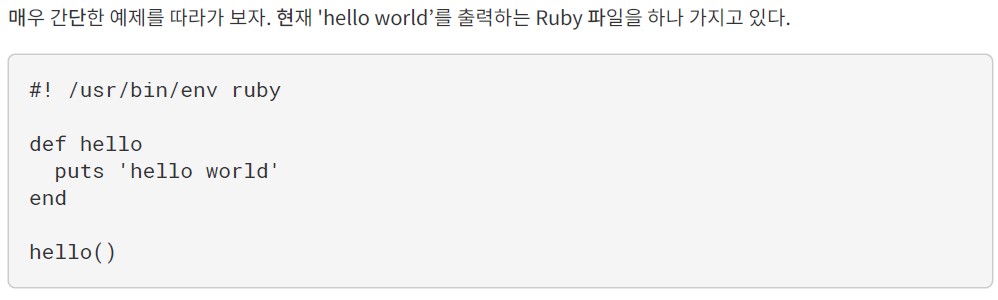
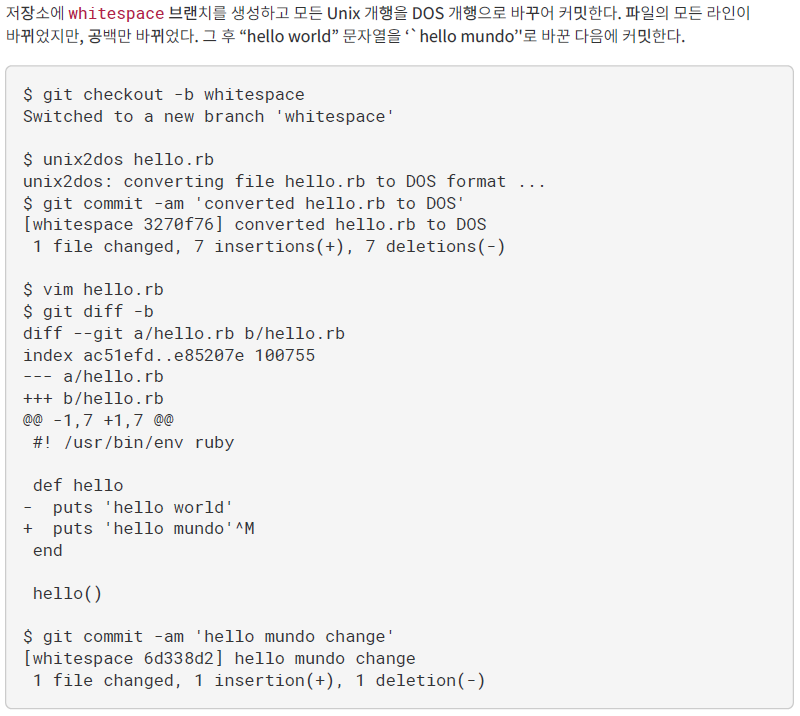
**고급 Merge**

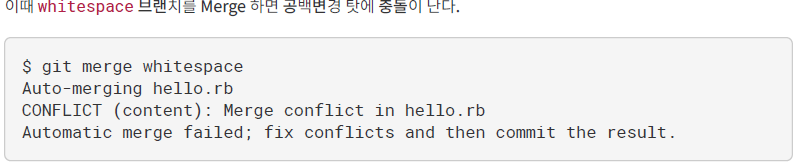
* 오랫동안 따로 유지한 두 브랜치를 Merge하려면 몇 가지 해야할 일이 있음

Merge 충돌

* Merge 하기 전에 충돌을 피하기 위해서 워킹 디렉토리를 깔끔하게 하는 것이 좋음
* 워킹 디렉토리에 작업하던 게 있으면 임시 브랜치에 커밋하거나 Stash 해둔다.



윈도우(Win32)에서 작성한 텍스트 파일을 유닉스/리눅스 환경에서 읽으면 에러가 나거나 ^M 이라는 이상한 문자가 찍힌다.  
  
MS윈도우는 텍스트 파일의 끝에서 CR-LF로 줄바꿈을 하고, 유닉스는 LF 문자로 줄바꿈을 하기 때문이다.



Merge 취소하기

취소 방법 1. 상황 벗어나기

git merge --abort

이 명령으로 간단히 Merge 하기 전으로 되돌린다.

Merge 전에 워킹 디렉토리에서 Stash 하지 않았거나 커밋하지 않은 파일이 존재하고 있었을 때를 제외하곤 잘 돌아간다.

취소 방법 2. Merge를 처음부터 다시 하기

git reset --hard HEAD

워킹 디렉토리를 그 시점으로 완전히 되돌려서 저장하지 않은 것은 사라진다.

공백 무시하기

-Xignore-all-space: 모든 공백을 무시

-Xignore-space-change: 뭉쳐 있는 공백을 하나로 취급

무엇이 바뀌었는지 확인하려면 git diff

Merge 후의 결과를 Merge 하기 전의 브랜치와 비교하려면 git diff –ours

Merge 할 파일을 가져온 쪽과 비교해서 무엇이 바뀌었는지 보려면 git diff --theirs

양쪽 모두와 비교하여 바뀐 점을 알아보려면 git diff –base



충돌 파일 Checkout

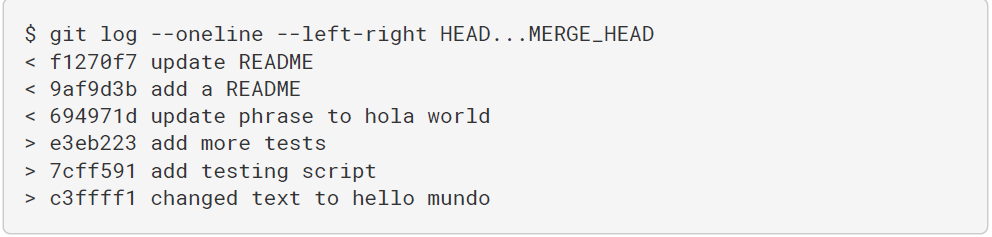
master 에만 있는 세 개의 커밋과 mundo 브랜치에만 존재하는 또 다른 세 개의 커밋이 있다. master 브랜치에서 mundo 브랜치를 Merge 하면 충돌이 난다.

git checkout 명령에 --conflict 옵션

-파일을 다시 Checkout 받아서 충돌 표시된 부분을 교체

Merge 로그

과거를 살짝 들춰보면 개발 당시에 같은 곳을 고쳐야만 했던 이유를 밝혀내는 데 도움이 됨



위와 같이 총 6개의 커밋을 볼 수 있다. 커밋이 어떤 브랜치에서 온 것인지 보여준다.

--merge 옵션: 충돌이 발생한 파일이 속한 커밋만 보여준다.

-p 옵션: 충돌 난 파일의 변경사항만 볼 수 있다. 왜 충돌이 났는지 또 이를 해결하기 위해 어떻게 해야 하는지 이해하는 데 진짜 유용하다 함.

Combined Diff 형식

Merge가 성공적으로 끝난 파일은 Staging Area에 올려놓음

이 상태에서 충돌 난 파일들이 그대로 있을 때 git diff 명령을 실행하면 충돌 난 파일이 무엇인지 알 수 있다. 어떤 걸 더 고쳐야 하는지 아는 데에 도움이 된다.



첫 번째 컬럼은 “ours” 브랜치와 워킹 디렉토리의 차이(추가 또는 삭제)를 보여준다.

두 번째 컬럼은 “theirs” 와 워킹 디렉토리 사이의 차이를 나타낸다.

<<<<<<< 와 >>>>>>> 충돌 마커 표시는 어떤 쪽에도 존재하지 않고 추가된 코드

충돌을 다 해결하고 git diff 명령을 다시 실행



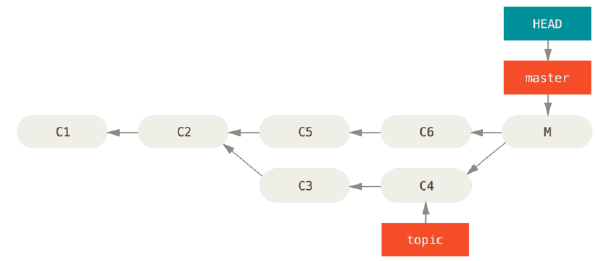
이 결과는 세 가지 사실을 보여준다.

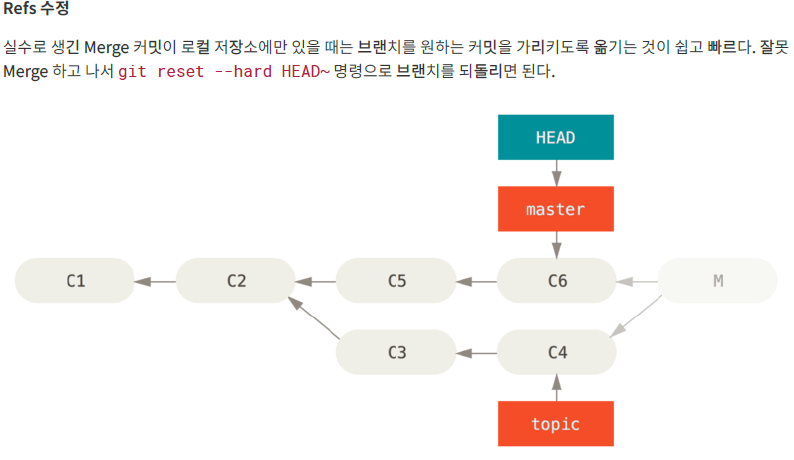
“hola world”는 Our 브랜치에 있었지만 워킹 디렉토리에는 없다.

“hello mundo”는 Their 브랜치에 있었지만 워킹 디렉토리에는 없다.

“hola mundo”는 어느 쪽 브랜치에도 없고 워킹 디렉토리에는 있다.

Merge 되돌리기

토픽 브랜치에서 일을 하다가 master 로 잘못 Merge 했다고 생각해보자.



reset --hard 명령

1. HEAD의 브랜치를 지정한 위치로 옮긴다. 이 경우엔 master 브랜치를 Merge 커밋(C6) 이전으로 되돌린다.

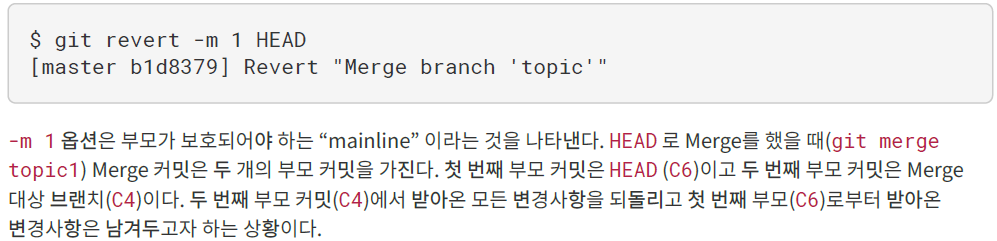
2. Index를 HEAD의 내용으로 바꾼다.

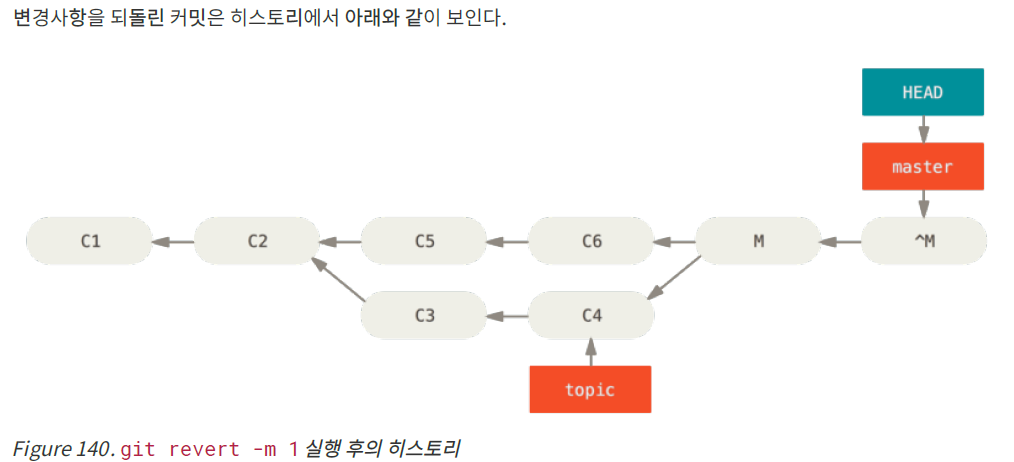
3. 워킹 디렉토리를 Index의 내용으로 바꾼다.

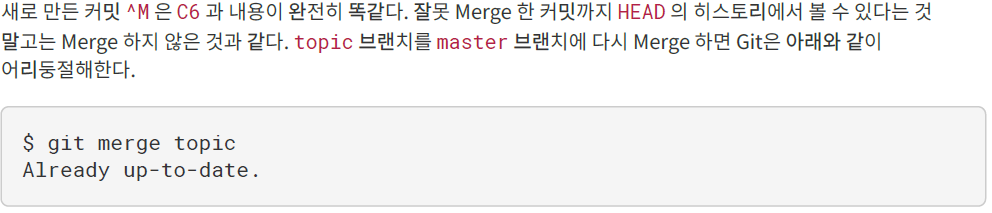
이 방법의 단점: 히스토리를 다시 씀

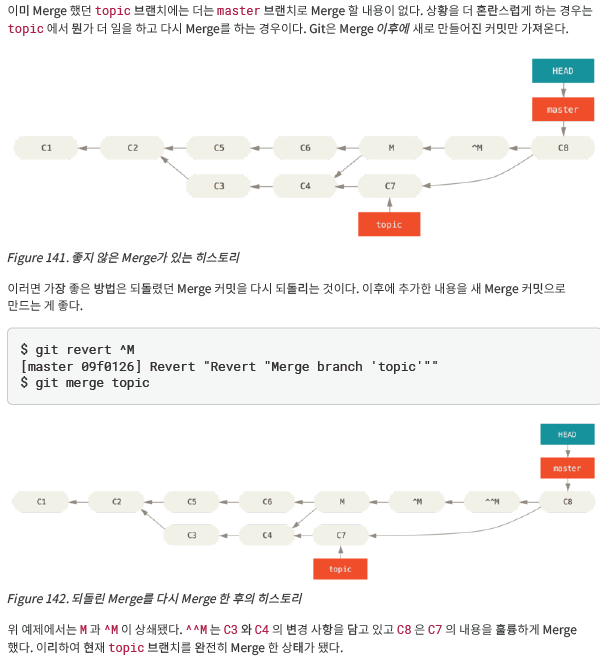
다시 쓰는 커밋이 이미 다른 사람들과 공유한 커밋이라면 reset 하지 않는 게 좋다.

커밋 되돌리기

Revert: 모든 변경사항을 취소하는 새로운 커밋을 만듬







다른 방식의 Merge들도 꽤 있음. (시간 없어서 생략 궁금하면 찾아보셈 ㅈㅅㅈㅅ)

Rerere

“reuse recorded resolution” 이라고 해서 기록한 해결책 재사용하기란 뜻으로 이름 그대로 동작

Git은 충돌이 났을 때 각 코드 덩어리를 어떻게 해결했는지 기록을 해 두었다가 나중에 같은 충돌이 나면 기록을 참고하여 자동으로 해결

언제 쓰냐?

호흡의 브랜치를 깔끔하게 Merge하고 싶은데 Merge 커밋은 많이 만들고 싶지 않을 때

* rerere 기능을 켜고 자주 Merge를 해서 충돌을 해결하고 Merge 이전으로 돌아간다. 이 과정을 반복해서 기록을 쌓아두면 rerere 기능은 나중에 한 번에 Merge 할 때 기록을 참고한다.

브랜치를 Rebase 할 때

서브모듈

깃의 서브모듈은 깃 리파지터리 아래에 다른 하위 깃 리파지터리를 관리하기 위한 도구이다.

현재 작업하고 있는 아래와 같은 부모 프로젝트가 있고,

git@github.com:ohgyun/submodule\_test\_parent.git

이 프로젝트 하위에 이미 존재하는 다른 깃 프로젝트를 포함하려고 한다고 가정해보자.

아래는 자식 프로젝트의 리파지터리 링크이다.

git@github.com:ohgyun/submodule\_test\_child.git

워킹 디렉토리는 ~/mywork 라고 가정하고, 여기에서 부모 프로젝트를 클론하는 걸로 시작해보자.

~/mywork

$ git clone git@github.com:ohgyun/submodule\_test\_parent.git

~/mywork

$ cd submodule\_test\_parent

**# 서브 모듈 추가하기**

서브모듈은 아래 명령으로 추가할 수 있다.

$ git submodule add <repository> [path]

path 는 생략 가능하고, 생략 시 리파지터리 이름과 동일한 디렉토리를 사용한다.

우리는 생략하고 아래와 같이 추가해보자.

~/mywork/submodule\_test\_parent

$ git submodule add https://github.com/ohgyun/submodule\_test\_child

Cloning into 'submodule\_test\_child'...

remote: Counting objects: 3, done.

remote: Compressing objects: 100% (2/2), done.

remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0

Unpacking objects: 100% (3/3), done.

Checking connectivity... done.

아무 것도 없는 상태에서 서브모듈을 추가하고 상태를 조회해보면,

아래와 같이 .gitmodules 파일과 추가한 서브모듈 디렉토리가 생성된다.

~/mywork/submodule\_test\_parent

$ git status

On branch master

Your branch is up-to-date with 'origin/master'.

Changes to be committed:

  (use "git reset HEAD <file>..." to unstage)

    new file:   .gitmodules

    new file:   submodule\_test\_child

.gitmodules 파일에는 프로젝트에서 관리하고 있는 서브모듈 목록에 대한 정보가 들어있다. 조회해보면, 아래와 같이 서브모듈 정보를 볼 수 있다.

~/mywork/submodule\_test\_parent

$ cat .gitmodules

[submodule "submodule\_test\_child"]

    path = submodule\_test\_child

    url = git@github.com:ohgyun/submodule\_test\_child.git

submodule\_test\_child 는 하위 깃 리파지터리를 포함하고 있는 디렉토리이지만, 깃에서는 서브모듈 정보를 포함하고 있는 파일처럼 처리한다.

실제로 하위 깃 리파지터리의 커밋 해시를 포함하고 있으며, 이 정보를 diff 로 확인할 수 있다.

~/mywork/submodule\_test\_parent

$ git diff --cached submodule\_test\_child/

diff --git a/submodule\_test\_child b/submodule\_test\_child

new file mode 160000

index 0000000..0d2bb5b

--- /dev/null

+++ b/submodule\_test\_child

@@ -0,0 +1 @@

+Subproject commit 0d2bb5b91c235fa77fdd8859d9ecbd270fd576d2

명령 옵션의 --cached 은 기존에 서브 모듈이 없어 캐시된 값과 비교하기 위한 옵션이라 무시해도 되고, 유심히 봐야할 것은 추가된 파일의 모드와 diff 내용이다.

mode 160000 이라는 건, 일반 파일이 아니라는 의미이고, +Subproject commit 0d2bb5b... 는 현재 부모 리파지터리에서 하위 리파지터리의 0d2bb5b 커밋을 바라보고 있단 의미이다.

서브모듈을 추가한 후 submodule\_test\_child 디렉토리에 들어가보면, 대상 리파지터리가 fetch 되어 있는 걸 확인할 수 있다.

이제 서브모듈을 변경했다는 커밋을 하나 추가하자.

~/mywork/submodule\_test\_parent

$ git commit -am "Add submodule"

[master b90da54] Add submodule

**2. 부모 프로젝트에서 자식 프로젝트의 내용 변경하고 업데이트하기**

이제, 부모 프로젝트 내에서 자식 프로젝트의 내용을 변경해보자.

먼저 부모 리파지터리의 상태를 조회해보자.

~/mywork/submodule\_test\_parent

$ git log --pretty=short -1

commit b90da54bc5f7b0ca12eb313e1a57581f35ddce38

Author: Ohgyun Ahn <ohgyun@gmail.com>

    Add submodule

리모트도 조회해보자.

~/mywork/submodule\_test\_parent

$ git remote -v

origin    git@github.com:ohgyun/submodule\_test\_parent.git (fetch)

origin    git@github.com:ohgyun/submodule\_test\_parent.git (push)

다음으로 자식 리파지터리의 상태도 조회해보자.

~/mywork/submodule\_test\_parent

$ cd submodule\_test\_child/

~/mywork/submodule\_test\_parent/submodule\_test\_child

$ git log --pretty=short -1

commit 0d2bb5b91c235fa77fdd8859d9ecbd270fd576d2

Author: Ohgyun Ahn <ohgyun@gmail.com>

    Initial commit

자식 리파지터리의 리모트도 조회해보자.

~/mywork/submodule\_test\_parent/submodule\_test\_child

$ git remote -v

origin    https://github.com/ohgyun/submodule\_test\_child (fetch)

origin    https://github.com/ohgyun/submodule\_test\_child (push)

각 리파지터리는 각각의 origin 을 갖고 있고,

부모 리파지터리는 b90da54 커밋을, 자식 리파지터리는 0d2bb5b 커밋을 바라보고 있다.

이제 자식 리파지터리(submodule\_test\_child 디렉토리)에서 새 커밋을 생성해보자.

~/mywork/submodule\_test\_parent/submodule\_test\_child

$ git commit --allow-empty -m "Add new commit"

[master af2a90a] Add new commit

참고로, --allow-empty 옵션은 빈 커밋을 생성하는 옵션이다.

상태를 조회해보자.

~/mywork/submodule\_test\_parent/submodule\_test\_child

$ git log --pretty=short -1

commit af2a90a19766d5ab5ed7d5f59e88403245f99ab1

Author: Ohgyun Ahn <ohgyun@gmail.com>

    Add new commit

새 커밋으로 자식 리파지터리의 커밋이 0d2bb5b 에서 af2a90a 로 변경됐다.

부모 리파지터리로 돌아가 상태를 조회해보자.

~/mywork/submodule\_test\_parent/submodule\_test\_child

$ cd ..

~/mywork/submodule\_test\_parent

$ git status

On branch master

Your branch is ahead of 'origin/master' by 1 commit.

  (use "git push" to publish your local commits)

Changes not staged for commit:

  (use "git add <file>..." to update what will be committed)

  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   submodule\_test\_child (new commits)

no changes added to commit (use "git add" and/or "git commit -a")

변경된 정보를 조회해보면, 서브모듈의 값이 변경된 걸 알 수 있다.

~/mywork/submodule\_test\_parent

$ git diff

diff --git a/submodule\_test\_child b/submodule\_test\_child

index 0d2bb5b..af2a90a 160000

--- a/submodule\_test\_child

+++ b/submodule\_test\_child

@@ -1 +1 @@

-Subproject commit 0d2bb5b91c235fa77fdd8859d9ecbd270fd576d2

+Subproject commit af2a90a19766d5ab5ed7d5f59e88403245f99ab1

현재 부모 리파지터리이니까, 서브모듈을 변경했다는 커밋을 추가해보자.

~/mywork/submodule\_test\_parent

$ git commit -am "Update submodule"

[master 3a0e087] Update submodule

 1 file changed, 1 insertion(+), 1 deletion(-)

새로 생성된 부모 리파지터리의 커밋 3a0e087에서는, 자식 리파지터리의 af2a90a 커밋을 바라보고 있다.

하지만 아직 자식 리파지터리는 로컬만 변경되어 있는 상태이기 때문에, 변경 내용을 리모트에도 적용해줘야 한다.

~/mywork/submodule\_test\_parent

$ cd submodule\_test\_child/

~/mywork/submodule\_test\_parent/submodule\_test\_child

$ git log --pretty=short -1

commit af2a90a19766d5ab5ed7d5f59e88403245f99ab1

Author: Ohgyun Ahn <ohgyun@gmail.com>

    Add new commit

~/mywork/submodule\_test\_parent/submodule\_test\_child

$ git push origin master

Counting objects: 1, done.

Delta compression using up to 4 threads.

Compressing objects: 100% (3/3), done.

Writing objects: 100% (3/3), 440 bytes | 0 bytes/s, done.

Total 1 (delta 1), reused 0 (delta 0)

To git@github.com:ohgyun/submodule\_test\_child.git

   0d2bb5b..af2a90a  master -> master

필요하다면, 부모 리파지터리도 리모트에 푸시하면 된다.

~/mywork/submodule\_test\_parent/submodule\_test\_child

$ cd ..

~/mywork/submodule\_test\_parent

$ git push origin master

Bundle

데이터를 한 파일에 몰아넣는 것

언제 쓰냐?

네트워크가 불통인데 변경사항을 동료에게 보낼 때

출장을 나갔는데 보안상의 이유로 로컬 네트워크에 접속하지 못할 때

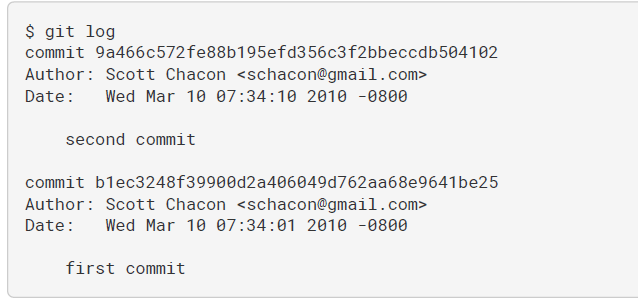
통신 인터페이스 장비가 고장났을 때

갑자기 공용 서버에 접근하지 못할 때

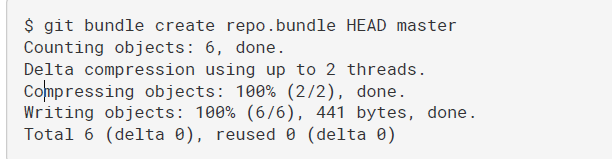
누군가에게 수정사항을 이메일로 보내야 하는데 40개 씩이나 되는 커밋을 format-patch 로 보내고 싶지 않을 때

bundle 명령은 보통 git push 명령으로 올려 보낼 모든 것을 감싸서 한 바이너리 파일로 만든다. 이 파일을 이메일로 보내거나 USB로 다른 사람에게 보내서 다른 저장소에 풀어서 (Unbundle) 사용한다.

이 저장소에는 커밋 두 개가 있다.



이 저장소를 다른 사람에게 통째로 보내고 싶은데 그 사람의 저장소에 Push 할 권한이 없거나, 그냥 Push 하고 싶지 않을 때, git bundle create 명령으로 Bundle을 만들 수 있다.



이렇게 repo.bundle 이라는 이름의 파일을 생성할 수 있다. 이 파일에는 이 저장소의 master 브랜치를 다시 만드는 데 필요한 모든 정보가 다 들어 있다. bundle 명령으로 모든 Refs를 포함하거나 Bundle에 포함할 특정 구간의 커밋을 지정할 수 있다. 이 Bundle을 다른 곳에서 Clone 하려면 위의 명령처럼 HEAD Refs를 포함해야 한다.

repo.bundle 파일을 다른 사람에게 이메일로 전송하거나 USB 드라이브에 담아서 나갈 수 있다.

혹은 repo.bundle 파일을 일할 곳으로 어떻게든 보내놓으면 이 Bundle 파일을 마치 URL에서 가져온 것처럼 Clone 해서 사용할 수 있다.

Bundle 파일에 HEAD Refs를 포함하지 않으려면 -b master 옵션을 써주거나 포함시킬 브랜치를 지정해줘야 한다. 그렇지 않으면 Git은 어떤 브랜치로 Checkout 할지 알 수 없다.

Replace

replace 명령은 "어떤 개체를 읽을 때 항상 다른 개체로 보이게" 한다. 히스토리에서 어떤 커밋이 다른 커밋처럼 보이도록 할 때 이 명령이 유용하다.

예를 들어 현재 프로젝트의 히스토리가 아주 방대한 상태다. 히스토리를 둘로 나누어서 새로 시작하는 개발자에게는 히스토리를 아주 간단한 몇 개의 커밋으로 만들어서 제공하고, 프로젝트 히스토리를 분석할 사람에게는 전체 히스토리를 제공하는 상황을 생각해보자. replace 명령으로 간단해진 히스토리를 전체 히스토리의 마지막 부분에 연결해서 사용할

수 있다. 이렇게 히스토리를 변경하는 데도 커밋을 새로 쓰지 않는 매우 훌륭한 기능이다(Rebase를 생각해보면 한 부모를 변경하면 이후의 커밋은 모두 재작성된다).

Credential 저장소

SSH 프로토콜을 사용하여 리모트 저장소에 접근할 때 Passphase 없이 생성한 SSH Key를 사용하면 사용자이름과 암호를 입력하지 않고도 안전하게 데이터를 주고받을 수 있다. 반면 HTTP 프로토콜을 사용하는 경우는 매번 사용자이름과 암호를 입력해야 한다.

다행히도 Git은 이렇게 매번 인증정보(Credential)를 입력하는 경우 인증정보를 저장해두고 자동으로 입력해주는 시스템을 제공한다. Git Credential 기능이 제공하는 옵션은 아래와 같다.

• 일단 기본적으로 아무런 설정도 하지 않으면 어떤 암호도 저장하지 않는다. 이 경우 인증이 필요한 때 매번 사용자이름과 암호를 입력해야 한다.

• “cache” 모드로 설정하면 일정 시간 동안 메모리에 사용자이름과 암호 같은 인증정보를 기억한다. 이 정보를 Disk에 저장하지는 않으며 메모리에서도 15분 까지만 유지한다.

• “store” 모드로 설정하면 인증정보를 Disk의 텍스트 파일로 저장하며 계속 유지한다. 계속 유지한다는 말은 리모트의 인증정보를 변경하지 않는 한 다시 인증정보를 입력하지 않아도 접근할 수 있다는 말이다. “store” 모드를 사용할 때 주의할 점은 인증정보가 사용자 홈 디렉토리 아래에 일반 텍스트 파일로 저장된다는 점이다.

• Mac에서 Git을 사용하는 경우 “osxkeychain” 모드를 사용하면 Mac에서 제공하는 Keychain 시스템에 사용자이름과 암호를 현재 로그인 계정에 속하게 저장한다. “store” 모드하면 인증정보를 Disk에 저장하고 인증정보가 만료되지 않는 점은 같지만, Safari 브라우저가 인증정보를 저장하는 것과 같은 수준으로 암호화해서 저장한다는 점이 다르다.

• Windows 환경에서는 “wincred” 라는 Helper가 있다. “osxkeychain” Helper와 비슷하게 동작하며 Windows Credential Store를 사용하여 안전하게 인증정보를 저장한다.