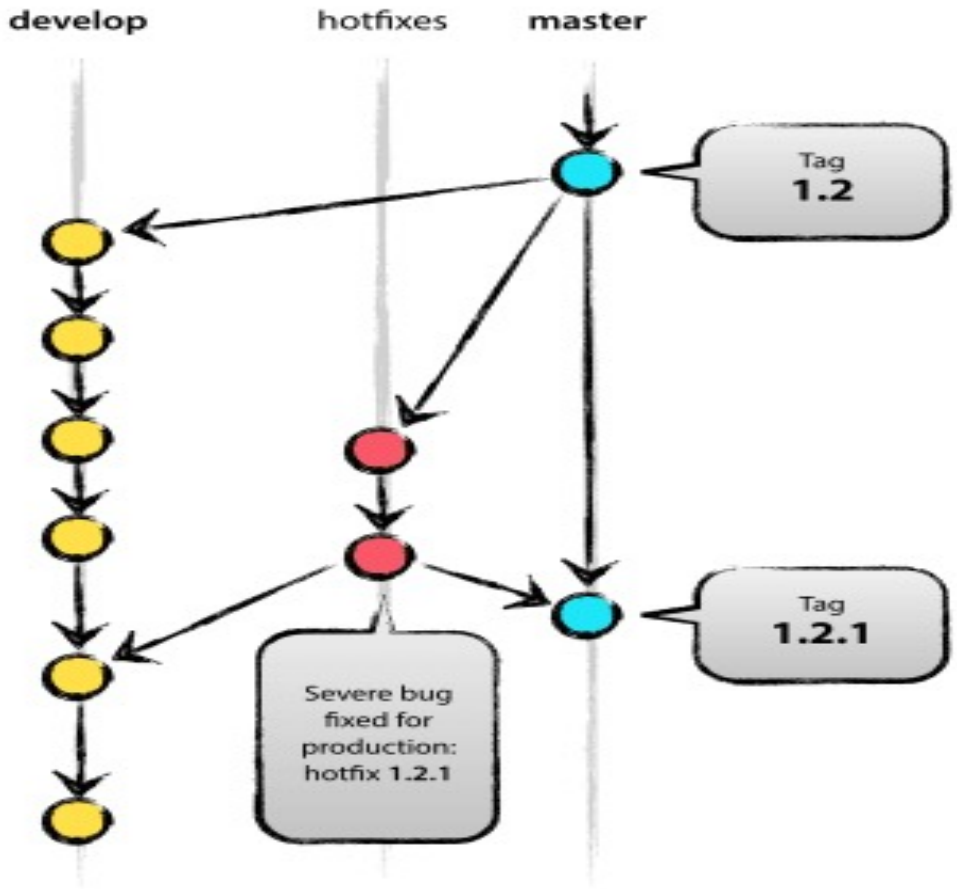
· hotfix 브랜치는 배포된 버전에 긴급한 변경 사항이 있을 때 활용합니다.

· hotfix 브랜치는 master로부터 생성합니다.

· 긴급한 이슈가 해결되면 다시 master와 develop에 머지 합니다.

· hotfix 브랜치 역시 따로 가져가기 때문에 develop 브랜치에서는 다음 배포를 위한 기능 개발을 계속 진행할 수 있습니다.

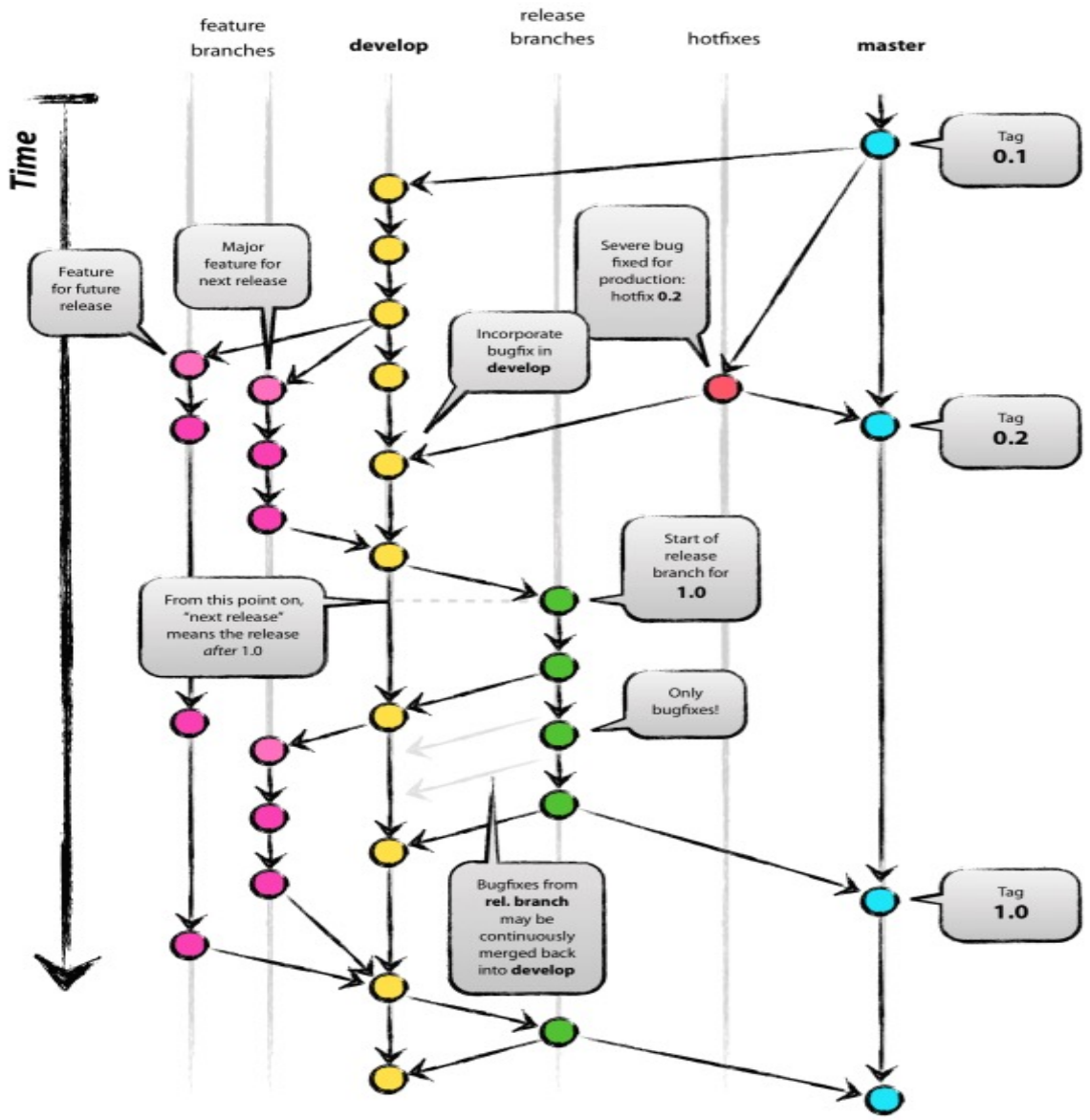
· hotfix 브랜치를 develop으로 머지하기 때문에 hotfix 브랜치의 변경 사항이 develop에 반영됩니다.



Git-flow [Look]

NHN Academy

요약



원격 저장소 (remote) [Look]

NHN Academy

- Git은 로컬 저장소만 사용할 수도 있지만 원격 저장소를 추가하여 사용할 수 있습니다.
- 우리가 GitHub의 저장소를 복제했을 때 git은 자동으로 GitHub에 있는 원격 저장소를 origin이라는 이름으로 추가하였습니다.

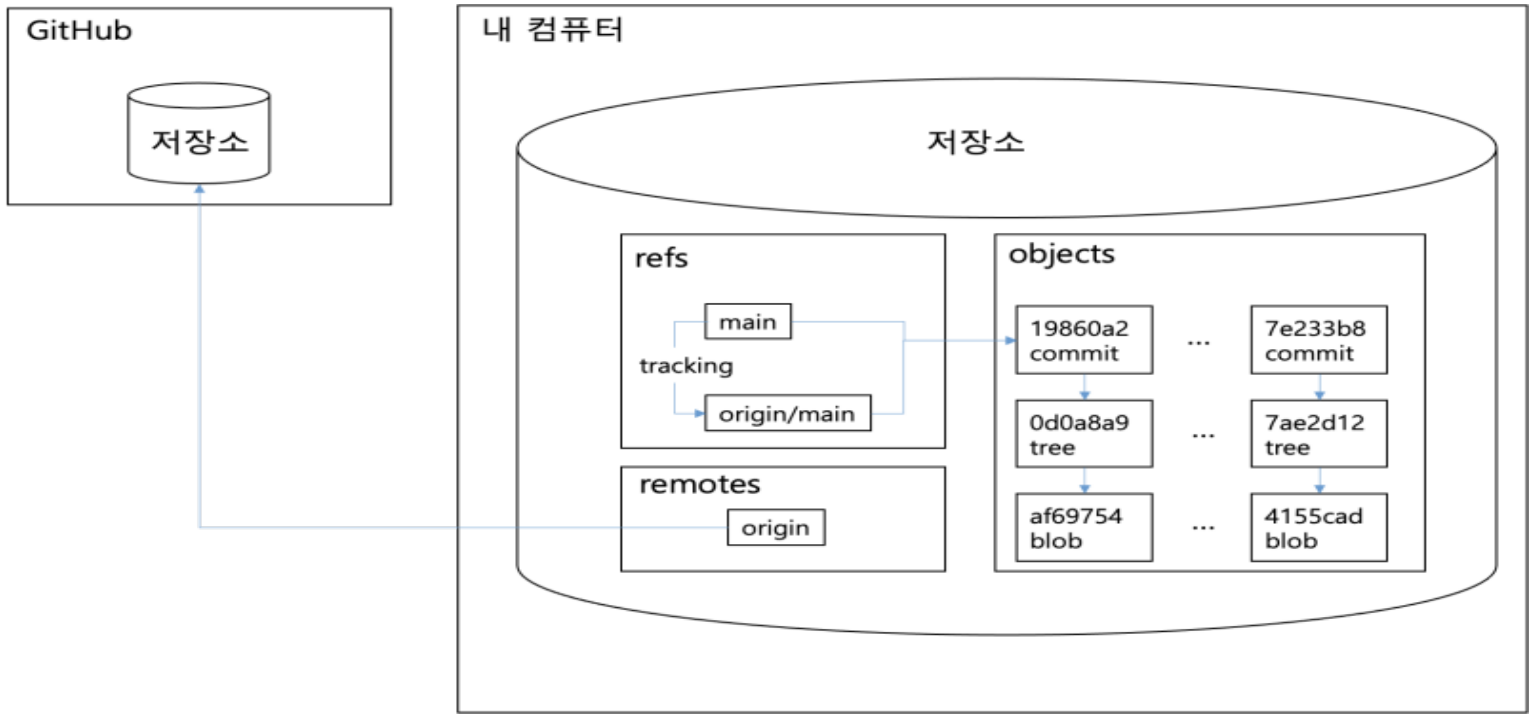
원격 저장소 (remote) [Look]

NHN Academy

로컬 브랜치와 원격 브랜치

· 로컬 브랜치인 main와 원격 브랜치인 origin/main은 서로 별개의 브랜치입니다.

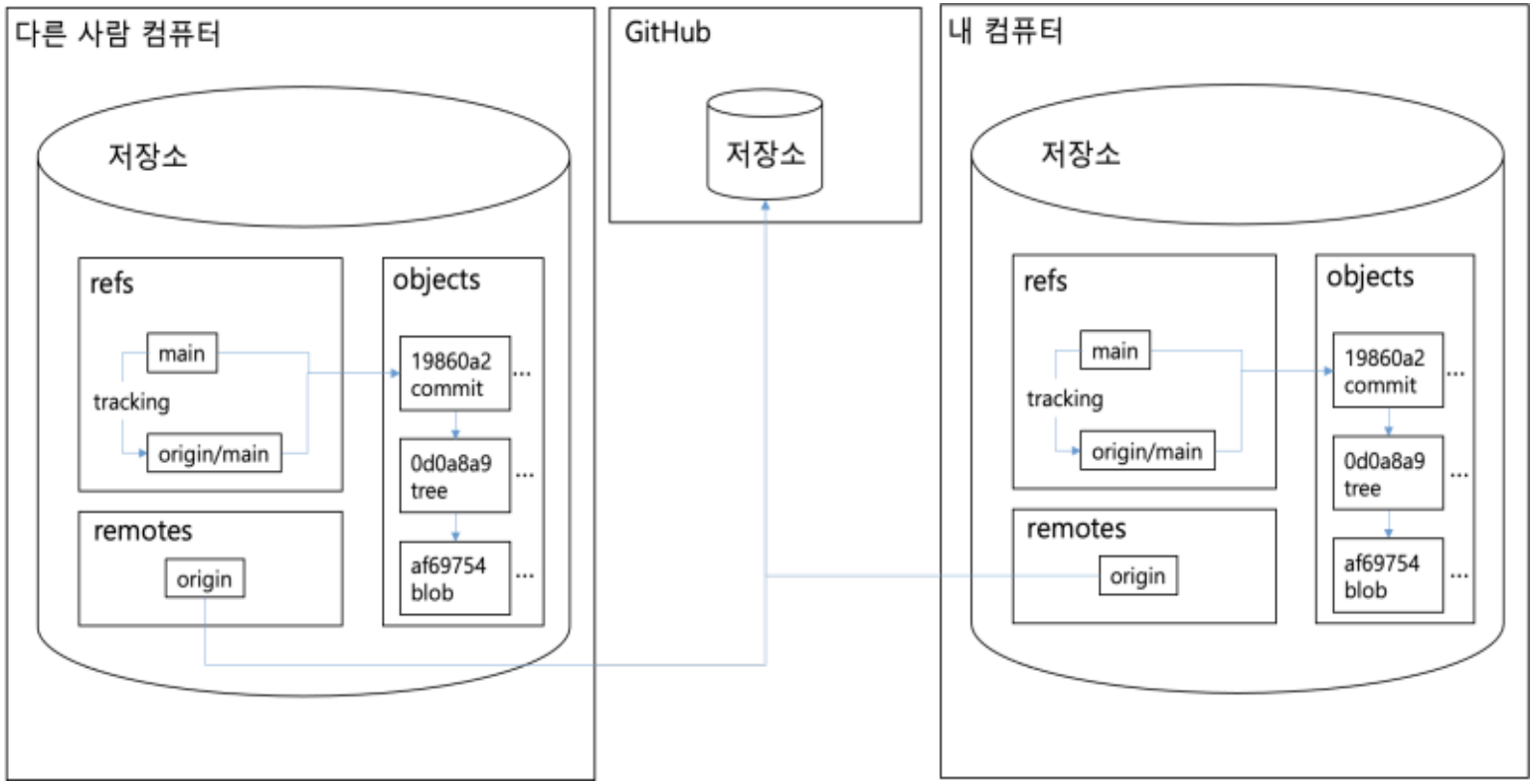
· Git이 origin/main를 체크아웃하여 로컬 브랜치인 main을 생성하였습니다.



원격 저장소 (remote) [Look]

NHN Academy

· A라는 사람의 main과 B라는 사람의 main 또한 별개의 브랜치입니다.



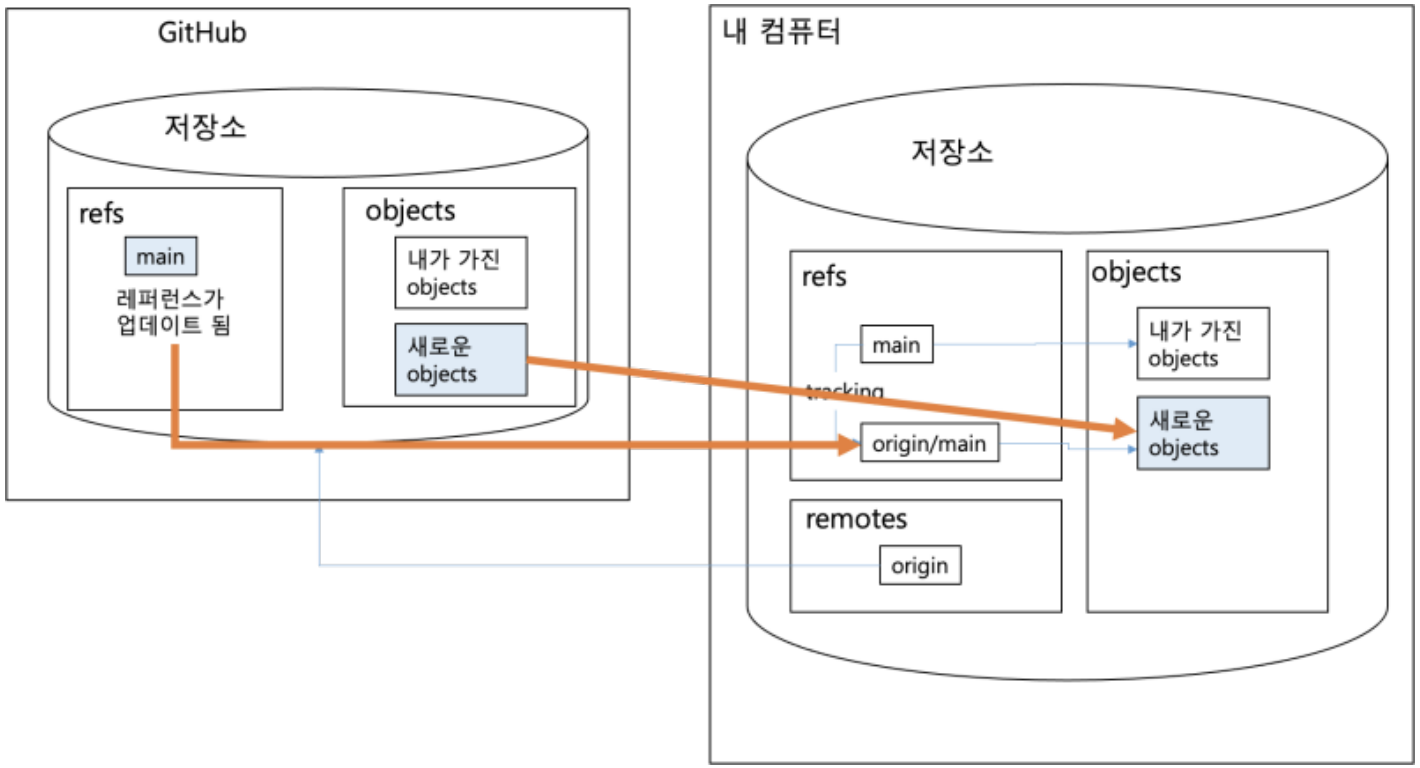
이 개념을 이해하면 git에서 헷갈리던 많은 것이 해결됩니다

원격 저장소 (remote) [Look]

NHN Academy

Fetch

· 원격 저장소의 변경 사항을 내려 받는 것을 fetch라고 합니다.
· Fetch까지만 하면 변경 사항을 받았을 뿐 아직 로컬 브랜치에는 머지(반영)되지 않은 상태입니다.
· 로컬 브랜치에 원격 저장소의 변경 사항을 반영하고 싶다면 수동으로 원격 브랜치를 로컬 브랜치로 머지해야 합니다.

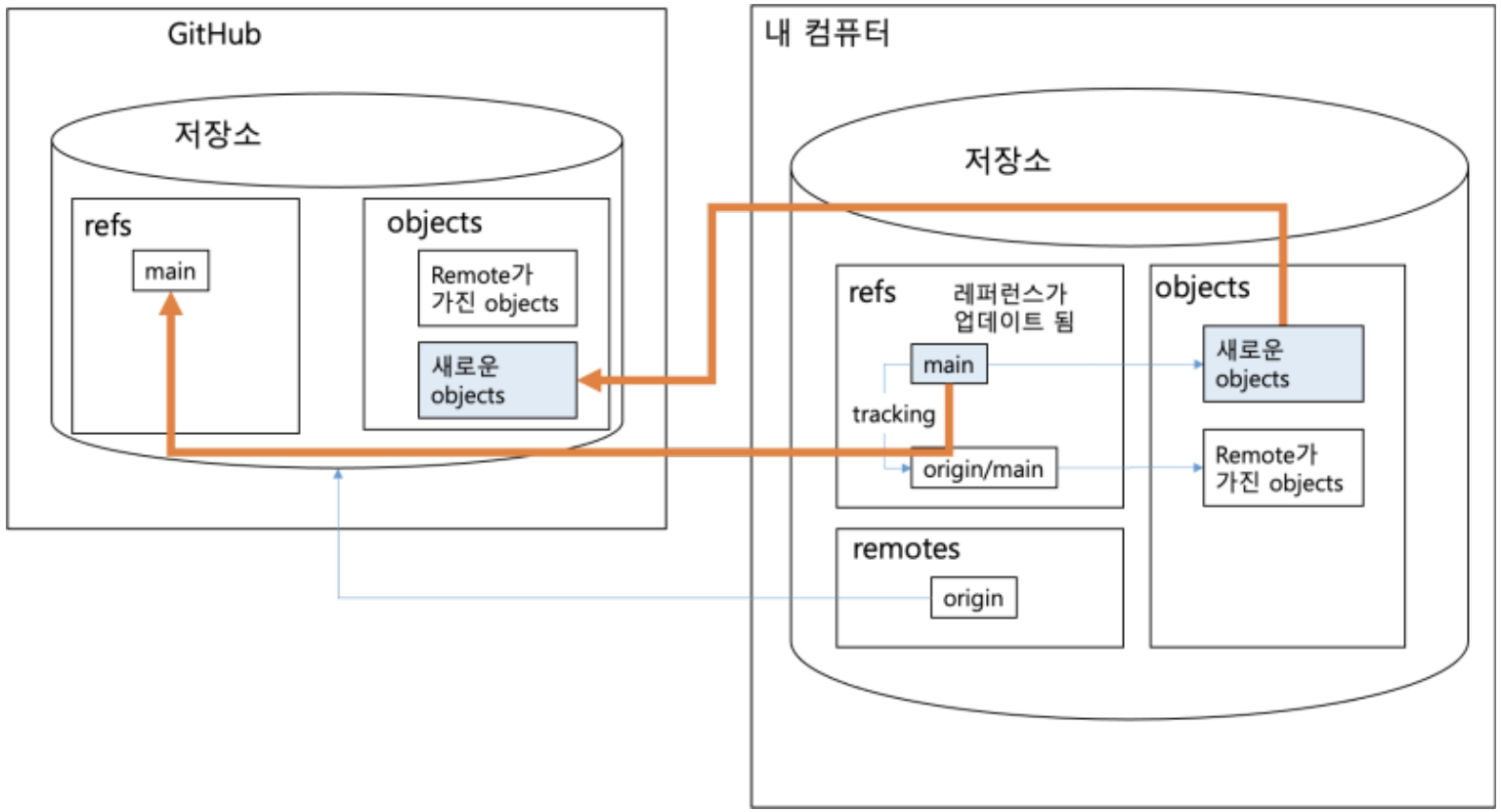


원격 저장소 (remote) [Look]

NHN Academy

Push

· 로컬 저장소의 변경 사항을 원격 저장소로 올리는 것을 push라고 합니다.

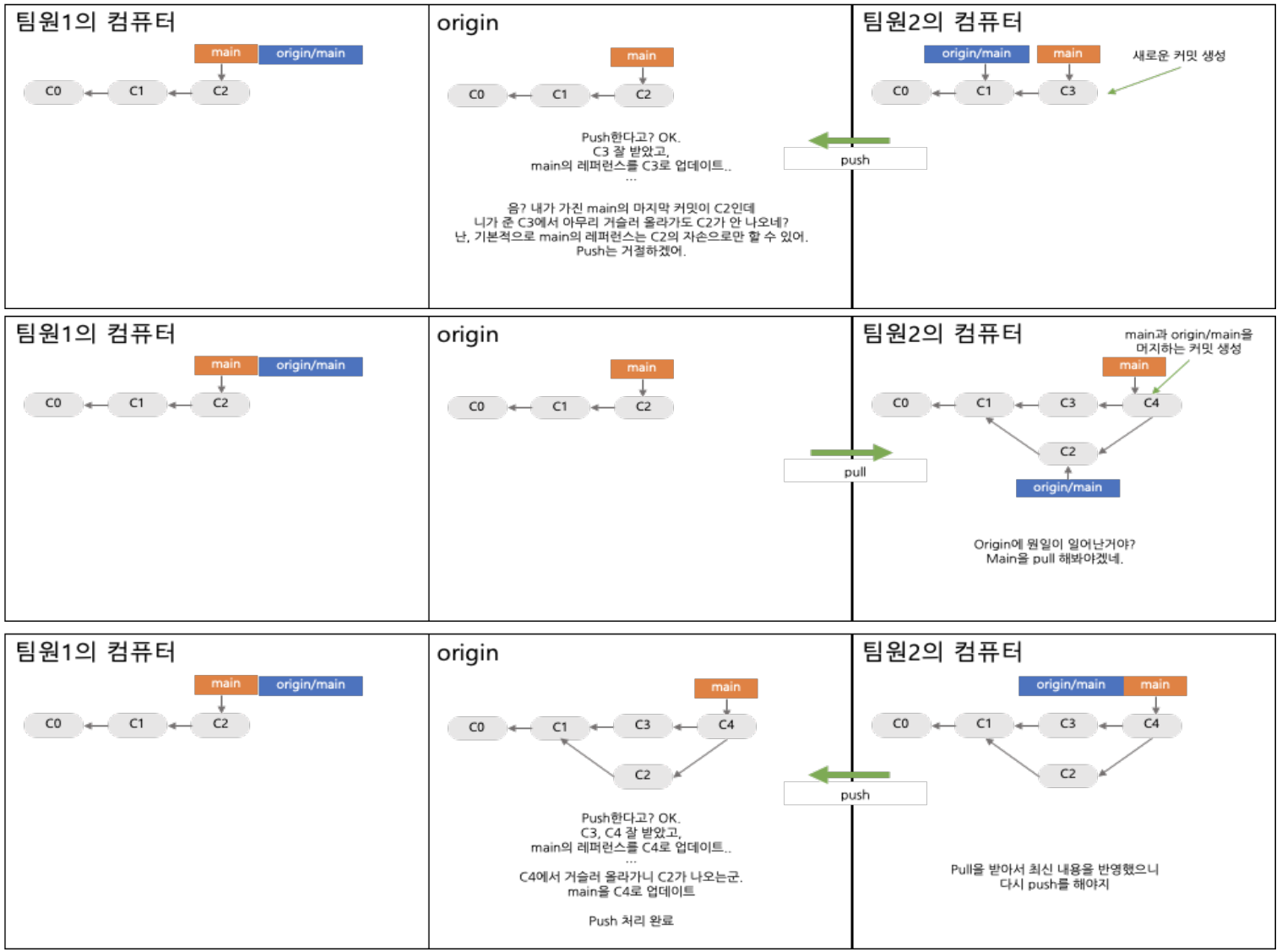


여러 사람이 한 브랜치에서 작업할 때 생기는 일 [Look]

NHN Academy

여러 사람이 한 브랜치에서 작업할 때 생기는 일 [Look]

NHN Academy



NHN Academy

Q&A

NHN Academy

감사합니다