|  |
| --- |
| clone |
| git clone https://.....git |

https://git-scm.com/book/ko/v2/Git%EC%9D%98-%EA%B8%B0%EC%B4%88-%EB%90%98%EB%8F%8C%EB%A6%AC%EA%B8%B0

|  |
| --- |
| .gitignore |
| \*\*/\*.class  /.git  \*\*/.git/  .classpath  .project  .gradle  .settings  # Logs and databases #  ######################  \*\*/\*.log  \*\*/\*.sql  \*\*/\*.sqlite  \*.log  \*.sql  \*.sqlite |
| 1. 폴더 .gitignore 설정할 때  폴더/  2. 만약, /폴더1/폴더2/폴더3 에서 폴더2 이하의 폴더를 무시하게 설정할 때  \*\*/폴더2/  만약, 폴더3 이하를 무시하고 싶으면  \*\*/폴더2/폴더3/ |

|  |
| --- |
| status 파일상태 |
| git status -s 파일상태 짤막하게 보여주기 |

|  |
| --- |
| reset (add 취소) |
| git reset \*\*/~\*  git reset |

|  |
| --- |
| commit |
| git commit –m “comment”  커밋 히스토리조회 : git log |

|  |
| --- |
| checkout |
| git checkout –f 모르고 delete한거 체크아웃 다시하기(현재 커밋된거 다시 불러와서 덮어쒸우기) |

|  |
| --- |
| branch |
| 리스트보기 : git branch  브런치별 마지막 커밋내용 : git branch –v  브런치삭제 : branch –d name  강제 삭제 : -D |

|  |
| --- |
| git config credential.helper store |
| 비밀번호 저장해놓기. |

|  |
| --- |
| log |
| log 로그보기 히스토리보기  log –p 파일 파일별 보기 |

|  |
| --- |
| stash |
| [시나리오] http://www.dreamy.pe.kr/zbxe/CodeClip/157790 branch에서 새로운 기능을 개발 중인데 긴급한 수정사항이 생겼을 경우.  방법 1. 현재까지 작업한 파일들을 어딘가에 백업해두고 branch를 롤백한다. 그리고 긴급한 수정 사항을 처리한 후 백업해둔 파일들을 다시 가져와서 개발을 진행.  방법 2. 다른 디렉토리에서 현재 배포 버전을 받은 뒤 수정 사항을 처리하고 배포.  이 외에도 몇몇 꼼수가 있겠지만 git stash를 사용하면 간단히 해결할 수 있다.  #현재 작업을 저장해두고 branch를 head로 돌린다.(git reset --hard) git stash save  #저장되어 있는 stash들 보기 git stash list  #stash를 꺼내와서 적용 #stash들은 stack에 저장된다. 따라서 가장 최근에 save한 stash가 현재 branch에 적용된다. git stash pop  #git stash pop 과 비슷한 명령어지만 stash list에서 삭제하지 않는다는 점이 다르다. git stash apply  #필요 없는 stash를 삭제 git stash drop  #전체 stash list를 삭제 git stash clear  git stash pop 을 했는데 conflict가 발생한 경우.  stash list 에서는 그대로 남아 있으므로 수동으로 drop 을 해줘야 한다. 왜 list 에서 삭제하지 않고 남겨두는걸까 생각해보면 남겨두는게 맞다. 문제가 생겼을 때 문제를 처리하고 다시 git stash pop을 할 수 있다. |

|  |
| --- |
| 트랙 스테이즈쪽에 파일을 남겨두고 그냥 DELETE쪽만 repository(commit stage)쪽에만 delete 보낼때 |
| <http://stackoverflow.com/questions/1143796/remove-a-file-from-a-git-repository-without-deleting-it-from-the-local-filesyste>  file :  git rm --cached mylogfile.log  For a directory:  git rm --cached -r mydirectory  다시 추가하고싶을때. 즉 캐쉬쪽에 빠진걸 다시 add하고싶을때에는  git add -f ./file |

|  |
| --- |
| 강제적 add force |
| git add -f ./file |

|  |
| --- |
| .gitignore gitinore사이트 (https://www.gitignore.io/) |
| 화이트 리스트 whitelist  \*  !.gitignore  !src/  !src/\*  !build.gradle  !settings.gradle |

**개별파일 원복**

git checkout  -- <파일> : 워킹트리의 수정된 파일을 index에 있는 것으로 원복

git checkout HEAD -- <파일명> : 워킹트리의 수정된 파일을 HEAD에 있는 것으로 원복(이 경우 --는 생략가능)

git checkout FETCH\_HEAD -- <파일명> : 워킹트리의 수정된 파일의 내용을 FETCH\_HEAD에 있는 것으로 원복? merge?(이 경우 --는 생략가능)

**index 추가 취소**

git reset -- <파일명> : 해당 파일을 index에 추가한 것을 취소(unstage). 워킹트리의 변경내용은 보존됨. (--mixed 가 default)

git reset HEAD <파일명> : 위와 동일

**commit 취소**

git reset HEAD^ : 최종 커밋을 취소. 워킹트리는 보존됨. (커밋은 했으나 push하지 않은 경우 유용)

git reset HEAD~2 : 마지막 2개의 커밋을 취소. 워킹트리는 보존됨.

git reset --hard HEAD~2 : 마지막 2개의 커밋을 취소. index 및 워킹트리 모두 원복됨.

git reset --hard ORIG\_HEAD : 머지한 것을 이미 커밋했을 때,  그 커밋을 취소. (잘못된 머지를 이미 커밋한 경우 유용)

git revert HEAD : HEAD에서 변경한 내역을 취소하는 새로운 커밋 발행(undo commit). (커밋을 이미 push 해버린 경우 유용)

**워킹트리 전체 원복**

git reset --hard HEAD : 워킹트리 전체를 마지막 커밋 상태로 되돌림. 마지막 커밋이후의 워킹트리와 index의 수정사항 모두 사라짐.   
                                  (변경을 커밋하지 않았다면 유용)

git checkout HEAD . : ??? 워킹트리의 모든 수정된 파일의 내용을 HEAD로 원복.

git checkout -f : 변경된 파일들을 HEAD로 모두 원복(아직 커밋하지 않은 워킹트리와 index 의 수정사항 모두 사라짐. 신규추가 파일 제외)

**\* 참조 : reset 옵션**

--soft : index 보존, 워킹트리 보존. 즉 모두 보존.

--mixed : index 취소, 워킹트리만 보존 (기본 옵션)

--hard : index 취소, 워킹트리 취소. 즉 모두 취소.

**\* untracked 파일 제거**

git clean -f

git clean -f -d : 디렉토리까지 제거

git log 명령을 실행하면 저장소의 커밋 히스토리를 시간순으로 보여준다

**서브모듈**

프로젝트를 수행하다 보면 다른 프로젝트를 사용해야 하는 경우가 종종 있다. 보통 사용할 프로젝트들은 독립적으로 개발된 라이브러리들이다. 이런 상황에서 자주 생기는 이슈는, 두 프로젝트를 서로 별개로 다루면서도 그 중 하나를 다른 하나 안에서 사용할 수 있어야 한다는 것이다.

Atom 피드를 제공하는 웹사이트를 만든다고 가장하자. Atom 피드를 생성하는 코드는 직접 작성하지 않고 라이브러리를 가져다 쓰기로 했다. 그러면 CPAN이나 Ruby gem 같은 라이브러리 관리 도구를 사용하거나 해당 소스코드를 프로젝트로 복사해야 한다. 사실 라이브러리를 수정하는 것은 어렵다. 하지만 수정한 라이브러리를 모든 사용자가 이용할 수 있도록 배포하는 것은 더 어렵다. 그래서 프로젝트에 라이브러리 코드를 포함시켜서 수정하는 방법도 사용한다. 이렇게 라이브러리 코드를 포함시키면 원래 라이브러리 프로젝트의 코드와 Merge하기 어렵게 된다.

Git의 서브모듈은 이런 문제를 해결해준다. 서브모듈은 Git 저장소 안에 다른 Git 저장소를 둘 수 있게 해준다. 이렇게 해도 두 Git 저장소 모두 여전히 독립적으로 관리된다.

[**서브모듈 시작하기**](https://git-scm.com/book/ko/v1/Git-%EB%8F%84%EA%B5%AC-%EC%84%9C%EB%B8%8C%EB%AA%A8%EB%93%88#서브모듈-시작하기)

한 번 Ruby 웹서버 게이트웨이 인터페이스인 Rack 라이브러리를 프로젝트에 추가해보자. 추가하고 나서도 앞으로 여전히 해당 저장소에서 관리할 수 있기 때문에 마음 놓고 코드를 수정할 수 있다. 먼저 git submodule add 명령으로 프로젝트를 서브모듈로 추가한다:

$ git submodule add git://github.com/chneukirchen/rack.git rack

Initialized empty Git repository in /opt/subtest/rack/.git/

remote: Counting objects: 3181, done.

remote: Compressing objects: 100% (1534/1534), done.

remote: Total 3181 (delta 1951), reused 2623 (delta 1603)

Receiving objects: 100% (3181/3181), 675.42 KiB | 422 KiB/s, done.

Resolving deltas: 100% (1951/1951), done.

이제 프로젝트 디렉토리를 보면 rack이라는 디렉토리가 생겼을 것이다. 그 디렉토리가 Rack 프로젝트이다.rack 디렉토리 안에서 수정하고 Push할 권한이 있는 저장소를 하나 추가하고 나서 그 저장소에 Push한다. 물론 원래 프로젝트 저장소에서도 Fetch하고 Merge할 수 있다. 서브모듈을 추가한 직후 바로 git status라는 명령을 실행하면 아래와 같이 두 파일이 생긴 것을 알 수 있다:

$ git status

# On branch master

# Changes to be committed:

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

#

# new file: .gitmodules

# new file: rack

#

.gitmodules 파일을 살펴보자. 이 것은 로컬 디렉토리와 프로젝트 URL의 매핑 정보가 저장된 설정파일이다:

$ cat .gitmodules

[submodule "rack"]

path = rack

url = git://github.com/chneukirchen/rack.git

서브모듈 개수만큼 이 항목이 생긴다. 이 파일도 .gitignore 파일처럼 버전 관리된다. 다른 파일처럼 Push하고 풀한다. 이 프로젝트를 Clone하는 사람은 .gitmodules 파일을 보고 어떤 서브모듈 프로젝트가 있는지 알 수 있다.

.gitmodules은 살펴봤고 이제 rack 항목에 대해 살펴보자. git diff 명령을 실행시키면 흥미로운 점을 발견할 수 있다:

$ git diff --cached rack

diff --git a/rack b/rack

new file mode 160000

index 0000000..08d709f

--- /dev/null

+++ b/rack

@@ -0,0 +1 @@

+Subproject commit 08d709f78b8c5b0fbeb7821e37fa53e69afcf433

Git은 rack 디렉토리를 서브모듈로 취급하기 때문에 파일들을 직접 추적하지 않고 커밋 하나만 저장한다.rack 디렉토리에서 수정을 하고 커밋하면 다른 사람이 같은 환경을 만들 수 있도록 HEAD가 가리키는 커밋이 슈퍼프로젝트에 저장된다.

master처럼 브랜치 이름 같은 레퍼런스가 저장되는 것이 아니라 커밋의 SHA-1 값이 저장된다.

슈퍼프로젝트도 커밋해야 된다:

$ git commit -m 'first commit with submodule rack'

[master 0550271] first commit with submodule rack

2 files changed, 4 insertions(+), 0 deletions(-)

create mode 100644 .gitmodules

create mode 160000 rack

rack 디렉토리의 모드는 160000이다. 160000 모드는 일반적인 파일이나 디렉토리가 아니라는 의미다.

하위 프로젝트의 마지막 커밋이 바뀔 때마다 슈퍼프로젝트에 저장된 커밋도 바꿔준다. rack 디렉토리를 별도의 프로젝트로 취급하기 때문에 모든 Git 명령은 독립적으로 동작한다:

$ git log -1

commit 0550271328a0038865aad6331e620cd7238601bb

Author: Scott Chacon <schacon@gmail.com>

Date: Thu Apr 9 09:03:56 2009 -0700

first commit with submodule rack

$ cd rack/

$ git log -1

commit 08d709f78b8c5b0fbeb7821e37fa53e69afcf433

Author: Christian Neukirchen <chneukirchen@gmail.com>

Date: Wed Mar 25 14:49:04 2009 +0100

Document version change

[**서브모듈이 있는 프로젝트 Clone하기**](https://git-scm.com/book/ko/v1/Git-%EB%8F%84%EA%B5%AC-%EC%84%9C%EB%B8%8C%EB%AA%A8%EB%93%88#서브모듈이-있는-프로젝트-Clone하기)

서브모듈을 사용하는 프로젝트를 Clone하면 해당 서브모듈 디렉토리는 빈 디렉터리다:

$ git clone git://github.com/schacon/myproject.git

Initialized empty Git repository in /opt/myproject/.git/

remote: Counting objects: 6, done.

remote: Compressing objects: 100% (4/4), done.

remote: Total 6 (delta 0), reused 0 (delta 0)

Receiving objects: 100% (6/6), done.

$ cd myproject

$ ls -l

total 8

-rw-r--r-- 1 schacon admin 3 Apr 9 09:11 README

drwxr-xr-x 2 schacon admin 68 Apr 9 09:11 rack

$ ls rack/

$

분명히 rack 디렉토리가 있지만 비워져 있다. 먼저 git submodule init 명령으로 서브모듈을 초기화하고 git submodule update 명령으로 서버에서 데이터를 가져온다. 데이터를 전부 가져오면 슈퍼프로젝트에 저장된 커밋으로 Checkout된다:

$ git submodule init

Submodule 'rack' (git://github.com/chneukirchen/rack.git) registered for path 'rack'

$ git submodule update

Initialized empty Git repository in /opt/myproject/rack/.git/

remote: Counting objects: 3181, done.

remote: Compressing objects: 100% (1534/1534), done.

remote: Total 3181 (delta 1951), reused 2623 (delta 1603)

Receiving objects: 100% (3181/3181), 675.42 KiB | 173 KiB/s, done.

Resolving deltas: 100% (1951/1951), done.

Submodule path 'rack': checked out '08d709f78b8c5b0fbeb7821e37fa53e69afcf433'

rack 디렉토리는 이제 복원했다. 그리고 누군가 rack을 수정하면 그 코드를 가져다 Merge한다:

$ git merge origin/master

Updating 0550271..85a3eee

Fast forward

rack | 2 +-

1 files changed, 1 insertions(+), 1 deletions(-)

[master\*]$ git status

# On branch master

# 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: rack

#

Merge해서 서브모듈의 HEAD 값이 변경됐다. 슈퍼프로젝트가 아는 커밋과 서브모듈의 HEAD가 달라서 아직 워킹 디렉토리의 상태는 깨끗한 상태가 아니다:

$ git diff

diff --git a/rack b/rack

index 6c5e70b..08d709f 160000

--- a/rack

+++ b/rack

@@ -1 +1 @@

-Subproject commit 6c5e70b984a60b3cecd395edd5b48a7575bf58e0

+Subproject commit 08d709f78b8c5b0fbeb7821e37fa53e69afcf433

이럴 때 git submodule update 명령을 실행해서 해결한다:

$ git submodule update

remote: Counting objects: 5, done.

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

remote: Total 3 (delta 1), reused 2 (delta 0)

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

From git@github.com:schacon/rack

08d709f..6c5e70b master -> origin/master

Submodule path 'rack': checked out '6c5e70b984a60b3cecd395edd5b48a7575bf58e0'

서브모듈 프로젝트를 풀할 때마다 git submodule update 명령을 실행해야 한다. 뭔가 속는 것 같지만 잘 된다.

개발자들이 흔히 저지르는 실수로 서브모듈의 코드를 수정하고 나서 서버에 Push하지 않는 경우가 있다. 슈퍼프로젝트는 Push했지만 프로젝트가 아는 커밋은 아직 Push하지 않고 개발자 PC에만 있다. 만약 다른 개발자가 git submodule update를 실행하면 슈퍼프로젝트에 저장된 커밋을 서브모듈 프로젝트에서 찾을 수 없어서 에러가 발생한다:

$ git submodule update

fatal: reference isn’t a tree: 6c5e70b984a60b3cecd395edd5b48a7575bf58e0

Unable to checkout '6c5e70b984a60b3cecd395edd5ba7575bf58e0' in submodule path 'rack'

누가 마지막으로 서브모듈을 수정했는지 확인하고:

$ git log -1 rack

commit 85a3eee996800fcfa91e2119372dd4172bf76678

Author: Scott Chacon <schacon@gmail.com>

Date: Thu Apr 9 09:19:14 2009 -0700

added a submodule reference I will never make public. hahahahaha!

그 개발자에게 이메일을 보내거나 전화를 건다.

[**슈퍼프로젝트**](https://git-scm.com/book/ko/v1/Git-%EB%8F%84%EA%B5%AC-%EC%84%9C%EB%B8%8C%EB%AA%A8%EB%93%88#슈퍼프로젝트)

프로젝트 규모가 크면 CVS나 Subversion에서는 모듈 프로젝트를 간단히 하위 디렉토리로 만들었다. 가끔 Git에서도 이런 Workflow을 사용하려는 개발자들이 있다.

Git에서는 각 하위 디렉토리를 별도의 Git 저장소로 만들어야 한다. 그리고 그 저장소를 포함하는 상위 저장소를 만든다. 슈퍼프로젝트의 태그와 브랜치를 이용해서 각 프로젝트의 관계를 구체적으로 정의할 수 있다는 것은 Git만의 장점이다.

[**서브모듈 사용할 때 주의할 점들**](https://git-scm.com/book/ko/v1/Git-%EB%8F%84%EA%B5%AC-%EC%84%9C%EB%B8%8C%EB%AA%A8%EB%93%88#서브모듈-사용할-때-주의할-점들)

전체적으로 서브모듈은 어렵지 않게 사용할 수 있지만, 서브모듈의 코드를 수정하는 경우에는 주의가 필요하다. git submodule update 명령을 실행시키면 특정 브랜치가 아니라 슈퍼프로젝트에 저장된 커밋을 Checkout해 버린다. 그러면 detached HEAD라고 부르는 상태가 된다. detached HEAD는 HEAD가 브랜치나 태그 같은 간접 레퍼런스를 가리키지 않고 커밋을 가리키는 것을 말한다. 데이터를 잃어 버릴 수도 있기 때문에 일반적으로 detached HEAD 상태는 피해야 한다.

submodule update를 실행하고 나서 별도의 작업용 브랜치를 만들지 않고 서브모듈 코드를 수정하고 커밋한다. 그리고 나중에 커밋한 것을 잊은 채로 슈퍼프로젝트에서 다시 git submodule update를 실행시키면 Git은 아무 말 없이 Checkout해 버린다. 엄밀히 말해서 커밋을 없어진 것은 아니지만 브랜치에 속하지 않는 커밋을 찾아내기란 정말 어렵다.

git checkout -b work 같은 명령으로 작업할 때마다 work 브랜치를 만들면 이 문제를 피할 수 있다. 실수로 submodule update 명령을 실행해 버려서 하던 일을 놓쳐버려도 포인터가 있어서 언제든지 되찾을 수 있다.

그리고 서브모듈이 있는 슈퍼프로젝트의 브랜치를 오갈 때는 약간의 추가작업이 필요하다. 브랜치를 만들고 서브모듈을 추가한다. 그 다음에 서브모듈이 없는 브랜치로 돌아간다. 그렇지만, 이미 추가한 서브모듈 디렉토리가 untracked 상태로 보인다:

$ git checkout -b rack

Switched to a new branch "rack"

$ git submodule add git@github.com:schacon/rack.git rack

Initialized empty Git repository in /opt/myproj/rack/.git/

...

Receiving objects: 100% (3184/3184), 677.42 KiB | 34 KiB/s, done.

Resolving deltas: 100% (1952/1952), done.

$ git commit -am 'added rack submodule'

[rack cc49a69] added rack submodule

2 files changed, 4 insertions(+), 0 deletions(-)

create mode 100644 .gitmodules

create mode 160000 rack

$ git checkout master

Switched to branch "master"

$ git status

# On branch master

# Untracked files:

# (use "git add <file>..." to include in what will be committed)

#

# rack/

서브모듈 디렉토리를 다른 곳에 옮겨 두거나 삭제해야 한다. 삭제할 경우는 원래 브랜치로 돌아왔을 때 서브모듈을 다시 Clone해야 하고, 이 경우 아직 Push하지 않았던 변경사항이나 브랜치를 잃을 수 있다.

rack이라는 디렉토리가 있고 이것을 서브모듈로 바꾸려고 한다고 가정하자. 먼저 rack 디렉토리를 삭제하고submodule add를 실행하면 Git은 아래와 같은 에러를 뱉는다:

$ rm -Rf rack/

$ git submodule add git@github.com:schacon/rack.git rack

'rack' already exists in the index

rack 디렉토리를 Staging Area에서 제거하면 서브모듈을 추가할 수 있다.

$ git rm -r rack

$ git submodule add git@github.com:schacon/rack.git rack

Initialized empty Git repository in /opt/testsub/rack/.git/

remote: Counting objects: 3184, done.

remote: Compressing objects: 100% (1465/1465), done.

remote: Total 3184 (delta 1952), reused 2770 (delta 1675)

Receiving objects: 100% (3184/3184), 677.42 KiB | 88 KiB/s, done.

Resolving deltas: 100% (1952/1952), done.

한 브랜치에서는 해결했다. 아직 해당 디렉토리를 서브모듈로 만들지 않은 브랜치를 Checkout하려고 하면 아래와 같은 에러가 난다:

$ git checkout master

error: Untracked working tree file 'rack/AUTHORS' would be overwritten by merge.

다른 브랜치로 바꾸기 전에 rack 서브모듈 디렉토리를 다른 곳으로 옮겨 둔다:

$ mv rack /tmp/

$ git checkout master

Switched to branch "master"

$ ls

README rack

그리고 나서 다시 서브모듈이 있는 브랜치로 돌아가면 rack 디렉토리는 텅 비어 있다. git submodule update 명령으로 다시 Clone하거나 /tmp/rack/에 복사해둔 파일을 다시 복사한다.

[**git submodule 사용법**](http://bestend.co.kr/283)

z

원문 링크 : http://chrisjean.com/2009/04/20/git-submodules-adding-using-removing-and-updating/

GIT을 사용한지도 한달이 조금 더 되었네요. 솔직히 말하면 GIT을 사용하면서 좋은 점도 많이 있었지만, 개인적으로 사용하면서 불만도 있었던거 같습니다.

특히 서브모듈은 문제가 되는것중 하나였구요. 서브모듈의 개념은 간단하지만, 서브모듈을 사용하여 실질적인 공동작업 하는 방법을 아는것도 하나의 '일'입니다. 서브모듈에 관한 내용이 대체적으로는 문서가 잘 되었지만, 완벽히는 아니기 때문에, 제가 알아낸 두세가지 어려운 팁들을 공유합니다~법

"GIT 저장소에서 서브모듈을 제거하고 업데이트 하기"

**- 서브 모듈이란?**

서브모듈의 컨셉은 훌륭합니다. 본질적으로 하나의 저장소가 또다른 저장소에 들어갈수 있다는 개념이죠. 이러한 서브모듈의 장점을 증명하기위해서, 제가 서브모듈을 어떻게 사용하는지 설명해보겠습니다~

저는 WordPress에서 themes 담당을 맞고 있습니다. 테마에 기능들을 향하는 개발을 맡고 있죠. 저는 이러한 향상된 기능을 가진 모듈을 개발하는데요, 그 모듈들은 각자의 폴더를 다 가지고 있어요. 이 코드들은 다른 테마에 쉽게 추가 될수도 있고, 혹은 업데이트가 될수도 있는데, 이러한 작업은 계속 이루어집니다.

각 테마든 각자의 GIT저장소에 저장이 됩니다. 그리고, 테마에 들어가는 모듈을 직접 테마 저장소에 넣지 않았구요, 모듈별로 따로 저장소를 만들어 두었구요. 간단하게 서브모듈로 모듈들을 테마에 넣었습니다.

예를들면, FlexxBold라는 테마가 있는데, FlexxBold에는 현재 7가지 서브모듈이 추가 되어 있죠 billboard, contact-page-plugin, featured-images, feedburner-widget- file-utility- flexx-layout-editor, and, tutorials. 제가 서브모듈을 사용하기때문에, 각각의 모듈 저장소에서 수동적으로 업데이트를 하는게 아니라, 한번에 pull을 할 수 있었습니다.

제가 위에서 말했듯이, GIT의 모든 작업이 쉬운것이 아닙니다!! 제가 여러분께 알려드릴 서브모듈의 4가지 기능을 소개합니다. (역주 : 왜 한말을 또 할까.. 중략합니다.) 아래를 보세요

**GIT저장소에 서브모듈 추가하기**

다행스럽게, GIT저장소에 서브모듈 추가하는 법은 쉽습니다. 예를 들어보죠, SampleTheme 저장소가 있고, billboard라는 모듈 저장소를 lib/bilboard 디렉토리에 추가한다고 가정해보죠. 그럼 아래와 같이 하면 됩니다.

**[user@office SampleTheme]$ git submodule add git@mygithost:billboard lib/billboard**

**Initialized empty Git repository in ~/git\_dev/SampleTheme/lib/billboard/.git/**

**remote: Counting objects: 1006, done.**

**remote: Compressing objects: 100% (978/978), done.**

**remote: Total 1006 (delta 631), reused 0 (delta 0)**

**Receiving objects: 100% (1006/1006), 408.22 KiB, done.**

**Resolving deltas: 100% (631/631), done.**

위 예제에 3가지 파트를 자세히 살펴보죠

-> git submodule add - 단순히 GIT에게 서브모듈을 추가하는 명령어입니다. 이 구문은 항상 고정입니다

-> git@mygithost:billboard 서브모듈로 지정할 바깥저장소에 대한 정보입니다. 문법은 저장소에따라 달라질수 있구요. 그리고 저장소가 이 기능을 지원하는지 확인해봐야됩니다.

-> lib/billboard - 서브모듈 저장소에 추가될 경로를 지정하는것입니다.

그럼 저장소에서는 어떻게 하고 있는지 체크해보죠.

**[user@office SampleTheme]$ git status**

**# On branch master**

**# Changes to be committed:**

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

**#**

**#       new file:   .gitmodules**

**#       new file:   lib/billboard**

**#**

아까 넣은 경로가 생성되었고, 커밋할 부분이 추가 되었습니다. 또, .gitmodules 라는 파일이 추가 되었구요. 이 파일이 서브모듈에 대한 정보를 담고 있습니다. 자 이제 이 파일을 한번 보죠.

**[user@office SampleTheme]$ cat .gitmodules**

**[submodule "lib/billboard"]**

**path = lib/billboard**

**url = git@mygithost:billboard**

나중에 이 파일을 수정할 수 있다는점 기억하세요.

이제 남은거는 변경된 사항을 commit하시고 remote 저장소에 push하는거네요~

**서브모듈 사용하기**

저장소에 서브모듈을 추가하는것은 좋은 일이지만, 제 저장소를 보면, 폴더를 보면 실제 내용은 없는 빈 폴더네요. 외부 저장소로부터 서브모듈을 채우려면, 먼저 초기화를 하시고 다음에 update를 해야됩니다.

참고 : 위 내용은 GIT의 최신버전에는 적용되지 않습니다. 지금은 그냥 add하면 복사가 됩니다.  참고로 서브모듈 저장소에 또다른 서브모듈이 있는경우, 서브모듈의 서브모듈은 서브모듈 디렉토리에서 다음과 같은 정차를 진행해야됩니다. (해깔리시죠?)

만약에 phone-app 프로젝트 작업을 한다고 가정하고, graphics-lib라는 서브모듈을을 추가하고, render라는 graphics-lib의 서브모듈을 추가 하고 싶으면, graphics-lib를 먼저 add하세요, 그러면 phone-app/graphic-lib 는 복사가 되어 있을겁니다. 그런데 phone-app/graphic-lib/renderer는 역시 비어있을것입니다. phone-app/graphics-lib/renderer폴더를 복사되게 하고싶으시면, phone-app/graphics-lib 디렉토리로 이동하시고, 아래대로 하시면 됩니다. ^^

첫번째, 서브모듈을 초기화합니다. 아래처럼 하세요.

**[user@office SampleTheme]$ git submodule init**

**Submodule 'lib/billboard' (git@mygithost:billboard) registered for path 'lib/billboard'**

그다음, update 명령어를 이용해서 파일들을 불러옵니다.

**[user@office SampleTheme]$ git submodule update**

**Initialized empty Git repository in ~/git\_dev/SampleTheme/lib/billboard/.git/**

**remote: Counting objects: 26, done.**

**remote: Compressing objects: 100% (22/22), done.**

**remote: Total 26 (delta 5), reused 0 (delta 0)**

**Receiving objects: 100% (26/26), 17.37 KiB, done.**

**Resolving deltas: 100% (5/5), done.**

**Submodule path 'lib/billboard': checked out '1c407cb2315z0847facb57d79d680f88ca004332'**

이제 lib/billboard 디렉토리를 보시면, 파일들이 채워져 있는것을 보실수 있게 될겁니다!

**서브모듈 제거하기**

서브모듈을 제거해야 한다면 어떻게 하실건가요? 서브모듈에 치명적인 코드가 담겨서, 전체적으로 복구 할수가 없고 새로 서브모듈을 만들어야 한다면 어떻게 하시겠습니다. 걱정마세요!! (역주 : 무슨 광고 같네요 ) 그냥 "git submodule rm [submodule path]" 를 치시면 됩니다. 그게 다입니다. 진짜라니깐요.

**[user@office SampleTheme]$ git submodule rm lib/billboard**

**error: pathspec 'rm' did not match any file(s) known to git.**

**Did you forget to 'git add'?**

**b8ff8f68eb56938b9b4bf993619218fa848c5848 lib/billboard (1.2.25)**

불행히도, 위의 명령은 틀렸네요. (역주 : 얘가 장난친거네요) GIT에는 내장된 서브모듈을 제거하는 명령이 없습니다. 하지만 걱정마세요. 제가 서브모듈을 수동으로 제거하는 방법을 알려드릴테니깐요.

SampleTheme에 lib/billboard 서브모듈을 제거한다고 가정하겠습니다. 모든 명령어는 SampleTheme저장소 디렉토리에서 실행 될것이구요. 그러기 위해 할 일이 있습니다.

1. 서브모듈의 entry파일인 .gitmodules 파일을 제거하세요. lib/billboard 서브모듈이 저장소에 유일한 서브모듈이기때문에, 그냥 간단히 git rm lib/billboard 하면됩니다. 근데, lib/billboard말고 다른 서브모듈이 또 존재하면, .gitsubmodules file을 제거 하시면 안되고, 수동으로 고쳐야됩니다. vi혹은 좋아하는 에디터로 파일 여시고요, 제거할 모듈 내용이 담긴 세줄을 지워주세요. 이경우애는 아래 내용을 지우면 되겠네요.

[submodule "lib/billboard"]

path = lib/billboard

url = git@mygithost:billboard

2. .git/config의 서브모듈 관련 애용도 지워야됩니다. 엄격히 말하면 이거 안하셔도 상관없는데, 저장소를 깔끔하게 유지하고싶고, 미래에 닥칠 문제(?)에 대비하고 싶으시면 하시는게 좋습니다. .git/config의 서브모듈관련 entry는 "git submodule init"을 했을때만 들어간다는거 참고하시구요. 없으면 무시하셔도 됩니다. 아까의 예제에서는 아래 라인들을 지우면 되겠군요.

[submodule "billboard"]

url = git@mygithost:billboard

3. 마지막으로 서브모듈에서 만들어졌던 디렉토리를 제거하시면됩니다. 쉽죠. 그냥 "git rm -cached [plugin path]" 하시면 되구요. 예제의 경우에는 "git rm -cached lib/billboard" 하면 되겠네요. 아시겠지만, 마지막 슬레시는 꼭 빼주셔야됩니다. 예제의 경우에서 "git rm -cached lib/billboard/" 라고 치시면 "fatal: pathspec 'lib/billboard/' did not match any files" 라는 에러메시지가 나올겁니다.

**[user@office SampleTheme]$ git rm --cached lib/billboard**

**rm 'lib/billboard'**

**서브모듈 업데이트 하기**

저장소에서 처음에 서브모듈 했을 당시에, 완벽히 구현안되었던 것들이 있을수도 있습니다. 서브모듈을 추가하면, 서브모듈을 가장 최근 commit이 main 저장소의 index에 저장됩니다. 이 의미는, 서브모듈의 저장소가 업데이트되면, 새로운 코드역시 서브모듈 저장소에서 pull 될것입니다.

테스트 할때나, 검수 넣을때 조심해야됩니다. 서브모듈이 업데이트 되면, 메인 저장소 코드랑 호환이 안될수도 있다는거 잊지마세요.

서브모듈 제거와 마찬가지로 불행하게도, git은 나중 commit에 대한 뚜렷한 대책을 세워놓지 않았습니다. 다행히, 그 대책이 어려운건 아닙니다.

1. 서브모듈을 "git submodule init"으로 초기화 합니다. 그 다음 "git submodule update"를 합니다.

**[user@office SampleTheme]$ git submodule init**

**Submodule 'lib/billboard' (git@mygithost:billboard) registered for path 'lib/billboard'**

**[user@office SampleTheme]$ git submodule update**

**Initialized empty Git repository in ~/git\_dev/SampleTheme/lib/billboard/.git/**

**remote: Counting objects: 26, done.**

**remote: Compressing objects: 100% (22/22), done.**

**remote: Total 26 (delta 5), reused 0 (delta 0)**

**Receiving objects: 100% (26/26), 17.37 KiB, done.**

**Resolving deltas: 100% (5/5), done.**

**Submodule path 'lib/billboard': checked out '1c407cb2315z0847facb57d79d680f88ca004332'**

2. 서브모듈 디렉토리로 이동합니다. 이경우에는 "cd lib/billboard"하면 되겠군요.

**[user@office SampleTheme]$ cd lib/billboard**

**[user@office SampleTheme/lib/billboard]$**

3. "git submodule update"로 추가된 서브모듈의 저장소들은 "headless" 상태입니다. 무슨말이냐면요. 현재 branch에 존재하지 않는다는 거죠. 이걸 고칠려면, branch switch를 써야됩니다. 이경우에는, "git checkout master"라고 입력합니다.

**[user@office SampleTheme/lib/billboard]$ git status**

**# Not currently on any branch.**

**nothing to commit (working directory clean)**

**[user@office SampleTheme/lib/billboard]$ git checkout master**

**Previous HEAD position was b8ff8f6... re-ordering**

**Switched to branch 'master'**

**Your branch is behind 'origin/master' by 8 commits, and can be fast-forwarded.**

**[user@office SampleTheme/lib/billboard]$ git status**

**# On branch master**

**# Your branch is behind 'origin/master' by 8 commits, and can be fast-forwarded.**

**#**

**nothing to commit (working directory clean)**

4. 다음에는 그냥 저장소를 최신상태로 update하면됩니다. 그냥 "git pull"하면 됩니다.

**[user@office SampleTheme/lib/billboard]$ git pull**

**remote: Counting objects: 31, done.**

**remote: Compressing objects: 100% (24/24), done.**

**remote: Total 24 (delta 15), reused 0 (delta 0)**

**Unpacking objects: 100% (24/24), done.**

**From mygithost:billboard**

**b8ff8f6..5cab93f  master     -> origin/master**

**\* [new tag]         1.2.28     -> 1.2.28**

**From mygithost:billboard**

**\* [new tag]         1.2.26     -> 1.2.26**

**\* [new tag]         1.2.27     -> 1.2.27**

**Updating c547e0d..5cab93f**

**Fast-forward**

**billboard.php                           |  109 ++++++++++++++-**

**classes/view\_gettingstarted.php         |  107 ++++++++++++++**

**classes/view\_gettingstarted\_content.php |   38 +++++**

**css/admin.css                           |   26 ++++**

**history.txt                             |   22 +++-**

**js/admin.js                             |   17 +++**

**lib/updater/get.php                     |   94 +++++++-----**

**lib/updater/history.txt                 |    9 ++**

**lib/updater/updater.php                 |  241 ++++++++++++++++++-------------**

**9 files changed, 519 insertions(+), 144 deletions(-)**

**create mode 100644 classes/view\_gettingstarted.php**

**create mode 100644 classes/view\_gettingstarted\_content.php**

**create mode 100644 css/admin.css**

**create mode 100644 js/admin.js**

**create mode 100644 lib/updater/history.txt**

5. 그리고 저장소의 root 디렉토리로 돌아가시면 됩니다. 예제의 경우에는 "cd ../.." 하면 되겠네요.

**[user@office SampleTheme/lib/billboard]$ cd ../..**

**[user@office SampleTheme]$**

6. commit하고 push할 모든 준비가 끝났습니다. (push할 게 있고, remote 저장소가 있다면) "git status"를 입력하시면, 서브모듈의 경로가 수정되었다고 표시되는게 보일겁니다. 이게 우리가 바라던 바죠. 간단하게 경로를 add하고 commit합니다. commit하시면, 서브모듈의 commit 메시지가 update된게 나올것입니다.

**[user@office SampleTheme]$ git status**

**# On branch master**

**# Changed but not updated:**

**#   (use "git add ..." to update what will be committed)**

**#   (use "git checkout -- ..." to discard changes in working directory)**

**#**

**# modified:   lib/billboard (new commits)**

**#**

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

**[user@office SampleTheme]$ git add lib/billboard**

**[user@office SampleTheme]$ git status**

**# On branch master**

**# Changes to be committed:**

**#   (use "git reset HEAD ..." to unstage)**

**#**

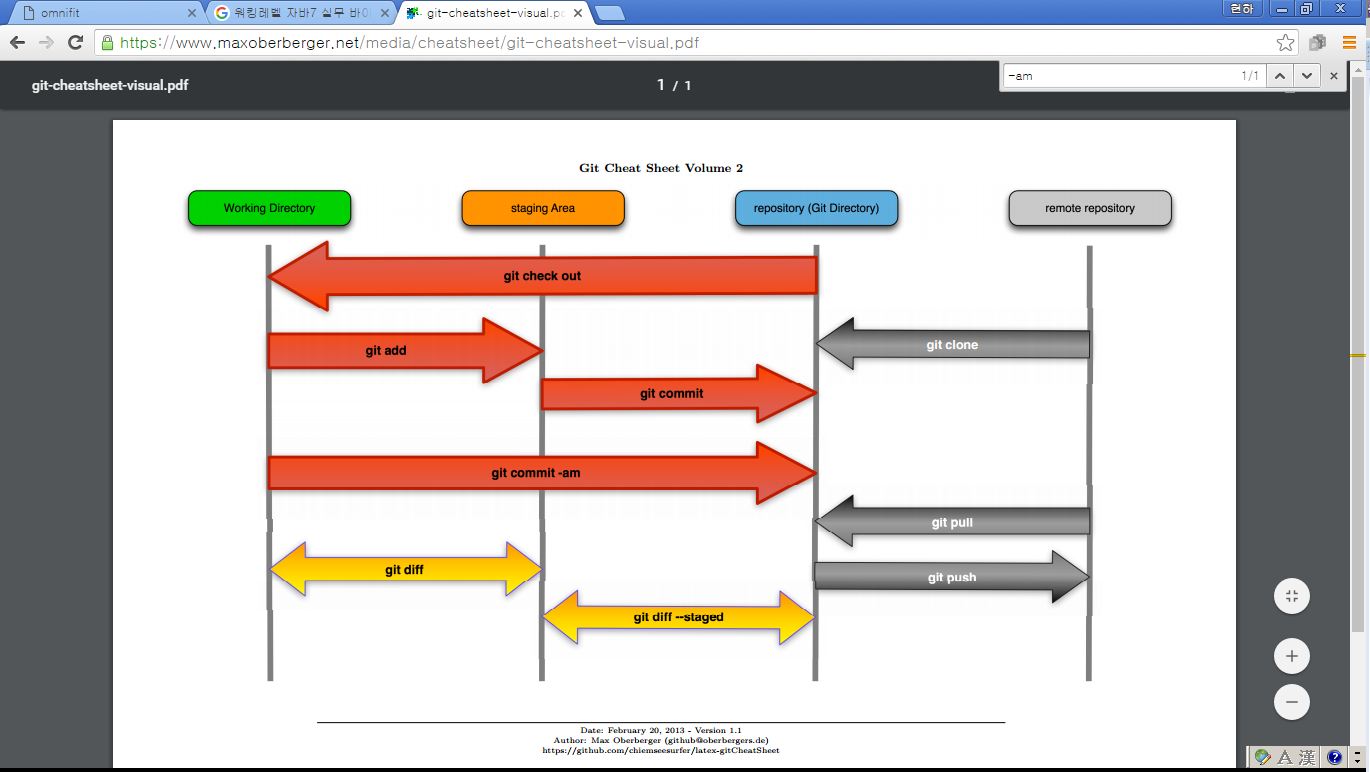
**# modified:   lib/billboard**

**#**

**마치면서...**

짧은 시간이지만 Git에 대해 많이 배운거같습니다. Git사용하는것에 대한 더 자세한 내용을 보길 기대합니다. 이거 포스팅하는데 엄청난 시행착오를 겪었거든요. 그리고 제가 이 내용들을 자동화하는데 필요한 스크립트를 개발했고, 여러분들과 공유하고 싶네요.

수정할 내용이 있거나, 한수 가르쳐주실분은 덧글 남겨주시거나 여기로 (http://chrisjean.com/contact/) 연락주세요. ^^ 감사합니다.



# 2.3 Git의 기초 - 커밋 히스토리 조회하기https://git-scm.com/book/ko/v1/Git%EC%9D%98-%EA%B8%B0%EC%B4%88-%EC%BB%A4%EB%B0%8B-%ED%9E%88%EC%8A%A4%ED%86%A0%EB%A6%AC-%EC%A1%B0%ED%9A%8C%ED%95%98%EA%B8%B0

## 커밋 히스토리 조회하기

새로 저장소를 만들어서 몇 번 커밋을 했을 수도 있고, 커밋 히스토리가 있는 저장소를 Clone했을 수도 있다. 어쨌든 가끔 저장소의 히스토리를 보고 싶을 때가 있다. Git에는 히스토리를 조회하는 명령어인 git log가 있다.

이 예제에서는 simplegit이라는 매우 단순한 프로젝트를 사용한다. simplegit은 Git을 설명하는데 자주 사용하는 예제다. 아래와 같이 이 프로젝트를 Clone한다:

git clone git://github.com/schacon/simplegit-progit.git

이 프로젝트 디렉토리에서 git log 명령을 실행하면 아래와 같이 출력된다:

$ git log

commit ca82a6dff817ec66f44342007202690a93763949

Author: Scott Chacon <schacon@gee-mail.com>

Date: Mon Mar 17 21:52:11 2008 -0700

changed the version number

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7

Author: Scott Chacon <schacon@gee-mail.com>

Date: Sat Mar 15 16:40:33 2008 -0700

removed unnecessary test code

commit a11bef06a3f659402fe7563abf99ad00de2209e6

Author: Scott Chacon <schacon@gee-mail.com>

Date: Sat Mar 15 10:31:28 2008 -0700

first commit

특별한 아규먼트 없이 git log 명령을 실행하면 저장소의 커밋 히스토리를 시간순으로 보여준다. 즉, 가장 최근의 커밋이 가장 먼저 나온다. 그리고 이어서 각 커밋의 SHA-1 체크섬, 저자 이름, 저자 이메일, 커밋한 날짜, 커밋 메시지를 보여준다.

원하는 히스토리를 검색할 수 있도록 git log 명령은 매우 다양한 옵션을 지원한다. 여기에서는 자주 사용하는 옵션을 설명한다.

-p가 가장 유용한 옵션 중 하나다. -p는 각 커밋의 diff 결과를 보여준다. 게다가 -2는 최근 두 개의 결과만 보여주는 옵션이다:

$ git log -p -2

commit ca82a6dff817ec66f44342007202690a93763949

Author: Scott Chacon <schacon@gee-mail.com>

Date: Mon Mar 17 21:52:11 2008 -0700

changed the version number

diff --git a/Rakefile b/Rakefile

index a874b73..8f94139 100644

--- a/Rakefile

+++ b/Rakefile

@@ -5,7 +5,5 @@ require 'rake/gempackagetask'

spec = Gem::Specification.new do |s|

s.name = "simplegit"

- s.version = "0.1.0"

+ s.version = "0.1.1"

s.author = "Scott Chacon"

s.email = "schacon@gee-mail.com"

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7

Author: Scott Chacon <schacon@gee-mail.com>

Date: Sat Mar 15 16:40:33 2008 -0700

removed unnecessary test code

diff --git a/lib/simplegit.rb b/lib/simplegit.rb

index a0a60ae..47c6340 100644

--- a/lib/simplegit.rb

+++ b/lib/simplegit.rb

@@ -18,8 +18,3 @@ class SimpleGit

end

end

-

-if $0 == \_\_FILE\_\_

- git = SimpleGit.new

- puts git.show

-end

\ No newline at end of file

이 옵션은 직접 diff를 실행한 것과 같은 결과를 출력하기 때문에 동료가 무엇을 커밋했는지 리뷰하고 빨리 조회하는데 유용하다.

가끔은 diff 결과를 줄 단위로 보기보다는 단어 단위로 보는 것이 좋을 때도 있다. git log -p와 같은 명령에 --word-diff 옵션을 사용하면 줄 단위 대신 단어 단위로 변경사항을 보여준다. 단어 단위로 다른 부분을 확인하는 것은 소스코드에는 별로 유용하지 않다. 책이나 에세이 같이 문장이 긴 글을 쓸 때는 단어 단위로 보는 것이 편하다. --word-diff 옵션은 다음과 같이 사용한다:

$ git log -U1 --word-diff

commit ca82a6dff817ec66f44342007202690a93763949

Author: Scott Chacon <schacon@gee-mail.com>

Date: Mon Mar 17 21:52:11 2008 -0700

changed the version number

diff --git a/Rakefile b/Rakefile

index a874b73..8f94139 100644

--- a/Rakefile

+++ b/Rakefile

@@ -7,3 +7,3 @@ spec = Gem::Specification.new do |s|

s.name = "simplegit"

s.version = [-"0.1.0"-]{+"0.1.1"+}

s.author = "Scott Chacon"

위의 예제는 줄 단위로 보여주는 일반적인 diff와 좀 다르다. 줄 안에서 변경한 부분을 단어 단위로 표시한다. 추가한 단어는 {+ +} 기호가 둘러싸고 삭제한 단어는 [- -] 기호가 둘러싼다. diff는 기본적으로 다른 줄과 위아래 줄을 포함해서 3줄을 보여준다. 줄 단위가 아니라 단어 단위로 비교해서 볼 때는 굳이 3줄을 다 볼 필요가 없다. 예제에서 처럼 -U1 옵션을 주면 해당 줄만 보여준다.

또 git log 명령에는 히스토리의 통계를 보여주는 옵션도 있다. --stat 옵션으로 각 커밋의 통계 정보를 조회할 수 있다:

$ git log --stat

commit ca82a6dff817ec66f44342007202690a93763949

Author: Scott Chacon <schacon@gee-mail.com>

Date: Mon Mar 17 21:52:11 2008 -0700

changed the version number

Rakefile | 2 +-

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

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7

Author: Scott Chacon <schacon@gee-mail.com>

Date: Sat Mar 15 16:40:33 2008 -0700

removed unnecessary test code

lib/simplegit.rb | 5 -----

1 file changed, 5 deletions(-)

commit a11bef06a3f659402fe7563abf99ad00de2209e6

Author: Scott Chacon <schacon@gee-mail.com>

Date: Sat Mar 15 10:31:28 2008 -0700

first commit

README | 6 ++++++

Rakefile | 23 +++++++++++++++++++++++

lib/simplegit.rb | 25 +++++++++++++++++++++++++

3 files changed, 54 insertions(+)

이 결과에서 --stat 옵션은 어떤 파일이 수정됐는지, 얼마나 많은 파일이 변경됐는지, 또 얼마나 많은 줄을 추가하거나 삭제했는지 보여준다. 요약정보는 가장 뒤쪽에 보여준다.

다른 또 유용한 옵션은 --pretty 옵션이다. 이 옵션을 통해 log의 내용을 보여줄 때 기본 형식 이외에 여러 가지 중에 하나를 선택할 수 있다. oneline 옵션은 각 커밋을 한 줄로 보여준다. 이 옵션은 많은 커밋을 한 번에 조회할 때 유용하다. 추가로 short, full, fuller 옵션도 있는데 이것은 정보를 조금씩 가감해서 보여준다:

$ git log --pretty=oneline

ca82a6dff817ec66f44342007202690a93763949 changed the version number

085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7 removed unnecessary test code

a11bef06a3f659402fe7563abf99ad00de2209e6 first commit

가장 재밌는 옵션은 format 옵션이다. 나만의 포맷으로 결과를 출력하고 싶을 때 사용한다. 특히 결과를 다른 프로그램으로 파싱하고자 할 때 유용하다. 이 옵션을 사용하면 포맷을 정확하게 일치시킬 수 있기 때문에 Git을 새 버전으로 바꿔도 결과 포맷이 바뀌지 않는다:

$ git log --pretty=format:"%h - %an, %ar : %s"

ca82a6d - Scott Chacon, 11 months ago : changed the version number

085bb3b - Scott Chacon, 11 months ago : removed unnecessary test code

a11bef0 - Scott Chacon, 11 months ago : first commit

표 2-1 형식에서 사용하는 유용한 옵션들.

| **Option** | **Description of Output** |
| --- | --- |
| %H | Commit hash |
| %h | Abbreviated commit hash |
| %T | Tree hash |
| %t | Abbreviated tree hash |
| %P | Parent hashes |
| %p | Abbreviated parent hashes |
| %an | Author name |
| %ae | Author e-mail |
| %ad | Author date (format respects the --date= option) |
| %ar | Author date, relative |
| %cn | Committer name |
| %ce | Committer email |
| %cd | Committer date |
| %cr | Committer date, relative |
| %s | Subject |

저자(Author) 와 커미터(Committer) 를 구분하는 것이 조금 이상해 보일 수 있다. 저자는 원래 작업을 수행한 원작자이고 커밋터는 마지막으로 이 작업을 적용한 사람이다. 만약 당신이 어떤 프로젝트에 패치를 보냈고 그 프로젝트의 담당자가 패치를 적용했다면 두 명의 정보를 모두 알 필요가 있다. 그래서 이 경우 당신이 저자고 그 담당자가 커미터다. 5장에서 이 주제에 대해 자세히 다룰 것이다.

oneline과 format 옵션은 --graph 옵션과 함께 사용할 때 더 빛난다. 이 명령은 브랜치와 머지 히스토리를 보여주는 아스키 그래프를 출력한다. 이 명령을 Grit 프로젝트 저장소에서 사용해보면 아래와 같다:

$ git log --pretty=format:"%h %s" --graph

\* 2d3acf9 ignore errors from SIGCHLD on trap

\* 5e3ee11 Merge branch 'master' of git://github.com/dustin/grit

|\

| \* 420eac9 Added a method for getting the current branch.

\* | 30e367c timeout code and tests

\* | 5a09431 add timeout protection to grit

\* | e1193f8 support for heads with slashes in them

|/

\* d6016bc require time for xmlschema

\* 11d191e Merge branch 'defunkt' into local

git log 명령의 기본적인 옵션과 출력물의 형식에 관련된 옵션을 살펴보았다. git log 명령은 앞서 살펴본 것보다 더 많은 옵션을 지원한다. 표 2-2 는 지금 설명한 것과 함께 유용하게 사용할 수 있는 옵션이다. 각 옵션으로 어떻게 log 명령을 제어할 수 있는지 보여준다.

| **옵션** | **설명** |
| --- | --- |
| -p | 각 커밋에 적용된 패치를 보여준다. |
| --word-diff | diff 결과를 단어 단위로 보여준다. |
| --stat | 각 커밋에서 수정된 파일의 통계정보를 보여준다. |
| --shortstat | --stat 명령의 결과 중에서 수정한 파일, 추가된 줄, 삭제된 줄만 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 주전처럼 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 옵션을 함께 사용한 것과 동일하다. |

### [조회 제한조건](https://git-scm.com/book/ko/v1/Git%EC%9D%98-%EA%B8%B0%EC%B4%88-%EC%BB%A4%EB%B0%8B-%ED%9E%88%EC%8A%A4%ED%86%A0%EB%A6%AC-%EC%A1%B0%ED%9A%8C%ED%95%98%EA%B8%B0#조회-제한조건)

출력 형식과 관련된 옵션을 살펴봤지만 git log 명령은 조회 범위를 제한하는 옵션들도 있다. 히스토리 전부가 아니라 부분만 조회한다. 이미 최근 두 개만 조회하는 -2 옵션은 살펴봤다. 실제 사용법은 -<n>이고 n은 최근 n개의 커밋을 의미한다. 사실 이 옵션은 잘 쓰이지 않는다. Git은 기본적으로 출력을 pager류의 프로그램을 거쳐서 내보내므로 한 번에 한 페이지씩 보여준다.

반면 --since나 --until같은 시간을 기준으로 조회하는 옵션은 매우 유용하다. 지난 2주 동안 만들어진 커밋들만 조회하는 명령은 아래와 같다:

$ git log --since=2.weeks

이 옵션은 다양한 형식을 지원한다. 2008-01-15같이 정확한 날짜도 사용할 수 있고 2 years 1 day 3 minutes ago같이 상대적인 기간을 사용할 수도 있다.

또 다른 기준도 있다. --author 옵션으로 저자를 지정하여 검색할 수도 있고 --grep 옵션으로 커밋 메시지에서 키워드를 검색할 수도 있다(author와 grep 옵션을 함께 사용하면 모두 만족하는 커밋을 찾는다).

grep 옵션을 여러개 사용하면 그중 하나이상 만족하는 커밋을 찾는다. 모두 만족하는 커밋을 찾으려면 --all-match 옵션을 추가해야 한다.

마지막으로 파일 경로로 검색하는 옵션이 있는데 이것도 정말 유용하다. 디렉토리나 파일 이름을 사용하여 그 파일이 변경된 log의 결과를 검색할 수 있다. 이 옵션은 --와 함께 경로 이름을 사용하는데 명령어 끝 부분에 쓴다(역주, git log -- path1 path2).

표 2-3은 조회 범위를 제한하는 옵션들이다.

| **옵션** | **설명** |
| --- | --- |
| -(n) | 최근 n 개의 커밋만 조회한다. |
| --since, --after | 명시한 날짜 이후의 커밋만 검색한다. |
| --until, --before | 명시한 날짜 이전의 커밋만 조회한다. |
| --author | 입력한 저자의 커밋만 보여준다. |
| --committer | 입력한 커미터의 커밋만 보여준다. |

아래 예제는 2008년 10월에 Junio Hamano가 커밋한 히스토리를 조회하는 것이다. 그 중에서 테스트 파일을 수정한 커밋 중에서 머지 커밋이 아닌 것들만 조회한다:

$ git log --pretty="%h - %s" --author=gitster --since="2008-10-01" \

--before="2008-11-01" --no-merges -- t/

5610e3b - Fix testcase failure when extended attribute

acd3b9e - Enhance hold\_lock\_file\_for\_{update,append}()

f563754 - demonstrate breakage of detached checkout wi

d1a43f2 - reset --hard/read-tree --reset -u: remove un

51a94af - Fix "checkout --track -b newbranch" on detac

b0ad11e - pull: allow "git pull origin $something:$cur

총 2만여 개의 커밋 히스토리에서 이 명령의 검색 조건에 만족하는 것은 단 6개였다.

### [GUI 도구로 히스토리를 시각화하기](https://git-scm.com/book/ko/v1/Git%EC%9D%98-%EA%B8%B0%EC%B4%88-%EC%BB%A4%EB%B0%8B-%ED%9E%88%EC%8A%A4%ED%86%A0%EB%A6%AC-%EC%A1%B0%ED%9A%8C%ED%95%98%EA%B8%B0#GUI-도구로-히스토리를-시각화하기)

GUI 도구로 커밋 히스토리를 시각화하고 싶다면 gitk를 사용할 수 있다. gitk는 Tcl/Tk 프로그램이고 git log 명령을 시각화해주는 도구다. gitk는 git log 명령이 지원하는 필터링 옵션을 거의 모두 지원한다. 프로젝트 디렉토리에서 gitk를 실행하면 그림 2-2처럼 보일 것이다.



그림 2-2. gitk의 히스토리

위쪽 반을 차지하는 윈도에서는 히스토리를 그래프로 예쁘게 보여준다. 아래쪽 반을 차지하는 윈도는 diff 결과를 보여주는데 위쪽 윈도에서 선택한 커밋에 대한 diff 결과를 보여준다.