Github

Commax 인턴

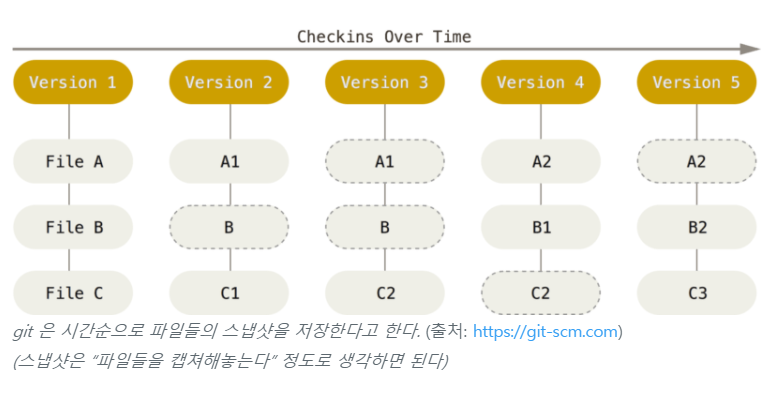
오 수찬

1. Github란?

Linux커널의 버전관리를 위해 사용되던 BitKeeper라는 상용 DVSC(Distributed Version Control System)가 무료 사용을 제한하면서 Linux개발 커뮤니티에서 자체 VCS(Version Control System)를 만들자는 목표아래 개발된 Version control System. Github는 빠른 속도, 단순한 구조, 비 선형적인 개발, 완벽한 분산, 빠른 속도와 작은 데이터 크기라는 5가지 기조아래 개발되고 발전되어왔다.

1. Github의 동작원리

Github 프로젝트에 담겨있는 데이터들은 파일 시스템 상의 \*‘스냅샷’이라고 볼 수 있다. Github는 기본적으로 이런 스냅샷의 연속으로 여러 버전의 파일을 모두 가지고 있다고 말한다. 하지만 실제로 모든 버전의 파일을 가지고 있는 것은 용량면에서 몹시 비효율적이다. 그렇기 때문에 Github에서는 프로젝트에서 바뀐 부분만 인지하여 변경이 있는 파일만 저장소에 저장되고 나머지 바뀌지 않은 부분은 링크만 건 채로 새로운 버전의 스냅샷을 만들어 저장하는 방식을 사용한다. 덕분에 적은 저장소 공간으로 모든 버전 별 히스토리 관리가 가능하다. 도식화해 보면 아래와 그림과 같다.



빗금 친 부분이 전 버전과 변경이 없어 링크만 연결된 부분이고 넘버링이 바뀐 부분이 버전업에 따라서 실제 변경이 있었던 부분이다. Github가 사용하는 링크 방식 대신 물리적으로 모든 버전을 저장한다면 압축해서 저장한다고 해도 A 1개, B 3개, C 1개, A1 2개 ….. 총 15개의 파일이 그중 5개의 파일은 중복 저장되어 낭비가 일어난다. Github가 사용하는 방식으로 저장 후 저장소를 열어보면 기본 A, B, C가 각 1개씩 A1, A2, B1, B2, C1, C2,

* 스냅샷: 특정 시간에 저장장치의 상태. 쉽게 말해서 스토리지의 백업 복사본.

C3가 각 1개씩 저장되어 총 10개의 파일이 저장되어 있을 것이다.

Github가 사용하는 스냅샷 방식은 기존 VSC들과의 차이점인데. 기존의 VSC는 파일의 목록과 파일 별로 변경 점 및 변경 히스토리를 시간순으로 관리했다. 이러한 방식에서 (표에서)Version 5를 복제해 오려고 하면 파일별로 히스토리를 따라가 Version 5때의 각 파일의 상태를 추적 후 초기 파일에 변화를 단계적으로 적용해서 Version 5때의 상태로 변경한 뒤 받아오는 방식을 사용한다. 반면 스냅샷 방식을 사용하면 Version 5때의 스냅샷을 이용해서 필요한 파일을 받아오면 된다.

|  |  |  |
| --- | --- | --- |
| **버전 관리 방식** | **기존 VSC(SVN 포함)** | **Github 스냅샷** |
| **속도** | 느리다 | 빠르다. |
| **저장 공간** | 작다 | 크다 |

위 표는 기존 VSC의 버전관리 방식과 Github의 버전관리 방식을 속도와 저장 공간 측면에서 비교한 것이다.

기존 VSC의 경우 히스토리를 따라가며 변경을 적용해야 하기 때문에 상대적으로 느리다. 반면 Github의 경우 스냅샷에 저장된 링크를 따라가거나 그 파일을 그냥 전송하면 되기 때문에 상대적으로 빠르다.

모든 파일이 저장되는 것이 아니라 변경에 관한 내용만 저장되기 때문에 용량은 절약된다.

이제 Github의 내부 저장 흐름을 알아보자. SVN같은 CVCS와 달리 Github는 원격 저장소 이외에 로컬 저장소도 존재한다. 이 로컬 저장소는 또 3가지 영역으로 나뉘어진다. 각 영역은 Github가 보는 파일의 3가지 상태에 각각 매칭된다.

1. Working Directory – Modified 상태

Working Directory는 그 말 그대로 우리가 코드를 수정하고 파일을 추가하는 등의 작업을 하는 공간이다. (ex. 이클립스의 Workspace)

Modified상태란 기존상태에서 변경을 감지하는 것 이외의 아무 처리도 하지 않은 상태를 말한다.

1. Staging Area – Staged 상태

Index라고도 불린다. 로컬 저장소에 반영하기 전 준비 영역이다. 변경이 일어날 파일의 목록을 저장한 스냅샷 이라고 보면 된다.

+ Staging Area는 Directory가 아닌 파일의 형태로 존재한다. (.git/index)

Staged상태는 로컬 저장소에 저장하기 전 준비상태로 보면 된다.

1. Local Repository – Committed 상태

Local Repository는 Github가 사용하는 로컬 저장소를 의미한다. 로컬 저장소 안에는 프로젝트의 메타데이터와 객체 데이터베이스 등 많은 요소들이 존재하지만 그중 중요한 3가지만 짚어보면 아래와 같다.

* Conifg

git Config로 설정한 설정 정보들이 저장되는 파일.

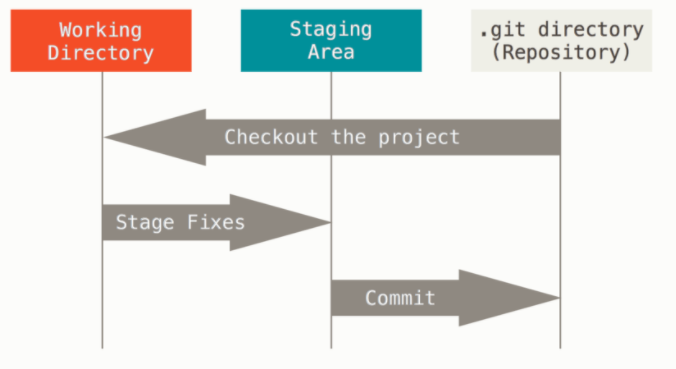
* HEAD

현재 선택한 branch를 가리키는 파일.

* Objects

Tree, Blob, Commit등의 정보를 관리하고 저장. (버전정보, 히스토리, 태그 등)

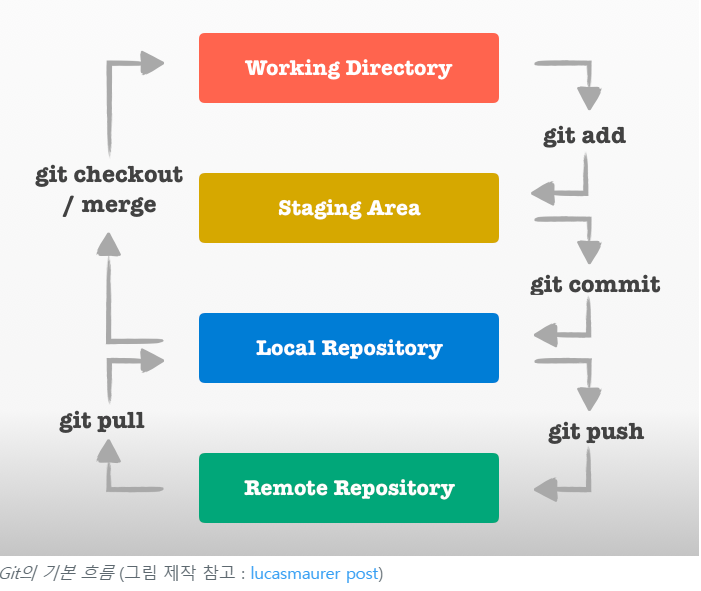
Committed상태는 수정한 파일이 저장되어 새 버전이 생성된 상태를 말한다. 달리 말하면 Github Directory에 영구적인 스냅샷으로 저장된 상태이다.



<https://git-scm.com> Git의 기초 문어에서 발췌

위의 그림이 3가지 상태와 3가지 영역과 그 흐름을 나타낸 그림이다.

반면 원격 저장소에는 단순 저장소만 존재한다. 이점이 다른 VCS와 큰 차이를 보인다 SVN이나 CVS같은 다른 VCS시스템의 경우 네트워크에 접속하지 않고는 할 수 있는 일이 거의 없다. 하지만 Github의 경우 거의 모든 명령을 로컬에서 실행한 뒤에 원격 저장소에 단지 Push할 뿐이다. 히스토리 조회뿐만 아니라 지난버전 파일까지 로컬에서 실행되고 로컬에 저장된다. 이는 단순히 네트워크가 필요 없을 뿐 아니라 CVCS에 비해서 속도면에서도 엄청난 차이를 보인다.



위의 그림이 앞서 설명한 내용을 총 정리한 흐름도이다. 각 과정에 필요한 명령어까지 함께 보여주고 있다.

1. Github의 원활한 사용을 위한 방법

설치부터 사용까지의 과정을 간단하게 정리함으로써 원활한 사용법을 보이고자 한다. 위의 큰 흐름을 이해한 이후라면 그 사이의 세부적인 과정을 살펴본다고 이해하면 되겠다.

Github는 GUI환경과 CLI환경을 모두 지원한다. 두가지 환경은 같은 기능을 제공하기 때문에 이 부분은 사용하기 편한 환경을 선택해 사용하면 된다.

설치 이후에는 환경 설정을 해야 한다. Github이용에 있어 가장 중요한부분 중 하나가 바로 환경설정 부분이 되겠다. 위에서 기술한 Config파일 같은 설정 파일들에 저장되는 내용으로 Git config라는 명령으로 설정내용을 확인하고 변경할 수 있다. 환경설정은 크게 3파일에 나뉘어 저장된다.

* /etc/gitconifg 파일

시스템의 모든 사용자 및 모든 저장소에 걸쳐 적용되는 설정.

Git config –system 옵션으로 읽거나 쓸 수 있다. (단 관리자 권한이 필요하다.)

* ~/.gitconifg, ~/.config/git/config 파일

특정 사용자에게 적용되는 설정. 단 그 사용자가 사용하는 모든 저장소에 걸쳐 적용된다.

Git config –global 옵션으로 이 파일을 읽고 쓸 수 있다.

* .git/config 파일

Github Directory에 있는 파일. 위에서 설명했던 설정파일이 이 파일이다. 특정 저장소 혹은 프로젝트에 적용되는 설정.

Git config –local 옵션으로 사용할 수 있다. (--local이 default이므로 생략 가능.)

단 로컬 저장소로 이동 후 사용해야한다.

각 설정들은 작은 범위의 설정이 더 우선시된다. 로컬 저장소의 .git/config파일이 가장 우선시되며 ~/.gitconfig, ~/.config/git/config 파일이 다음 우선순위를 가진다.

각각의 설정파일을 변경하기 전에 가장 먼저 해야 할 설정이 존재한다. 바로 사용자 정보를 등록하는 것이다. Git config –global user.name “” 명령을 사용함으로써 사용자의 이름과 Git config –global user.email [abc@d.com](mailto:abc@d.com) 명령으로 사용자 이메일을 등록할 수 있다.

--global옵션을 사용하면 동일 시스템 동일 유저로 접속 시 모든 프로젝트에 동일한 이름과 이메일 주소가 적용된다. 특정 프로젝트에서 다른 이름과 이메일을 사용하고 싶으면 --global옵션을 제거하면 된다.

이외에 편집기를 설정할 수 있다. 깃은 기본적으로 시스템 기본 편집기를 사용한다 기본 편집기를 변경하고 싶다면 git config –global core.editor (Editor name) 명령어를 사용하여 변경이 가능하다.

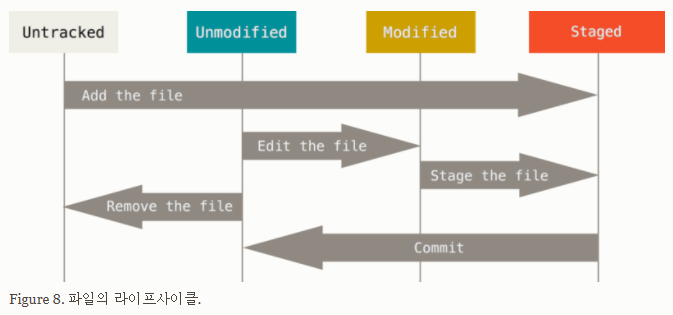
이외에 git config –list 명령어를 사용하면 변경이 가능한 설정의 목록을 볼 수 있다. 설정은 Key-Value쌍으로 관리되며 git config (Key)명령으로 특정 설정에 대한 값을 볼 수 있다.

환경 설정이 끝난 이후에는 저장소를 결정할 필요가 있다. 기존 파일시스템에 존재하던 한 Directory를 로컬 저장소로 변경해서 사용할 수도 있고 존재하던 깃 저장소를 내려 받아 사용할 수도 있다.

일단 기존 Directory를 로컬 저장소로 변경하는 과정을 살펴보자. 우선 git init 명령으로 .git이라는 하위 디렉토리를 만들어야 한다. 이후 git add (추가할 파일)명령으로 파일을 추가해 준 뒤에 git commit 명령으로 커밋해주면 로컬 저장소로 지정이 끝난다.

기존 저장소를 내려 받는 경우 git clone명령을 사용하면 된다. Git clone <https://github.com/(유저ID)/>(저장소 이름) (원하는 저장소 이름)의 명령어로 간단하게 저장소를 내려 받아 사용할 수 있다.

기본적인 사용에 있어 가장 중요한 단계이다. 작업공간에서 원하는 코드를 수정하고 저장소에 저장하는 과정이다. 이 과정을 파일의 상태와 연관 지어 살펴보자.



<https://git-scm.com/book> git의 기초 - 수정하고 저장소에 저장하기 문서에서 발췌

위에서 봤던 저장소의 구조와 파일의 상태와 유사하지만 조금 더 파일의 상태에 집중한 그림이다. 실제 동작과 명령에 기초에 위의 상태에 대해 살펴보자.

Git status 명령으로 저장소 안의 파일이 현재 어떤 상태인지 확인할 수 있다. 상태에 대해 자세히 살펴보자.

Untracked상태의 경우 commit시에 저장되지 않는 파일이다. 다음과 같은 경우 파일은 Untracked상태로 취급된다.

* 새로 만든 파일이 git add 되지 않은 경우.
* .gitignore 파일에 추가되어 Github가 의도적으로 무시하는 경우.
* Git rm 명령에 의해서 Tracked 상태의 파일이 삭제된 경우.

Untracked상태의 파일은 Commit시에 스냅샷에 추가되지 않는다. Git rm명령으로 파일을 삭제하더라도 스냅샷을 이용하여 예전 버전으로 복구 시에 복구가 가능하다.

Unmodified상태는 현재 추적 중이나 변화가 일어나지 않은 파일이다. 이러한 파일의 경우 추가적인 작업이 필요하지 않다.

Modified상태는 현재 추적중인 파일에 변화가 일어난 상태이다. Git status 명령으로 살펴보면 Modified상태의 파일은 Changes not staged for commit상태로 보여 진다. 쉽게 말해서 추적중인 파일에 변경이 일어났으나 git add 명령으로 staged 상태로 변경되지 않은 파일들이 이 상태에 속한다.

Staged상태는 Modified상태의 파일을 git add 명령으로 commit시 저장될 스냅샷에 변화를 적용할 준비를 마친 상태이다. 이 스냅샷은 Staging Area파일에 저장된다고 위에서 기술한 바 있다.

이렇게 파일을 수정하고 Staged상태로 만들었다면 git commit 명령으로 commit을 실행할 수 있다. 이렇게 스냅샷을 기록해 둘 수 있다.

Cat .gitignore 명령어로 Tracked상태의 파일을 Untracked상태로 변환시킬 수 있고 git rm 명령으로 아예 삭제할 수도 있다. 여러 옵션과 설정들이 존재하니 자세히 살펴보고 활용할 수 있도록 하자.

그렇다면 로컬 저장소의 내용을 원격 저장소에 저장하는 방법은 무엇일까?

Git remote add (별칭) (URL) 명령으로 로컬 저장소와 원격 저장소의 연결이 먼저 이뤄져야 한다. 만약 저장소를 Clone한 경우 원격 저장소는 자동으로 origin이라는 이름으로 추가된다.

이후 git fetch (원격 저장소 명) 명령이나 git pull 명령으로 원격 저장소의 데이터를 가져올 수 있다. 둘의 차이점은 fetch명령의 경우 데이터를 가져올 뿐 자동으로 merge해주지는 않는다. 반면 pull 명령의 경우 데이터를 가져온 뒤 자동으로 merge 해준다. 반대로 로컬 저장소의 내용을 원격 저장소로 보내고 싶을 때는 git push (원격 저장소 명) (Branch 이름)으로 간단하게 전송이 가능하다.

깃의 기본적인 활용에 있어 마지막 단계이다. Commit 히스토리를 조회하고 복구하는 과정이다.

Commit히스토리 조회는 간단하게 git log 명령을 통해 확인할 수 있다. 여기에는 Commit한 유저와 날짜 SHA-1 checksum이 함께 보여 진다.

-p 옵션을 사용하면 각 commit한 스냅샷들의 차이를 보여준다. 또 –stat옵션의 경우 어떤 파일이 수정되었는지를 확인할 수 있다.

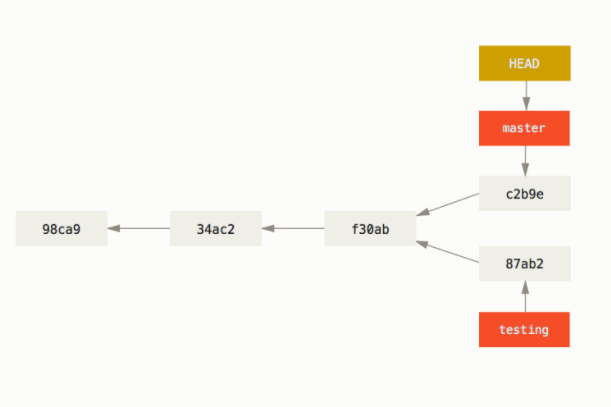
바로 이전 Commit에 에러가 있었을 시 Staging Area파일을 수정한 뒤 예를 들어 누락된 파일을 추가하는 등의 작업을 진행한 뒤 git commit –amend 명령을 사용해서 직전 commit되어 저장된 스냅샷을 수정할 수 있다. 이경우 1번의 commit으로 기록된다. 또 잘못해서 Staged상태로 변환해버린 파일은 git reset (파일명) 명령을 통해서 다시 Unstage상태로 복원할 수 있다.

마지막으로 수정된 파일을 어떻게 이전의 상태로 복원할 수 있을까? Git checkout 명령을 사용해서 변경이 가능하다. Git checkout – (파일 이름) 명령을 사용하면 특정 파일을 수정 전의 상태 즉 직전 commit에 저장된 스냅샷과 같은 상태로 복구할 수 있다.

그렇다면 예전 commit상태로 돌아가는 방법은 무엇일까? Git reset –hard (SHA-1 값) 명령을 통해 commit 로그에 존재하는 예전 상태로 돌아갈 수 있다.

복구 이전의 Commit을 저장해 두거나 현재 Commit을 저장해 두고 작업을 진행하고 싶은 경우 활용가능한 기능이 Branch이다. Branch는 쉽게 말해 특정 스냅샷을 가리키는 임시저장 팻말과 같은 역할을 한다. git branch (Branch 명)을 활용하면 새로운 Branch를 생성할 수 있고 git checkout (Branch 이름) 명령을 통해서 Branch를 바꿔가면서 작업할 수 있다.

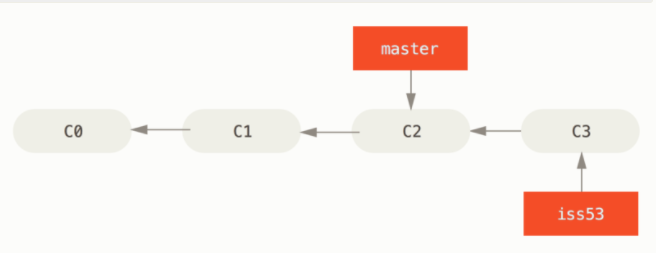
또 아래 그림과 같이 서로 다른 스냅샷이 저장되는 경우 완전히 다른 2개의 갈라지는 Branch가 생성될 수 있다.



<https://git-scm.com/book> 브랜치와 Merge의 기초 문서에서 발췌

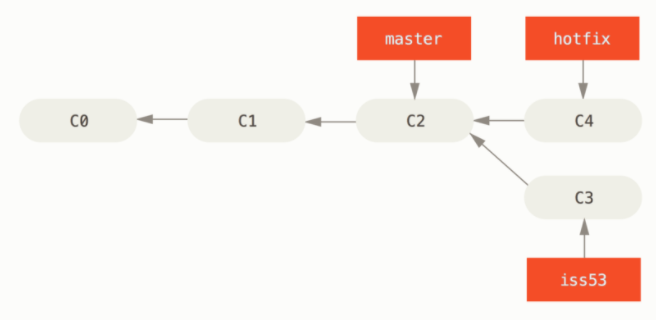
그러면 새로 만든 Branch에서 수정한 내용이 마음에 들어서 Master Branch에 적용하고 싶은 경우 어떻게 하면 될까? Merge기능을 사용하면 된다. 여기서부터 내용이 조금 복잡해지기 때문에 그림과 함께 설명하겠다.

현재 Master Branch에서 새로운 Branch를 만든 뒤 새로운 Branch로 이동하여 작업을 진행한 뒤 commit을 진행한 상태라고 하자 그 상태를 그림으로 나타내면 아래와 같다.



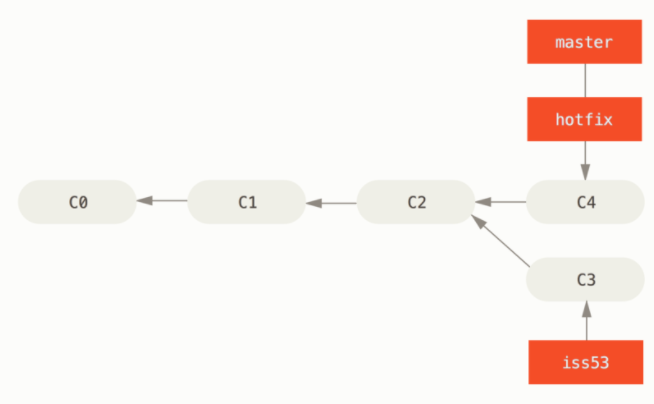
<https://git-scm.com/book> 브랜치와 Merge의 기초 문서에서 발췌

이 상태에서 새로운 문제가 발생하여 Master Branch를 기준으로 새로운 Branch인 hotfix 만들어 수정작업을 진행한 뒤 commit한 상태가 다음과 같다.



<https://git-scm.com/book> 브랜치와 Merge의 기초 문서에서 발췌

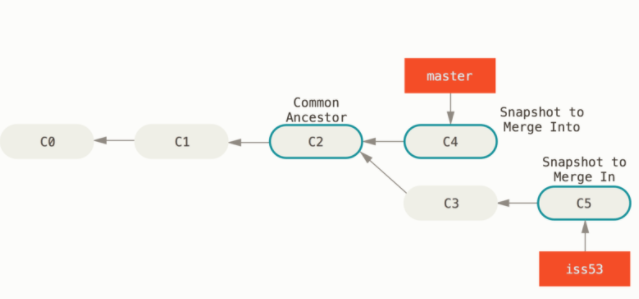
여기서 hotfix해야할 문제가 해결되어 Master Branch에 변경사항을 저장하고 싶을 때 어떻게 해야 할까? 해결책은 단순하다. Git merge hotfix 명령을 통해 hotfix에 적용된 내용을 Master에도 적용할 수 있다. 일반적으로 Master Branch로 hotfix에서 수정된 내용이 적용될 것 같으나 아래 그림과 같이 그저 Master Branch가 가리키는 스냅샷이 달라졌을 뿐이라는 것을 알 수 있다.



<https://git-scm.com/book> 브랜치와 Merge의 기초 문서에서 발췌

이후 git branch -d hotfix 명령을 사용해 임시적으로 만든 hotfix Branch를 제거하고 iss53 Branch에서 작업을 마친 뒤 iss53에서 작업한 내용도 Master Branch에 변경사항을 저장하고 싶은 경우 어떻게 하면 될까?

이 또한 간단하다 git merge master 명령을 통해서 간단하게 변경사항을 적용할 수 있다. 이렇게 갈래가 다른 Branch를 Merge할 경우 단순히 가리키는 스냅샷을 변경하는 것이 아니고 가장 가까운 갈래가 같은 조상까지 거슬러 올라가 3-way Merge가 진행된다.



<https://git-scm.com/book> 브랜치와 Merge의 기초 문서에서 발췌

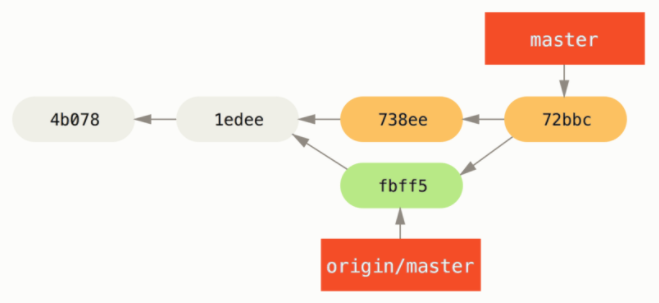
이렇게 쉽게 모든 문제가 해결됐으면 좋겠으나 Merge 과정에서 충돌이 일어날 수 있다. Hotfix Branch와 iss53 Branch에서 같은 파일을 수정한 경우 Merge 과정에서 충돌이 일어난다. 이런 경우 충돌 메시지와 함께 새로운 Commit 즉 스냅샷이 저장되지 않는다. 이런 문제가 발생했을 경우 git status 명령을 통해서 어떤 파일에서 충돌이 일어났는지 확인이 가능하다. 해결법은 크게 2가지가 있다.

1. 파일을 새로 수정 후 git add.
2. Git mergetool 명령 실행.
3. Github를 협업에 활용

위에서는 주로 개인 사용자가 버전관리를 위해 Github를 이용하는 방식을 살펴보았다. 그렇다면 협업에서는 어떻게 Github를 유용하게 사용할 수 있을까?

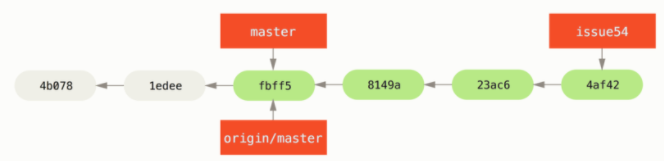
우선 Github서버를 만들어야 한다. Git clone –bare (프로젝트 명) (프로젝트 명.git) 명령을 사용해서 작업공간이 없는 bare 저장소를 생성한다. 이 저장소를 서버로 이동시켜보자. Scp -r 프로젝트 명.git (유저이름)@(서버이름):(원하는 디렉토리 경로 ex. /src/git) 이후 SSH를 통해서 서버에 접근해서 git init –bare –shared 명령을 통해서 그룹 쓰기권한을 제공하면 언제든지 SSH를 통해서 Push 및 Pull할 수 있다. 이로써 원격 저장소 설정을 마무리하면 된다. 단 SSH로 접근할 수 있도록 리눅스 설정을 변경해 주어야 한다. 또 보안을 위해 SSH 공개키 등을 이용해 접속을 암호화해 주는 것을 권장한다.

이렇게 공유되는 원격 저장소를 이용해서 협업하는 방식을 간단한 예로 설명해 보겠다. 두명의 개발자가 하나의 원격 저장소를 공유해서 개발한다고 하자. A B가 모두 본인의 로컬 저장소로 프로젝트를 Clone하고 파일을 수정한 뒤 Commit했다고 하자. 이 경우 문제가 없다면 A B는 서버에 Push할 것이다. 먼저 A가 Push했다고 가정하면 B의 Push는 거절된다. 그 이유는 서버에 변동 사항이 있기 때문에 그 변동사항을 B의 로컬 저장소에서 먼저 Merge한 뒤에 Push해야되기 때문이다. 이 때 B의 로컬저장소의 상태를 보면 다음과 같다.



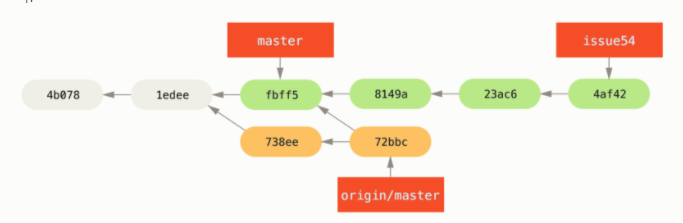
<https://git-scm.com/book> 분산 환경에서의 Git 문서에서 발췌

이후 원격 저장소에 Push함으로써 A B가 작업한 내용이 원격 저장소에 적용될 수 있다. 여기서 주의해야할 점이 존재한다. A는 B가 적용한 변경사항에 대해 아직 모를 뿐 아니라 A의 로컬 저장소에는 이 변경사항이 적용되어 있지 않다는 점이다. 변경사항이 저장되지 않은 채로 새로운 Branch를 만들어 작업한 상황이 아래의 그림이다.



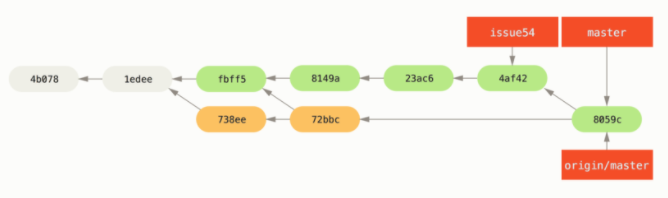
<https://git-scm.com/book> 분산 환경에서의 Git 문서에서 발췌

여기서 서버의 변동사항은 fetch해오면 아래와 같은 복잡한 그림이 된다.



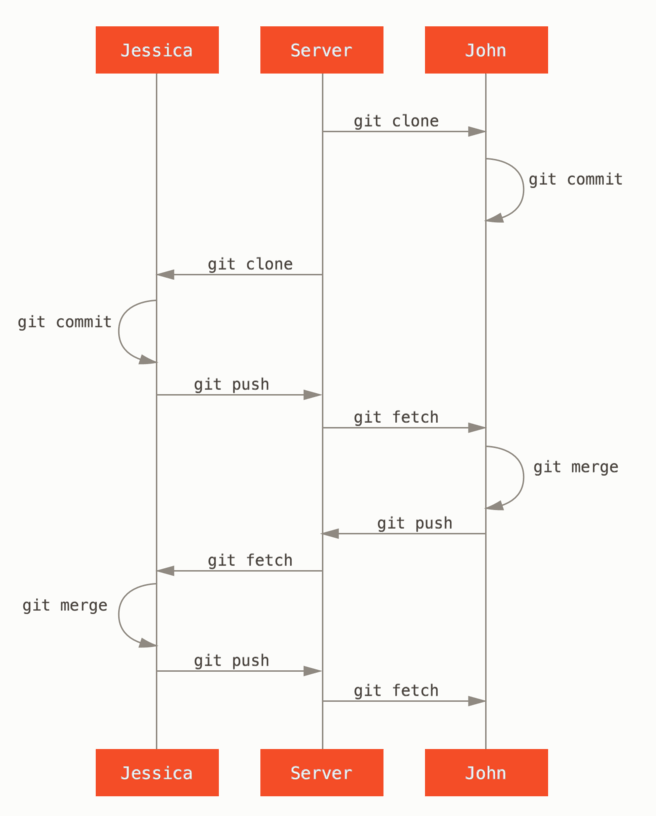
<https://git-scm.com/book> 분산 환경에서의 Git 문서에서 발췌

A는 issue54 Branch와 origin/master Branch를 모두 Merge함으로써 모든 변경사항을 적용할 수 있다. Git merge origin/master 명령과 git merge issue54 명령을 실행함으로써 이 문제를 처리할 수 있다는 뜻이다.



<https://git-scm.com/book> 분산 환경에서의 Git 문서에서 발췌

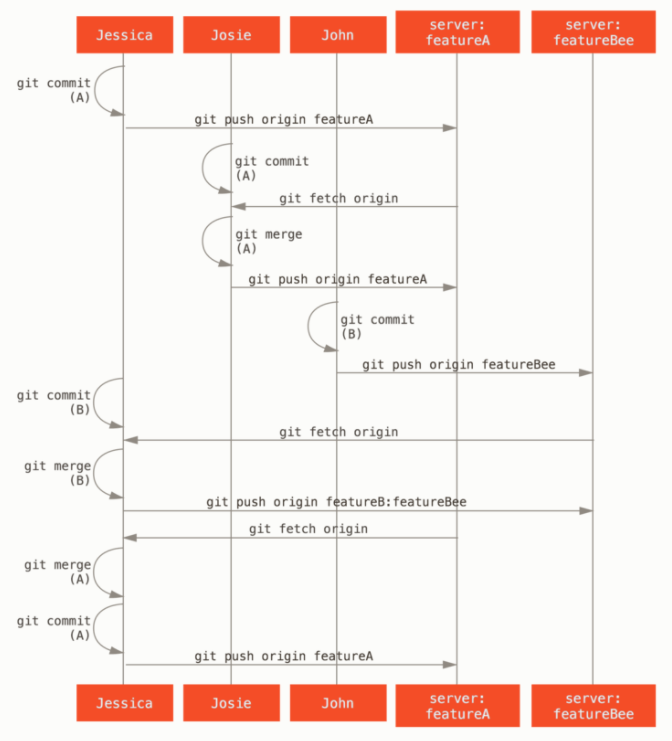
모든 Merge가 끝났을 때 A의 로컬 저장소 Commit 히스토리는 다음과 같다. 이제 Push하면 모든 변경 사항이 원격 저장소에 적용될 것이다.



<https://git-scm.com/book> 분산 환경에서의 Git 문서에서 발췌

위의 과정을 워크플로로 살펴보면 다음과 같다.

다수의 인원이 협업하는 경우는 위의 과정을 이해했다면 그렇게 어렵지 않다. 협업할 Branch를 팀별로 만들어 Branch를 팀별로 공유하고 해당 Branch를 2인 협업 시의 origin/master Branch로 생각하며 작업을 진행하면 된다. 이후 두 팀의 작업이 끝나 최종 Merge를 할 경우 각 팀의 Branch를 Master Branch에 Merge한 이후 원격 저장소에 Push하면 된다.



<https://git-scm.com/book> 분산 환경에서의 Git 문서에서 발췌

1. 참고문서

<https://git-scm.com/book/ko> - 깃 공식 가이드 문서.