|  |
| --- |
| 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  git reset --hard HEAD@{1} |

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

|  |
| --- |
| git merge |
| https://backlog.com/git-tutorial/kr/stepup/stepup7\_7.html  git merge --squash contact : contact의 커밋 두개를 하나로 만들어 master의 staging에 추가함 (git status 확인)  <http://so-blog.net/2016/05/15/git_squash/> squash  * 짓누르다, 납작하게 만들다라는 의미. * Git에서는 이미 수행한 여러개의 커밋들을 한 개의 커밋으로 합치는 행위를 말한다.  How to squash (their commits)상황 git checkout master  git checkout -b branchA  master 브랜치로부터 브랜치 A를 생성했다.  // edit something..  git add .  git commit -m "initial commit"  // edit something..  git add .  git commit -m "add file"  // edit something..  git add .  git commit -m "modify file"  브랜치 A에 3개의 커밋을 했다.  이 때 브랜치 A에 작업했던 커밋 3개를 마스터 브랜치에 머지하고 싶다. 그러나 3개의 커밋을 한 개로 합쳐서 머지하고 싶다. solution 1. git merge –squash git checkout master  git merge --squash branchA solution 2. git rebase -i git checkout branchA  git rebase -i master  그러면 이런 화면이 뜬다.  pick d9eb73f initial commit  pick 1f4c495 add file  pick 02a3484 modify file  # Rebase 6bdf69c..02a3484 onto 6bdf69c (3 command(s))  #  # Commands:  # p, pick = use commit  # r, reword = use commit, but edit the commit message  # e, edit = use commit, but stop for amending  # s, squash = use commit, but meld into previous commit  # f, fixup = like "squash", but discard this commit's log message  # x, exec = run command (the rest of the line) using shell  # d, drop = remove commit  #  # These lines can be re-ordered; they are executed from top to bottom.  #  # If you remove a line here THAT COMMIT WILL BE LOST.  #  # However, if you remove everything, the rebase will be aborted.  #  # Note that empty commits are commented out  맨 윗부분을 아래와 같이 수정하자.  pick d9eb73f initial commit  squash 1f4c495 add file  squash 02a3484 modify file  세 개의 커밋 중 첫 번째 커밋은 그대로 유지하고, 두번째와 세번째 커밋을 pick이 아닌 squash로 변경한다. 합쳐진 커밋의 커밋 메시지는 새로 작성할 수 있다. 만약 첫 번째 커밋의 메시지를 그대로 사용하려면 squash 대신 fixup을 사용하면 된다. How to squash (our commits) #### 상황 브랜치 A에 3개의 커밋을 했다. 마스터 브랜치에 머지하기 전에, 이 커밋들을 한 개로 합쳐버리고 싶다. solution 1. git rebase -i git checkout branchA  git rebase -i HEAD~3  이후에는 위에서 했던 것과 동일하게 rebase를 진행하면 된다. solution 2. git reset 아예 커밋 자체를 되돌리는 방법.  git checkout branchA  git reset --soft HEAD~4  그러면 세 개의 커밋이 모두 취소되고, 그 동안 작업했던 내용은 uncommited 상태가 된다.  git add .  git commit  다시 한번에 커밋.  soft, mixed, hard에 대해서는 [여기](https://git-scm.com/book/ko/v2/Git-%EB%8F%84%EA%B5%AC-Reset-%EB%AA%85%ED%99%95%ED%9E%88-%EC%95%8C%EA%B3%A0-%EA%B0%80%EA%B8%B0)에 아주 잘 나와있다.  만약 이미 remote repository에 push한 커밋에 대해서 변경을 했다면, 그냥 push가 되지 않는다. git push -f와 같이 force 옵션을 사용하여야 한다. |
| git checkout master  git merge contact  마스터에서 contact의 branch를 merge한다. |

|  |
| --- |
| git checkout |
| git checkout –f 모르고 delete한거 체크아웃 다시하기(현재 커밋된거 다시 불러와서 덮어쒸우기) |
| git checkout origin/master ./Hibernate\_khh.docx  리모트에있는걸로 다시받아오기 |
| **git checkout -b dev**  **git checkout -b dev origin/dev** |
| **git checkout -b 사전예약추가 origin/사전예약추가** |
| **git checkout -t origin/eventsource처럼 생성하면 원격저장소의 이름을 그대로 사용합니다.** |

|  |
| --- |
| branch |
| 리스트보기 : git branch  브런치별 마지막 커밋내용 : git branch –v  브런치삭제 : branch –d name  강제 삭제 : -D  삭제 git branch -rd origin/bugfix  브런치 생성 : git checkout -b name  브런치 생성 : git branch name  브런치 이동 : git checkout name |
| https://git-scm.com/book/ko/v2/Git-%EB%B8%8C%EB%9E%9C%EC%B9%98-%EB%A6%AC%EB%AA%A8%ED%8A%B8-%EB%B8%8C%EB%9E%9C%EC%B9%98리모트 브랜치 삭제 동료와 협업하기 위해 리모트 브랜치를 만들었다가 작업을 마치고 master 브랜치로 Merge 했다. 협업하는 데 사용했던 그 리모트 브랜치는 이제 더 이상 필요하지 않기에 삭제할 수 있다. git push 명령에 --delete 옵션을 사용하여 리모트 브랜치를 삭제할 수 있다. `serverfix`라는 리모트 브랜치를 삭제하려면 아래와 같이 실행한다.  $ git push origin --delete serverfix  To https://github.com/schacon/simplegit  - [deleted] serverfix  git -c core.quotepath=false -c log.showSignature=false branch -d 금천구\_자사화  git -c core.quotepath=false -c log.showSignature=false push --progress --porcelain origin :금천구\_자사화 |
| **브런치 리모트 브런치 맵핑하기**  git branch --set-upstream my\_branch origin/my\_branch |
| **리모트 사이트에서 브런치를 이미지워졌고 내 로컬에 아직 remote/origin/name 이 남아있을때**  **리모트 브런치와 로컬 브런치랑 싱크 맞춰주는것**  git fetch --all --prune  하면 사라짐 |
| **다른 브랜치의 파일 비교하기**  **git diff --name-status branch1..branch2**  **git difftool branch1:config/environment.rb branch2:config/environment.rb** |
| <http://seorenn.blogspot.kr/2014/04/git.html>  **1. 현재 브랜치와 다른 브랜치끼리의 차이점 비교**   $ git diff BRANCH\_NAME  하지만 좀 내용이 많아서 차이점이 많을 경우 정신붕괴의 위험(...)이 있을 수 있으므로 간단하게 표시하는 방법을 찾아보자. 다행히도 --name-status 라는 옵션이 있다.   $ git diff --name-status BRANCH\_NAME  **2. 파일 머지하기**   $ git checkout -p BRANCH\_NAME FILE\_PATH  checkout 시 -p 옵션을 주면 Patch 형태로 동작한다. 따라서 다른 브랜치의 특정 파일을 현재 브랜치의 파일로 패치 할 수 있다. 다만 **이 기능은 머지가 아니라 패치**이므로 현재 브랜치의 수정 내역과는 관계 없이 해당 파일의 내용으로 변경된다. 만약 수동으로 머지하길 원한다면 이 커맨드 이후 물어오는 화면에서 'e' 키를 눌러서 수동으로 편집하는 방법이 있다.  **3. 파일 복사해오기**   $ git checkout BRANCH\_NAME FILE\_PATH  현재 브랜치에 스테이징 되어 있지 않은 새로운 파일이 다른 브랜치에 있고 이 파일을 복사해 오려면 그냥 -p 옵션을 빼고 checkout 을 하면 된다.  자주 사용 할 만한 기능 같은데 더 단순한 방법이 있을지도 모르겠다. 좀 더 찾아봐야겠다.\_NAME |
| <http://seorenn.blogspot.kr/2014/04/emacs-magit.html>  앞서 [Git를 이용해 다른 브랜치의 일부 파일 복사해오기](http://seorenn.blogspot.kr/2014/04/git.html) 라는 주제의 글을 올렸는데 같은 작업을 Magit을 이용해 좀 더 간단히 할 수 있다.  아래 스크린캐스트는 Magit을 이용해 Git 저장소의 다른 브랜치의 파일 일부를 현재 브랜치로 머지해 오는 작업을 보여준다.  실제 작업에 사용한 단축키는 아래와 같다. (참고로 magit-status  화면에서 쓰이는 단축키이다)  브랜치 변경 **b b** 그냥 checkout 커맨드를 Magit 에서 빠르게 할 수 있다.  브랜치 간 차이점(diff) 비교 **D** 이 후 현재 브랜치 이름을 입력하고 다음에는 비교할 브랜치 이름을 입력한다. (상황에 따라 여러가지 방법으로 응용이 가능할 것이다.)  브랜치 간 차이점 비교(diff) 시 머지하기 **a** 위 스크린 캐스트를 보면 알 수 있지만, diff 목록에서 실제 patch 항목의 내용에 포커스를 주고 a 키를 누르면 자동으로 해당 부분이 머지가 된다. 거기다 Git를 커맨드라인에서 쓰는 것 보다 더 편한게, 다른 브랜치에서 추가된 파일을 선택하고 a를 누르면 자동으로 현재 브랜치에 복사도 된다는 점이다.  Magit은 정말 Git + Magic 이라는 표현이 맞는 것 같다. :-) |

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

|  |
| --- |
| git clone |
| $ git clone git://github.com/schacon/ticgit.git  Cloning into 'ticgit'...  remote: Reusing existing pack: 1857, done.  remote: Total 1857 (delta 0), reused 0 (delta 0)  Receiving objects: 100% (1857/1857), 374.35 KiB | 193.00 KiB/s, done.  Resolving deltas: 100% (772/772), done.  Checking connectivity... done.  $ cd ticgit  $ git remote  origin |

|  |
| --- |
| git show |
| git show --pretty="" --name-only bd61ad98  커밋한 그거의 파일들 |

|  |
| --- |
| git remote |
| git remote add <remote\_name> <url\_of\_open\_source\_project>  git remote add kr http://asdasdasd.kr  git fetch kr  git checkout -b kr\_master kr/master |
| 리모트의 브런치 refresh 하기 갱신하기  git remote update origin --prune |
| url변경  $ git remote -v  # View existing remotes  origin https://github.com/user/repo.git (fetch)  origin https://github.com/user/repo.git (push)  $ git remote set-url origin https://github.com/user/repo2.git  # Change the 'origin' remote's URL |

|  |
| --- |
| git cherry-pick |
| 다른 브랜치의 일부 커밋만 반영하고 싶을 때  보통 프로젝트를 진행할때는 master 브랜치만 릴리즈 하면서 운영을 하게 되는데요, 가끔 중규모/대규모 개편을 하게 되면 브랜치 두개를 동시에 관리를 해야 하는 애로사항이 있습니다.  # git branch  master  next-release  이렇게 두개의 브랜치가 있습니다. master는 지금 운영중인 브랜치고, next-release는 다음 대규모 개편때 반영될 코드죠. 근데 갑자기 (예를들어)UI상의 오타가 발견되었어요. 이런 경우 master와 next-release에 모두 반영되어야 하죠.  # git checkout master  # git commit -am "fixed: typo"  # git log --pretty=oneline  b14b975 fixed: typo  9f57292 ....  ....  이렇게 master 에 반영이 되었습니다. 이 내용을 next-release에 반영하려면 다음과 같이 하면 됩니다  # git checkout next-release  # git cherry-pick b14b975  # git log --pretty=oneline  23fa1e76 fixed: typo  dd0f27c ...  ...  참고: git flow, 즉 feature/BTS-### 같은 브랜치를 따서 merge 하는 방식으로 운영하는 경우, 굳이 cherry-pick을 이용할 필요까지는 없습니다. single master branch 위에서 작업하는 방식으로 진행하는 경우에 유용하죠.  자동으로 오토머지 되지 않을경우  visualkhh@KIMGRAM MINGW64 /d/omnicns/project/omnifit/source/omnifit-en (master|CHERRY-PICKING)  이런식으로 CHERRY-PICKING 중이라고 알려준다  git status로 확인해보면  both modified: api/src/main/resources/sqlmap/mappers/api/AI026/AI026\_sql.xml  deleted by us: api/src/test/java/api/test/AI026Test.java  both modified: cms/src/main/java/com/ko/omnicns/omnifit/cms/mnt/MntWaitController.java  both modified: cms/src/main/webapp/WEB-INF/config/springmvc/dispatcher-servlet.xml  both modified: cms/src/main/webapp/WEB-INF/jsp/login.jsp  both modified: common/src/main/java/com/ko/omnicns/omnifit/security/spring/CustomAuthenticationProvider.java  deleted by us: common/src/main/resources/message/message-common\_ko\_KR.properties  처럼 both (conflict) 난거며  delete bu us 면 머지하면서 현재브런치에는 지워졌지만 머지하면서 들어온거라는 말이다.  conflict, delete 해결후  git commit 하면 vi 뜨면서 수정할 commant 입력후 커밋된다. |

|  |
| --- |
| git log |
| log 로그보기 히스토리보기  log –p 파일 파일별 보기  git log master..관심군 --oneline | tail -1  처음 관심군 브런치 시작한..  git log –pertty=format: “%h %s” –graph...등으로 메시지 포맷을 정할수있다.  git log --branches --decorate --graph --oneline  <http://www.haruair.com/blog/2210> |
| 위처럼 하면 타임라인처럼 안나온다 그렇기 때문에 sort를 해줘야하는데..  git log --graph --decorate --branches --date=iso --author-date-order   * “Like git log does by default”—but the default for git log is [reverse chronological order](https://git-scm.com/docs/git-log#_commit_ordering), not --topo-order. On any sufficiently complicated repository (I tried [git.git](https://github.com/git/git) itself), git log, git log --topo-order, git log --date-order, and git log --author-date-order produce four different orders. In fact, the default git log misorders a handful of commits with respect to their parents, while the other three do extra work to avoid that. But the default is otherwise closest to --date-order.   git log --graph --decorate --branches --date=iso --author-date-order --all |
| Log format  git log --pretty=format:"%ad %h by %an, %s" |
| Date format  git log --graph --decorate --branches --date=format:'%Y-%m-%d %H:%M:%S' |
| Formats  log --graph --abbrev-commit --decorate --format=format:'%C(bold blue)%h%C(reset) - %C(bold green)(%ar)%C(reset) %C(white)%s%C(reset) %C(dim white)- %an%C(reset)%C(bold yellow)%d%C(reset)' --all  log --graph --abbrev-commit --decorate --format=format:'%C(bold blue)%h%C(reset) - %C(bold cyan)%aD%C(reset) %C(bold green)(%ar)%C(reset)%C(bold yellow)%d%C(reset)%n'' %C(white)%s%C(reset) %C(dim white)- %an%C(reset)' –all  git log --graph --date-order -C -M --pretty=format:"<%h> %ad [%an] %Cgreen%d%Creset %s" --all --date=short  log --all --graph --pretty=format:'%C(auto)%h%C(auto)%d %s %C(dim white)(%aN, %ar)' |
| [alias]  #quick look at all repo  loggsa = log --color --date-order --graph --oneline --decorate --simplify-by-decoration --all  #quick look at active branch (or refs pointed)  loggs = log --color --date-order --graph --oneline --decorate --simplify-by-decoration  #extend look at all repo  logga = log --color --date-order --graph --oneline --decorate --all  #extend look at active branch  logg = log --color --date-order --graph --oneline --decorate  #Look with date  logda = log --color --date-order --date=local --graph --format=\"%C(auto)%h%Creset %C(blue bold)%ad%Creset %C(auto)%d%Creset %s\" --all  logd = log --color --date-order --date=local --graph --format=\"%C(auto)%h%Creset %C(blue bold)%ad%Creset %C(auto)%d%Creset %s\"  #Look with relative date  logdra = log --color --date-order --graph --format=\"%C(auto)%h%Creset %C(blue bold)%ar%Creset %C(auto)%d%Creset %s\" --all  logdr = log --color --date-order --graph --format=\"%C(auto)%h%Creset %C(blue bold)%ar%Creset %C(auto)%d%Creset %s\"  loga = log --graph --color --decorate --all  # For repos without subject body commits (vim repo, git-svn clones)  logt = log --graph --color --format=\"%C(auto)%h %d %<|(100,trunc) %s\"  logta = log --graph --color --format=\"%C(auto)%h %d %<|(100,trunc) %s\" --all  logtsa = log --graph --color --format=\"%C(auto)%h %d %<|(100,trunc) %s\" --all --simplify-by-decoration  As you can see is almost a keystroke saving aliases, based on:   * --color: clear look * --graph: visualize parents * --date-order: most understandable look at repo * --decorate: who is who * --oneline: Many times all you need to know about a commit * --simplify-by-decoration: basic for a first look (just tags, relevant merges, branches) * --all: saving keystrokes with all alias with and without this option * --date=relative (%ar): Understand activity in repo (sometimes a branch is few commits near master but months ago from him) |

|  |
| --- |
| 가장 최근에 커밋 된 순서로 나열된 Git 분기 목록을 얻으려면 어떻게해야합니까? |
| <https://www.it-swarm.dev/ko/git/%EA%B0%80%EC%9E%A5-%EC%B5%9C%EA%B7%BC%EC%97%90-%EC%BB%A4%EB%B0%8B-%EB%90%9C-%EC%88%9C%EC%84%9C%EB%A1%9C-%EB%82%98%EC%97%B4%EB%90%9C-git-%EB%B6%84%EA%B8%B0-%EB%AA%A9%EB%A1%9D%EC%9D%84-%EC%96%BB%EC%9C%BC%EB%A0%A4%EB%A9%B4-%EC%96%B4%EB%96%BB%EA%B2%8C%ED%95%B4%EC%95%BC%ED%95%A9%EB%8B%88%EA%B9%8C/971874956/> |

|  |
| --- |
| git commit에서 나온 브랜치 찾기 |
| <https://www.it-swarm.dev/ko/git/git-commit%EC%97%90%EC%84%9C-%EB%82%98%EC%98%A8-%EB%B8%8C%EB%9E%9C%EC%B9%98-%EC%B0%BE%EA%B8%B0/969916417/>  git reflog show all 🡪 git log -g --abbrev-commit --pretty=oneline --all |

|  |
| --- |
| reflog |
| 로그보기. |

|  |
| --- |
| 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.  내용을보싶을때에는  git stash show stashname  git stash show -p stashname |

|  |
| --- |
| 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 |

|  |
| --- |
| git rebase |
| master쪽으로 rebase할때  master에서 브런치 따로 따서 작업이 진행안된상태에서 master는 계속 진행된상황에 master쪽으로 HEAD를 맞추고 싶을떄 rebase를 사용한다.  git rebase master  rebase로 커밋합치기. http://so-blog.net/2016/05/15/git\_squash/  --interactive (혹은 -i) 모드용 막강한 옵션들을 보유한 [rebase](http://gitready.com/intermediate/2009/01/31/intro-to-rebase.html) 명령은, 커밋들을 합치고 싶을 때 가장 널리 사용되는 명령 중 하나이다. 이 명령이 하는 일은 작은 커밋들을 결합하여 하나의 큰 커밋으로 만드는 것이며, 만약 그날 한 일을 마무리지으려 하거나 아니면 그냥 서로 다른 변경사항들을 묶고 싶어졌을 때 유용하다. 우리는 이제 이런 일들을 어떻게 쉽게 할 수 있는지 알아볼 것이다.  **주의사항 한마디:** 오직 외부 저장소에 push 되지 않은 커밋에 대해서만 이 기능을 사용하라. 만약 다른 사람들이 당신이 삭제하려고 하는 커밋에 기반을 둔 작업을 하고 있다면, 꽤 많은 충돌이 발생할 것이다. 쉽게 말해 다른 사람과 공유하고 있는 역사를 고쳐쓰지 말라.  그럼 당신이 커밋을 몇번 했고, 그것들을 합쳐서 큰 커밋 하나로 만들고 싶다고 하자. 우리 저장소의 역사는 현재 다음과 같다.  http://ko.gitready.com/images/squash1.png  이 최근 4개의 커밋들은 하나로 합쳐지면 더 적절할 것 같으니, interactive rebasing으로 그걸 해보자:-  $ git rebase -i HEAD~4  pick 01d1124 Adding license  pick 6340aaa Moving license into its own file  pick ebfd367 Jekyll has become self-aware.  pick 30e0ccb Changed the tagline in the binary, too.  # Rebase 60709da..30e0ccb onto 60709da  #  # Commands:  # p, pick = use commit  # e, edit = use commit, but stop for amending  # s, squash = use commit, but meld into previous commit  #  # If you remove a line here THAT COMMIT WILL BE LOST.  # However, if you remove everything, the rebase will be aborted.  #  자, 몇가지 일들이 여기서 일어났다. 우선, 나는 Git에게 HEAD 부터 HEAD~4 까지 지난 네 커밋을 rebase 하라고 시켰다. Git는 이제 무엇을 할 수 있는지에 대한 약간의 설명을 포함한 위의 텍스트와 함께 나를 편집기로 보냈다. 화면에 여러가지 옵션들이 나타났지만, 지금 우리는 그냥 커밋들을 하나로 합치는 것만 할 것이다. 자, 위 파일의 첫 네 줄을 이렇게 바꾸면 된다:  pick 01d1124 Adding license  squash 6340aaa Moving license into its own file  squash ebfd367 Jekyll has become self-aware.  squash 30e0ccb Changed the tagline in the binary, too.  기본적으로 이것은 Git에게 나열된 네 커밋을 하나의 커밋으로 합치도록 시킨다. 일단 저장하면 다음과 같이 또 에디터가 실행된다:  # This is a combination of 4 commits.  # The first commit's message is:  Adding license  # This is the 2nd commit message:  Moving license into its own file  # This is the 3rd commit message:  Jekyll has become self-aware.  # This is the 4th commit message:  Changed the tagline in the binary, too.  # Please enter the commit message for your changes. Lines starting  # with '#' will be ignored, and an empty message aborts the commit.  # Explicit paths specified without -i nor -o; assuming --only paths...  # Not currently on any branch.  # Changes to be committed:  # (use "git reset HEAD <file>..." to unstage)  #  # new file: LICENSE  # modified: README.textile  # modified: Rakefile  # modified: bin/jekyll  #  우리가 여러 커밋들을 합쳤기 때문에, Git은 이 작업과 관련된 커밋들의 메시지로 새 커밋 메시지를 작성할 수 있게 해준다. 메시지를 원하는 대로 수정하고, 저장하고, 종료하라. 그렇게 하면 그 커밋들은 성공적으로 합쳐진다!  Created commit 0fc4eea: Creating license file, and making jekyll self-aware.  4 files changed, 27 insertions(+), 30 deletions(-)  create mode 100644 LICENSE  Successfully rebased and updated refs/heads/master.  그리고 우리가 히스토리를 다시 들여다보면…  http://ko.gitready.com/images/squash2.png  이건 상대적으로 그다지 괴롭지 않다. 만약 rebase 중에 충돌 상태로 들어갔다면 그건 resolve 하기 상당히 쉽고 Git는 그걸 할 수 있도록 안내해 줄 것이다. 기본적으로 질문받은 충돌을 고치고, 그 파일을 git add 하고, git rebase --continue 로 계속 진행하면 된다. 물론 원한다면 git rebase --abort 로 이전 상태로 돌아갈 수 있다. 어떤 이유로 인해 rebase 과정에서 커밋을 잃어버렸다면, [reflog](http://gitready.com/intermediate/2009/02/09/reflog-your-safety-net.html) 를 사용해 되돌릴 수 있다.  여기서 다루지 않은 git rebase -i 의 사용법이 상당히 많다. 혹시 공유하고 싶은 것이 있다면, [부디 그렇게 해달라!](http://ko.gitready.com/submit.html) [GitCasts](http://gitcasts.com) 에도, 이 과정 전반 및 더 복잡한 예제도 다루고 있는 [환상적인 영상](http://gitcasts.com/posts/rebasing) 이 올라와 있다. |
| rebase하는도중 중지하고싶다  git rebase --abort |

|  |
| --- |
| 리모트 저장소에 Push하기 |
| $ git push origin master |
| 리모트쪽 브런치로 연결해주기  git push --set-upstream origin master  git push --set-upstream tp master  강제 밀어넣기  git push origin <your\_branch\_name> --force  git push --force |

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

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

|  |
| --- |
| 리모트 저장소 추가하기 remote add, 리모트 저장소 보기, 리모트 저장소 이름을 바꾸거나 리모트 저장소를 삭제하기 |
| $ git remote  origin  $ git remote add pb git://github.com/paulboone/ticgit.git  git remote update pb --prune  $ git remote -v  origin git://github.com/schacon/ticgit.git  pb git://github.com/paulboone/ticgit.git  ---  git checkout -b pb\_master pb/master  ---리모트 저장소 살펴보기  $ git remote show origin  \* remote origin  URL: git://github.com/schacon/ticgit.git  Remote branch merged with 'git pull' while on branch master  master  Tracked remote branches  master  ticgit  --- 리모트 저장소 이름을 바꾸거나 리모트 저장소를 삭제하기  $ git remote rename pb paul  $ git remote  origin  paul |

|  |
| --- |
| 리모트 저장소를 Pull 하거나 Fetch 하기 |
| $ git fetch [remote-name] |
| git pull origin master |
| 혹시 에러난다면 “-- fatal: refusing to merge unrelated histories”  git pull origin branchname --allow-unrelated-histories  처럼 해보자  이유는 "git merge" used to allow merging two branches that have no common base by default, which led to a brand new history of an existing project created and then get pulled by an unsuspecting maintainer, which allowed an unnecessary parallel history merged into the existing project. The command has been taught not to allow this by default, with an escape hatch "--allow-unrelated-histories" option to be used in a rare event that merges histories of two projects that started their lives independently. |

|  |
| --- |
| untracking 파일 지우기 |
| <https://blog.outsider.ne.kr/1164>  git clean -f  git clean -fd  --dry-run 옵션을 주면 지워질 파일을 미리 확인해볼수 있다. |

|  |
| --- |
| revert 커밋 취소 |
| git revert commitid <--하나만 했을때  git revert commitid commitid commitid <--여러개 했을때  git push |

|  |
| --- |
| config |
| git 한글 파일명 깨짐에 따른 조치  git config --global core.quotepath false |

|  |
| --- |
| submodule |
| 서브모듈 추가  git submodule add http://125.141.209.173:7990/scm/om/common.git common  git submodule add -b <branch> <url> <relative\_path\_4m\_root>  git submodule add -b master http://125.141.209.173:7990/scm/comdev/lib-typescript.git |
| Started by user OMNICNS  [EnvInject] - Loading node environment variables.  Building in workspace /var/lib/jenkins/workspace/omnifit2\_DEV  > git rev-parse --is-inside-work-tree # timeout=10  Fetching changes from the remote Git repository  > git config remote.origin.url <http://jenkins@125.141.209.173:7990/scm/om/api.git> # timeout=10  Fetching upstream changes from <http://jenkins@125.141.209.173:7990/scm/om/api.git>  > git --version # timeout=10  using GIT\_ASKPASS to set credentials jenkins  > git fetch --tags --progress <http://jenkins@125.141.209.173:7990/scm/om/api.git> +refs/heads/\*:refs/remotes/origin/\*  > git rev-parse refs/remotes/origin/master^{commit} # timeout=10  > git rev-parse refs/remotes/origin/origin/master^{commit} # timeout=10  Checking out Revision 969688cd4bb58fcb0f7da0862f62fa110fafdf55 (refs/remotes/origin/master)  Commit message: "logbackconfig"  > git config core.sparsecheckout # timeout=10  > git checkout -f 969688cd4bb58fcb0f7da0862f62fa110fafdf55  > git rev-list 4956ecb8482f0ffd7f8ab55b11c7260cb3250177 # timeout=10  > git remote # timeout=10  > git submodule init # timeout=10  > git submodule sync # timeout=10  > git config --get remote.origin.url # timeout=10  > git submodule init # timeout=10  > git config -f .gitmodules --get-regexp ^submodule\.(.\*)\.url # timeout=10  > git config --get submodule.common.url # timeout=10  > git remote # timeout=10  > git config --get remote.origin.url # timeout=10  > git config -f .gitmodules --get submodule.common.path # timeout=10  using GIT\_ASKPASS to set credentials jenkins  > git submodule update --init --recursive --remote common  [Gradle] - Launching build.  [omnifit2\_DEV] $ /usr/local/gradle/bin/gradle build -x test |
| 서브모듈 재설정  git submodule sync  git submodule update --init --recursive --remote  <https://stackoverflow.com/questions/34890313/deleting-all-git-cached-submodules-from-repository> |

|  |
| --- |
| 다른 리모트에 브런치따서 업하기 |
| git branch gemaster  git remote add ge https://khh@st.omnicns.co.kr/scm/onf/omnifit-ge.git  git branch --set-upstream-to ge/master gemaster –force  git push |

|  |
| --- |
| 리모트에 있는 강제로 pull땡기기 |
| git fetch --all  git reset --hard origin/master  git reset --hard origin/<branch\_name>  또는  git reset --hard HEAD  git pull  <https://stackoverflow.com/questions/1125968/how-do-i-force-git-pull-to-overwrite-local-files> |

서브모듈 멀티프로젝트

# [Git: 서브모듈 이해하기 (git submodule)](http://ohgyun.com/711)

**발생일:** 2016.06.23

**키워드:** git submodule, 깃 서브모듈

**문제:**

Git 에서 서브모듈을 사용하려고 한다.

**해결책:**

서브모듈에 대한 자세한 옵션은 [Pro Git 책의 설명](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)을 보는 게 더 나을 것 같아,

여기선 아래와 같이 튜토리얼 형식으로 설명해보려고 한다.

----------------------------------------------------------------

1. 서브모듈

2. 서브모듈 추가하기

3. 부모 프로젝트에서 자식 프로젝트의 내용 변경하고 업데이트하기

4. 자식 프로젝트에서 수정 후, 부모 프로젝트에 적용하기

5. 서브모듈이 있는 프로젝트 클론하기

6. 변경된 서브모듈 업데이트하기

----------------------------------------------------------------

**1. 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

**3. 자식 프로젝트에서 수정 후, 부모 프로젝트에 적용하기**

이번엔 자식 프로젝트에서 변경한 것을 부모 프로젝트에 적용해보자.

먼저 워크 디렉토리에서 자식 프로젝트를 다른 이름으로 클론한다.

~/mywork

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

~/mywork

$ cd submodule\_test\_child

우리가 위에서 추가했던 커밋이 제대로 있는지 확인해보자.

~/mywork/submodule\_test\_child

$ git log --pretty=short -1

commit af2a90a19766d5ab5ed7d5f59e88403245f99ab1

Author: Ohgyun Ahn <ohgyun@gmail.com>

    Add new commit

이제 새로 커밋을 추가하고 리모트에도 푸시해보자.

~/mywork/submodule\_test\_child

$ git commit --allow-empty -m "Second commit"

[master 4506909] Second commit

~/mywork/submodule\_test\_child

$ git push origin master

이렇게 자식 프로젝트에서 수정한 내용을 리모트에 업데이트했다.

현재 자식 프로젝트의 리모트 리파지터리의 master 브랜치는 4506909 커밋을 가리키고 있다.

이제 다시 부모 프로젝트로 돌아와보자.

~/mywork/submodule\_test\_child

$ cd ../submodule\_test\_parent

부모 프로젝트에 자식 프로젝트에서 수정된 내용을 반영하려면,

대상 디렉토리로 가서 리모트의 변경 사항을 머지하면 된다.

~/mywork/submodule\_test\_parent

$ cd submodule\_test\_child

먼저 현재 서브모듈로 포함되어 있는 자식 프로젝트의 마지막 커밋을 확인해보자.

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

$ git log --pretty=short -1

commit af2a90a19766d5ab5ed7d5f59e88403245f99ab1

Author: Ohgyun Ahn <ohgyun@gmail.com>

    Add new commit

예상대로 아직 리모트에 새로 반영된 커밋(4506909)과 다르다.

어떤 브랜치인지도 확인해보자.

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

$ git branch

\* master

서브모듈을을 add 한 후에 다른 처리를 하지 않았고,

(자식 리파지터리의 깃헙 base 브랜치는 master 라서) master 브랜치로 되어 있다.

리모트의 것을 적용해보자.

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

$ git pull origin master

...

From github.com:ohgyun/submodule\_test\_child

 \* branch            master     -> FETCH\_HEAD

   af2a90a..4506909  master     -> origin/master

Updating af2a90a..4506909

자식 리파지터리를 업데이트했으니, 부모 모듈의 참조도 바뀌었을 것이다.

확인해보자.

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

$ cd ..

~/mywork/submodule\_test\_parent

$ git status

On branch master

Your branch is up-to-date with 'origin/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:   submodule\_test\_child (new commits)

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

예상대로, 서브모듈 파일이 수정되었고, diff 도 확인해보자.

~/mywork/submodule\_test\_parent

$ git diff

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

index af2a90a..4506909 160000

--- a/submodule\_test\_child

+++ b/submodule\_test\_child

@@ -1 +1 @@

-Subproject commit af2a90a19766d5ab5ed7d5f59e88403245f99ab1

+Subproject commit 45069090990ad8c291f5f67b23c0aca83d6a4d6f

부모 프로젝트 참조하고 있는 서브모듈의 커밋 해시도 변경된 걸 볼 수 있다.

변경 사항을 커밋하고 푸시하자.

~/mywork/submodule\_test\_parent

$ git commit -am "Update submodule"

$ git push origin master

**4. 서브모듈이 있는 프로젝트 클론하기**

이번에는 서브모듈이 있는 프로젝트를 클론해보자.

워킹 디렉토리로 이동한 후에,

$ cd ~/mywork

다른 이름으로 부모 리파지터리를 클론해보자.

~/mywork

$ git clone git@github.com:ohgyun/submodule\_test\_parent.git submodule\_test\_parent\_2

~/mywork

$ cd submodule\_test\_parent\_2

이렇게 서브모듈이 포함된 리파지터리를 클론하면,

submodule\_test\_child 디렉토리는 존재하지만 내용은 비어있다.

~/mywork/submodule\_test\_parent\_2

$ ls submodule\_test\_child

먼저, 서브모듈을 가져오려면 먼저 초기화 해야한다.

~/mywork/submodule\_test\_parent\_2

$ git submodule init

Submodule 'submodule\_test\_child' (git@github.com:ohgyun/submodule\_test\_child.git) registered for path 'submodule\_test\_child'

다음으로 서브모듈을 업데이트한다.

~/mywork/submodule\_test\_parent\_2

$ git submodule update

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

...

Submodule path 'submodule\_test\_child': checked out '45069090990ad8c291f5f67b23c0aca83d6a4d6f'

서브모듈을 업데이트한다는 건,

현재 부모 리파지터리의 커밋에서 참조하고 있는 서브모듈의 커밋을,

자식 리파지터리의 리모트에서 체크아웃해온다는 뜻이다.

자식 프로젝트의 디렉토리로 이동해 로그를 확인해보면 대상 커밋으로 채워진 걸 확인할 수 있다.

~/mywork/submodule\_test\_parent\_2

$ cd submodule\_test\_child

~/mywork/submodule\_test\_parent\_2/submodule\_test\_child

$ git log --pretty=short -1

commit 45069090990ad8c291f5f67b23c0aca83d6a4d6f

Author: Ohgyun Ahn <ohgyun@gmail.com>

    Second commit

자식 프로젝트의 브랜치도 확인해보자.

~/mywork/submodule\_test\_parent\_2/submodule\_test\_child

$ git branch

\* (detached from 4506909)

  master

브랜치를 보면 detached 상태로 되어 있다.

부모 프로젝트에서 서브모듈을 업데이트하면, 브랜치나 태그 같은 간접 레퍼런스를 가져오는 게 아니라,

저장된 특정 커밋을 체크아웃하기 때문이다.

이 상태에서 2번 튜토리얼처럼 자식 브랜치를 수정하려고 한다면,

아래처럼 별도의 브랜치를 따서 작업하기를 권장한다.

~/mywork/submodule\_test\_parent\_2/submodule\_test\_child

$ git branch -b feature/fix-submodule

**5. 변경된 서브모듈 업데이트하기**

만약, 다른 개발자가 서브모듈을 변경하고 변경 내역을 부모 리파지터리에 적용해 푸시했다고 가정해보자.

이제 우리는 변경된 내용을 업데이트 받으면 된다.

먼저 부모 리파지터리를 pull 해온다.

~/mywork/submodule\_test\_parent

$ cd ~/mywork/submodule\_test\_parent

~/mywork/submodule\_test\_parent

$ git pull origin master

...

Fast-forward

 submodule\_test\_child | 2 +-

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

서브모듈이 변경됐다면, pull 받았을 때 위처럼 submodule\_test\_child 에 대한 참조도 변경되었을 것이다.

하지만 아직 서브모듈은 업데이트 되지 않은 상태다.

pull 받았을 때, 서브모듈이 변경됐다면 항상 아래 명령으로 서브모듈을 업데이트 해주도록 한다.

~/mywork/submodule\_test\_parent

$ git submodule update

**논의:**

- 다시 한 번, git submodule update 는 현재 부모 커밋에서 참조하고 있는 자식 커밋을 체크아웃 한다.

  하위 디렉토리에서 자식 프로젝트에서 작업한 후 리모트에 커밋하지 않고 git submodule update 를 실행하면,

  기존 커밋 대신 부모 리파지터리에서 참조하고 있는 커밋으로 체크아웃되니 주의한다.

- 위 경우,  커밋하지 않은 상태였다면 기존 작업본은 삭제된다.

  커밋만 하고 푸시하지 않은 상태라면, 로그를 뒤져 로컬 커밋을 찾아가는 방법이 있지만 브랜치를 별도로 따두지 않았다면 찾기 번거롭다.

  부모 프로젝트에서 자식 프로젝트를 작업하는 경우라면, 특히 별도의 브랜치를 따서 작업하기를 권장한다.

**참고:**

Pro Git: 6.6 Git 도구 - 서브모듈

<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>

**▶ Git 제어**

* git branch new : 새로운 branch 를 만든다. 아무런 내역이 없다.
* git checkout new : 새로 만든 new 브랜치로 이동한다
* git checkout -b alternate master : master를 복제하여 브랜치를 만들고 alternate로 checkout 이동한다
* git branch -m alternate alternative : 브랜치의 명칭을 alternative로 변경한다 (-M 기존 같은 명칭 브랜치 있어도 덮어씀)
* git brranch -d alternative : alternative 브랜치를 삭제한다 (-D 옵션은 강제 삭제)

**▶ Merge 전략**

* Straight Merge (바로 합치기) : 브랜치 변경 이력 전체를 합치는 방법
* Squashed Commit (커밋 합치기) : 한 브랜치 이력 압축하여 다른 브랜치의 최신 커밋으로 하나 만드는 방법
* Cherry-picking (선택하여 합치기) : 다른 브랜치의 하나의 커밋을 현재 브랜치에 적용하는 방법

각 합치기에 대해서 위->아래 명령순으로 살펴보자

**> 바로 합치기**

* git checkout -b alternate master
* git add about.html (내용 넣음) -> git commit -m "add about"
* git checkout master
* git merge alternate : master 브랜치에 alternate 브랜치를 합친다 (즉, about.html 파일이 신규로 합쳐짐)

**> 커밋 합치기**

* git checkout -b contact master
* git add contact.html (주소 내역 추가) -> git commit -m "add contact address"
* contact.html (email 내역 추가) -> git commit -a -m "add contact email" (즉, commit object 두개가 된다)
* git checkout master
* git merge --squash contact : contact의 커밋 두개를 하나로 만들어 master의 staging에 추가함 (git status 확인)
* git commit -m "add contact address" -m "add contact email"  (master 브랜치에 contact.html 을 커밋한다)

**> 선택하여 합치기**

* git checkout contact
* git commit -a -m "add contact sns id" (contact.html안에 twitter 계정추가 commit object Hash 코드를 34abcd 로 가정함)
* git checkout master
* git cherry-pick 34abcd  ( contact 브랜치의 34abcd 커밋을 master 브랜치에 커밋한다)
* git reset --hard HEAD 마지막 커밋을 master 브랜치에서 제거
* git cherry-pick -n 34abcd ( -n 옵션 사용하면 master 브랜치의 staging에 넣음, 커밋안함. git status 로 확인)
* git commit (최종 커밋함)

**▶ Merge시 충돌 해소하기**

두개의 브랜치를 합치기하다 같은 파일의 내용이 동시에 변경되었을 경우 해결방법을 알아보자.

* git checkout -b about master : about 브랜치 생성 후 checkout
* git add about.html (about 내역 입력) -> git commit -m "add about"
* git branch about2 abut : about 브랜치를 기반하여 about2 브랜치를 생성
* git commit -a -m "add XXX" : about 브랜치의 about.html 파일에 XXX 입력
* git checkout about2
* git commit -a -m "add YYY" : about2 브랜치의 about.html 파일에 YYY 입력 (이제 about 브랜치의 about.html과 about2 브랜치의 about.html 파일에 XXX 와 YYY 충돌이 발생함)
* git checkout about
* git merge about2 : about2 브랜치를 about 브랜치로 바로 합치기
* about.html 안에 conflict 내역이 표시됨 : 내역을 직접 수정하거나, git mergetool 명령으로 merge.tool 값 확인하고 도구로 합치기 시도함 (도구 사용법은 다시 상세히 봐야겠음)

출처: <http://mobicon.tistory.com/106> [Mobile Convergence]

**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

# 깃(Git) 유용한 팁

* [**Git**](https://tech.10000lab.xyz/archive.html?tag=Git)
* [**git guide**](https://tech.10000lab.xyz/archive.html?tag=git+guide)
* [**git tips**](https://tech.10000lab.xyz/archive.html?tag=git+tips)
* Nov 27, 2018

깃은 처음부터 리눅스 오픈 소스를 관리하기 위해 만들어졌습니다. 2005년에 그간 리눅스 소스를 관리하던 툴(BitKeeper)을 공급하던 회사와 관계가 안 좋아 지면서 자체적으로 사용할 수 있는 버전 관리 시스템이 필요하게 되었죠. 리눅스를 처음 만든 리누스 토발즈(Linus Torvalds)도 깃 개발에 적극적으로 참여하게 되고요. 깃 개발의 시작과 과정이 리눅스라는 복잡한 소프트웨어와 수많은 컨트리뷰터를 위한 것이다 보니 여러 상황에 대처할 수 있는 유용한 기능이 많이 있습니다.



[**Designed by Freepik**](https://www.freepik.com/free-vector/helpful-tip-composition-with-flat-design_2560701.htm)

깃을 사용하면서 알아두면 큰 자신감을 얻게되는 유용한 팁들을 알아보겠습니다. 관련된 내용은 오픈 소스 팀 채팅 소프트웨어인 [**줄립의 문서**](https://zulip.readthedocs.io/en/latest/git/index.html)에서 많은 부분 참고하였습니다.

## 커밋 내용 수정

### 마지막 커밋 수정

#### 메시지 수정

커밋하고 나서 커밋 메시지를 수정해야할 때는 커밋 명령어에 --amend 옵션을 사용합니다. git commit --amend -m "New Message"명령어를 사용해서 마지막 커밋 메시지를 변경해 보겠습니다. git log로 지금 로그를 확인해 보고 수정 명령어를 실행한 뒤 결과를 확인해 본 예입니다.

$ git log

commit 5e3da875aca86f5f36f543a22998e0df517328eb (HEAD -> new\_feature)

Author: Ken You <ken@10000lab.xyz>

Date: Sun Nov 25 22:04:05 2018 +1100

order: Add update function.

$ git commit --amend -m "New Message"

[new\_feature bf37424] New Message

Date: Sun Nov 25 22:04:05 2018 +1100

1 file changed, 3 insertions(+)

$ git log

commit bf37424a4662a34e8ddfc63c993bcf0008e6cc2b (HEAD -> new\_feature)

Author: Ken You <ken@10000lab.xyz>

Date: Sun Nov 25 22:04:05 2018 +1100

New Message

#### 마지막 커밋 수정

이상하게도 꼭 커밋을 하고 나면 바로 버그가 하나 튀어 나옵니다. 수정은 항상 단 한 줄로 해결이 되고요. 다시 커밋을 해야하는데, 그냥 이전 커밋에 포함이 되었더라면 좋았을 것을 하는 생각이 들죠. 이때는  git commit --amend라고 명령어를 실행하면 됩니다.

1. 수정이 필요한 파일을 수정합니다.
2. git add 명령어로 파일을 추가합니다.
3. git commit --amend를 실행합니다.

### 지나간 커밋 수정

인터렉티브 깃 리베이스 명령어를 사용해서 수정을 할 수 있습니다. [**‘깃 리베이스 사용하기’**](https://tech.10000lab.xyz/git/git-rebase-workflow.html)에서도 일부 다루었던 기능입니다. git rebase -i HEAD~n 명령어를 사용하게 됩니다. 여기서 n은 수정할 커밋을 포함하는 최근 커밋 갯수입니다. 상황에 맞도록 어떻게 사용하는지 보겠습니다.

#### 커밋 메시지 수정

1. 최근 5개의 커밋을 수정하고자 한다면 git rebase -i HEAD~5라고 실행합니다.
2. 에디터가 열리면서 5개의 커밋이 각각 줄로 표시되면서 pick이라고 첫 단어로 나옵니다. 수정하고자 하는 커밋을 reword라고 변경하고 저장합니다. 그리고 에디터를 종료합니다.
3. 새로 에디터가 열리면서 메시지를 수정하라고 합니다. 메시지를 수정하고 저장합니다.

#### 커밋을 삭제

1. 최근 5개의 커밋에 삭제할 커밋이 포함되어 있다면 git rebase -i HEAD~5라고 실행합니다. 커밋의 갯수에따라 숫자를 변경하면 됩니다.
2. 에디터가 열리면 삭제하고자 하는 커밋의 pick이라고 되어있는 것을 drop이라고 변경하고 저장합니다.

#### 커밋을 합치기(squashing)

깃으로 개발하는 과정에서는 가능하면 자주 커밋을 해서 변경사항을 실수로라도 잃어버리는 일이 없도록 습관을 들이는게 좋습니다. 단지 이러고 나면 커밋 갯수가 너무 많아져서 좀 합쳐서 간단하게 보이고 싶은 경우가 있습니다. 어떤때는 그냥 통째로 하나의 커밋으로 만들고 싶을 때도 있고요.

1. git rebase -i HEAD~n라고 실행합니다. 여기서 n은 합치고자 하는 커밋의 갯수에따라 숫자로 넣으면 됩니다.
2. 에디터가 열리면 보존하고 싶은 커밋은 그대로 두고 합치고 싶은 커밋을 pick대신 squash라고 써 줍니다. 저장합니다.

#### 커밋 순서 변경

1. git rebase -i HEAD~n라고 실행합니다. 여기서 n은 합치고자 하는 커밋의 갯수에따라 숫자로 넣으면 됩니다.
2. 에디터가 열리면 원하는 대로 커밋 순서를 변경합니다. 저장합니다.

### 변경 후 푸시

이미 푸시가 된 커밋 내용을 이렇게 변경하고 나서 다시 푸시를 하려면 에러가 납니다. 이때는 강제로 푸시를 해 줘야하는데요. 푸시하는 브랜치의 이름 앞에 +를 붙여 주면 됩니다. 예를 들어 new\_feature 브랜치를 푸시하려고 한다면 git push origin +new\_feature라고 실행하면 됩니다.

## 변경 내용 확인

### 로컬 변경 사항

파일을 수정하고 아직 스테이징 영역에 넣지 않은 상태에서 변경사항을 보는 경우입니다. 즉 아직 git add 명령어 실행전입니다.

$ git diff

git add를 해서 스테이징 영역에 있는 파일과 커밋되어 있는 내용을 비교하려면 --cached 옵션을 사용합니다.

$ git diff --cached

아직 스테이징 영역에 있지 않은 수정사항과 git add로 스테이징 영역에 들어있는 파일을 마지막 커밋된 내용과 비교할 수 있습니다.

$ git diff HEAD

### 같은 브랜치안에서

현재 브랜치 내에서 두 커밋을 비교할 수 있습니다. 각 커밋의 해쉬(git-ref)를 이름으로 사용하면 됩니다.

마지막 커밋과 그 바로 앞 커밋을 비교하는 경우입니다.

$ git diff HEAD^ HEAD

커밋 해쉬를 이용해서 임의의 두 커밋을 비교할 수 있습니다.

$ git diff 4ac8340 66655af

### 다른 브랜치

두개의 서로 다른 브랜치간의 비교를 하려면 브랜치 이름을 이용합니다.

$ git diff new\_feature master

새 브랜치가 만들어지고 난 후부터 master 브랜치에 변경된 내역을 보려면 아래와 같이 하면 됩니다.

$ git diff new\_feature...master

리모트에 있는 브랜치와도 비교를 할 수 있습니다. origin에 있는 master와 현재 로컬 브랜치를 비교하는 경우입니다.

$ git diff origin/master...master

## 깃 리셋

상당히 위험한 명령어이지만, 개발을 하다보면 한번쯤은 꼭 쓰게되는 명령어입니다. git reset 명령어와 커밋 레퍼런스 해쉬로 원하는 커밋 상태로 리셋을 해 줍니다. 몇가지 옵션을 줄 수 있는데 가장 많이 사용하게 되는 --hard와 --merge 옵션을 사용해 보겠습니다.

### 하드 모드

말 그대로 하드한 리셋입니다. 만일 수정한 소스가 있는데 아직 커밋을 하기 전이라면 해당 수정 사항은 영원히 잃어버리게 되죠. 여기서 커밋 전이라는 것은 스테이지 영역에 있는 것을 포함하고 있으니 각별히 주의를 해야합니다. 예를 들어 보겠습니다. 리뷰 관련 기능을 추가 중입니다. 일부 기능을 수정하면서 2번에 거쳐 커밋을 했습니다. 그리고 또 다른 기능을 수정하고 있었는데 갑자기 새로 구현하고 있는 기능이 필요가 없어졌습니다. 현재 상황을 볼까요.

$ git status

On branch master

Your branch is ahead of 'origin/master' by 2 commits.

(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: review.py

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

$ git reflog

e16690d (HEAD -> master) HEAD@{0}: commit: review: Add delete function

4e4af76 HEAD@{1}: commit: review: Add update function

5dc0734 (origin/master, origin/HEAD) HEAD@{2}: rebase finished: returning to refs/heads/master

git status로 보니 review.py를 수정중이었습니다. git reflog로 보니 최근 2개의 commit을 한게 보입니다. 기능이 필요가 없어졌으니 5dc0734 커밋 상태로 변경을 하면 됩니다. git reset --hard 5dc0734 명령어를 실행해 보겠습니다.

$ git reset --hard 5dc0734

HEAD is now at 5dc0734 Merge branch 'master' of [**https://github.com/saykent/gitwork**](https://github.com/saykent/gitwork)

macmini:gitwork ken$ git reflog

5dc0734 (HEAD -> master, origin/master, origin/HEAD) HEAD@{0}: reset: moving to 5dc0734

e16690d HEAD@{1}: commit: review: Add delete function

4e4af76 HEAD@{2}: commit: review: Add update function

5dc0734 (HEAD -> master, origin/master, origin/HEAD) HEAD@{3}: rebase finished: returning to refs/heads/master

$ git log

commit 5dc0734137b617428f5a8ee25ceb826b9c5cd2b4 (HEAD -> master, origin/master, origin/HEAD)

Merge: 6dbc75c c0180e7

Author: Ken You <ken@10000lab.xyz>

Date: Tue Nov 27 21:58:57 2018 +1100

Merge branch 'master' of [**https://github.com/saykent/gitwork**](https://github.com/saykent/gitwork)

commit 6dbc75c8e6cb11997dc3b277c594b2c8871240ed

Author: Ken You <ken@10000lab.xyz>

Date: Sun Nov 25 22:04:45 2018 +1100

order: Add delete function.

리멧 명령어를 실행하니 HEAD를 옮겼다는 메시지가 나옵니다. git reflog 명령어로 보니까 현재 HEAD가 옮겨졌다는 내용과 이전에 원래 있었던 2개의 커밋이 보입니다. 하지만 실제로 git log를 해보면 2개의 커밋을 사라졌다는 것을 알 수 있습니다. 주의할 점은 앞서 수정 중이던 review.py는 수정했던 내용이 없어지고 리셋이 되었다는 것입니다.

### 머지 모드

하드리셋과 머지 리셋의 다른 점은 커밋하지 않은 수정 내역을 다루는 방식입니다. 앞서 하드 리셋은 커밋하지 않은 작업 내용을 과감하게 리겟해 버리는 효과가 있었습니다. 하지만 머지 리셋은 작업을 했던 파일을 그대로 보존합니다. 그려면 앞서 예에서 보여줬던 review.py 파일의 경우는 어떻게 될까요? 앞의 두 커밋이 모두 review.py 파일을 수정한 것이었기때문에 리셋을 하면 현재 review.py의 수정 내역을 보존 할 수 없습니다. 그래서 에러를 내고 리셋을 하지 않습니다. 혹시 모를 실수를 방지할 수 있겠네요.

$ git status

On branch master

Your branch is ahead of 'origin/master' by 2 commits.

(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: review.py

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

$ git reset --merge 5dc0734

error: Entry 'review.py' not uptodate. Cannot merge.

fatal: Could not reset index file to revision '5dc0734'.

### 커밋 다시 살리기

만일 리셋을 한 상황에서 커밋을 다시 살리고 싶다면 어떻게 할 수 있을까요? 이때는 git cherry-pick 명령어를 사용하면 됩니다. cherry-pick 명령어 뒤에 다시 살리고 싶은 커밋의 해쉬를 적어주면 됩니다. git log를 해 보면 해당 커밋이 다시 살아난 것을 볼 수 있습니다.

$ git cherry-pick 4e4af76

[master d3c00c9] review: Add update function

Date: Tue Nov 27 22:02:40 2018 +1100

1 file changed, 3 insertions(+)

$ git log

commit d3c00c946a0b20128068b14c1a041117c9bdae3f (HEAD -> master)

Author: Ken You <ken@10000lab.xyz>

Date: Tue Nov 27 22:02:40 2018 +1100

review: Add update function

몇개 안되는 명령어이지만 깃을 사용하다보면 이 명령어를 사용할 일이 꼭 생깁니다. 한번씩 연습 삼아서 해 보는 것도 좋습니다. 물론 연습용 깃 리포짓토리를 새로 만들고 말이죠.

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)**

**#**

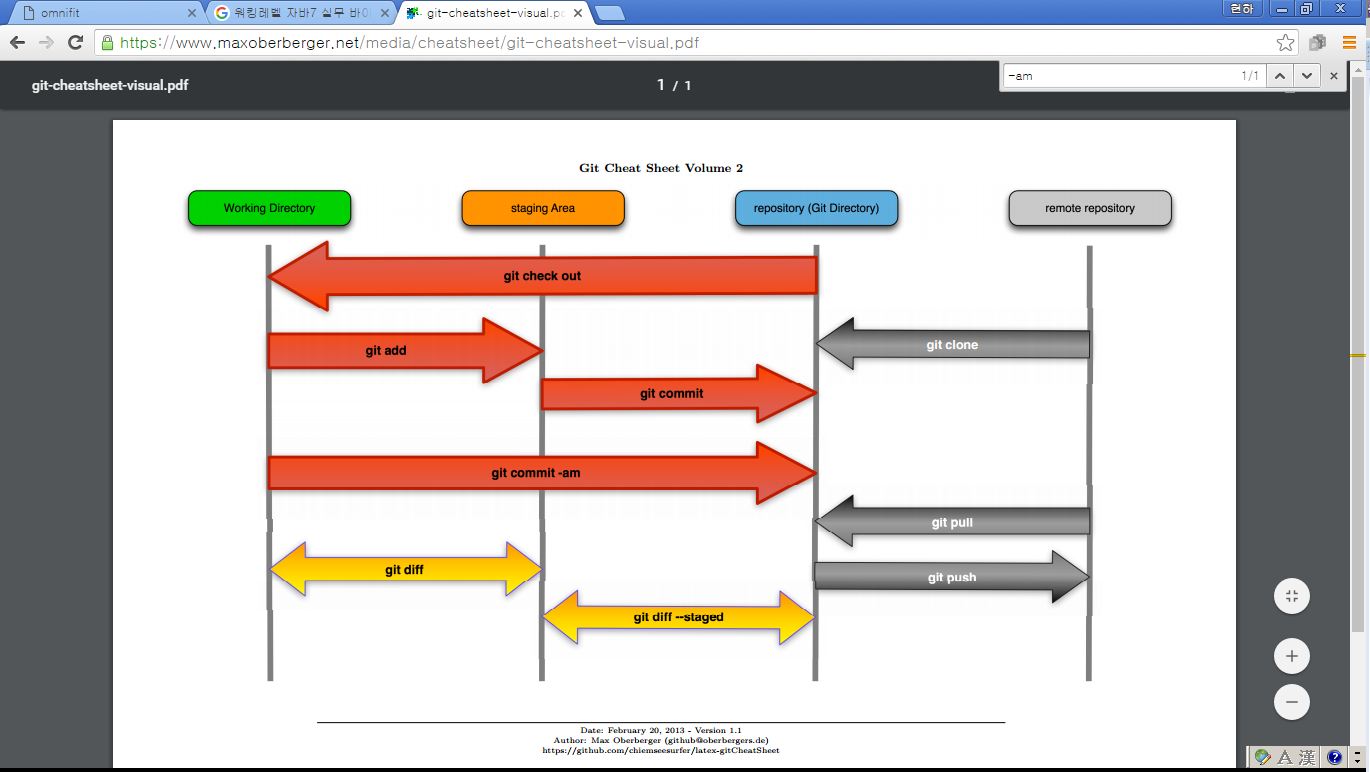
**# modified:   lib/billboard**

**#**

**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.

Git log

# How to Sort Git Commits by Author Date

Post by[*Joseph Purcell*](https://plus.google.com/u/0/118144364575173060176?rel=author)onJanuary 9, 2013

Git commits have both a Commit Date and an Author Date. As Alex Peattie [explains](http://www.alexpeattie.com/blog/working-with-dates-in-git/), the Commit Date is date time when the commit was added to the current branch and Author Date is when the author actually wrote the first commit message. By default, Git orders the git log by the Commit Date, but shows the Author Date in the commit's message.

There are times when you want to order the git log by the Author Date instead of the Commit Date. Long story short, there is no way to do this in Git. However, after scouring the internet I found a number of solutions that worked and didn't work. (Scroll to the bottom for my final solution.)

## Solution 1: Git Log --stdin Flag

Source: [*http://stackoverflow.com/a/13799883*](http://stackoverflow.com/a/13799883)

This solution doesn't work. Git still lists them in order of Commit Date, so I'm not sure what this guy was talking about.

$ git log --pretty="format:%at %H" | sort | sed 's/.\* //' | git log --stdin

## Solution 2: While Loop

Source: [*http://linuxshellaccount.blogspot.com/2008/08/using-bash-to-feed-command-output-to.html*](http://linuxshellaccount.blogspot.com/2008/08/using-bash-to-feed-command-output-to.html)

I was playing around with the idea of writing a for loop to loop through the sorted hashes and run a git log when I came accross the above thread. The result works, but it dumps everything to standard output which you can't pipe to less unless you make it into a script, which is fine, but not simple enough.

$ while read line; do git log -1 $line; echo ""; done < <(git log --pretty="format:%at %H" | sort -r | sed 's/.\* //')

## Solution 3: Sort One Line Output

Next, I decided that I would have to be okay with not having anything more than a one line output. That simplifies things significantly, so I wrote this:

$ git log --date=short --pretty="format:%at %H %ad | %s [%an]" | sort -r | cut -d" " -f2- | less

Note: change the date value of "short" to "iso" or "rfc" to get different date formats.

## Solution 4: Sort Using awk

I know that awk is a powerful tool and you can probably sort a git log using it, but the learning curve would be too high for me. Thus, I skipped awk as a solution.

## Solution 5 (final): Sort One Line Output and Replace Line Endings

Then, I got a brilliant idea: just replace the line endings! The pretty flag has the ability to add line endings using "%n", but the obvious issue is that sort only works on a line-by-line basis. So, I decided to add "\n" as a placeholder and replace it later.

### Version 1.0: Use a Bash Hack for New Line Character

Replacing "\n" with a real line ending is more complex than I could have imagined. After many of my own attempts and reading an absurdly long [thread](http://stackoverflow.com/questions/1251999/sed-how-can-i-replace-a-newline-n) on Stack Overflow about it, I finally found the following solution on an old Linux Questions [thread](http://www.linuxquestions.org/questions/linux-software-2/sed-insert-a-newline-why-does-not-it-work-158806/#post821881) from 2004:

$ git log --pretty="format:%at commit %H\nAuthor: %an <%ae>\nDate: %aD\n\n %s\n" | sort -r | cut -d" " -f2- | sed -e 's/\\n/\

/g'

### Version 2.0: Use Echo for New Line Character

The only problem with the 1.0 solution is that it requires the line break be inserted, thus it is a two-line command. I figured there HAD to be a way to add a silly line return. Fortunately, on that same thread, I saw an interesting [solution](http://www.linuxquestions.org/questions/linux-software-2/sed-insert-a-newline-why-does-not-it-work-158806/#post871948): use echo.

$ git log --pretty="format:%at commit %H\nAuthor: %an <%ae>\nDate: %aD\n\n %s\n" | sort -r | cut -d" " -f2- | sed -e "s/\\\n/\\`echo -e '\n\r'`/g" | less

### Version 3.0: Remove the ^M

The 2.0 solution works great! But, we have a minor issue. Now, less is showing a silly carriage return character in the output. After many failed attempts and much time spent scouring the internet, once again, I found a genius [solution](http://kb.iu.edu/data/acux.html) by Indiana University: tr -d '\15\32'.

$ git log --pretty="format:%at commit %H\nAuthor: %an <%ae>\nDate: %aD\n\n %s\n" | sort -r | cut -d" " -f2- | sed -e "s/\\\n/\\`echo -e '\n\r'`/g" | tr -d '\15\32' | less

### Version 4.0: Add Color

Finally, to complete the script, I had to add color. The convenient part of this is that the pretty flag [supports](http://www.kernel.org/pub/software/scm/git/docs/git-log.html) some default colors, so it was as simple as adding %C(yellow) and %Creset.

$ git log --pretty="format:%at commit %C(yellow)%H%Creset\nAuthor: %an <%ae>\nDate: %aD\n\n %s\n" | sort -r | cut -d" " -f2- | sed -e "s/\\\n/\\`echo -e '\n\r'`/g" | tr -d '\15\32' | less -R

And, notice that we have added the "R" flag to less; this allows less to interpret the color, thanks to a ServerFault [comment](http://serverfault.com/a/26521) for pointing that out.

**UPDATE:**I have added a set of tools, including this one, to the "[Git Extras](https://github.com/josephdpurcell/git-extras)" GitHub repository.

Permalink: <http://thesimplesynthesis.com/post/how-to-sort-git-commits-by-author-date>

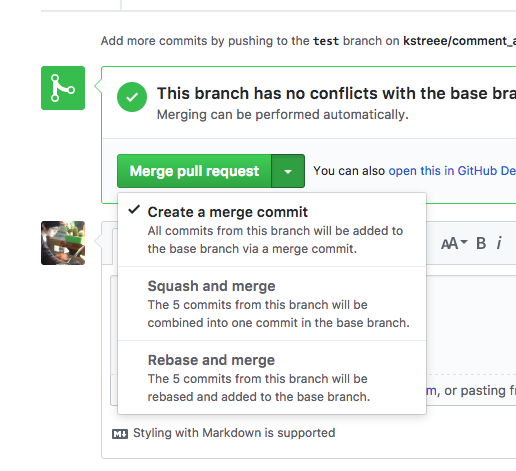
<https://gist.github.com/datagrok/4221767>

# GitHub의 Merge, Squash and Merge, Rebase and Merge 정확히 이해하기

등록일 2017.08.10 조회수 3893



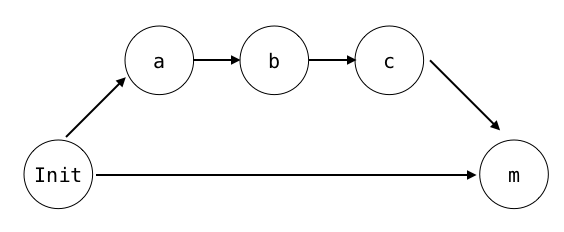
GitHub의 새 버전에서 merge, squash and merge, rebase and merge 세 종류의 merge를 모두 지원하기 시작했습니다. 각 머지 방식에 따라 커밋 히스토리가 천차만별로 달라지는데요, 어떤 경우에 어떤 머지를 사용하는 것이 좋은지 공유하고자 각 머지에 대해 설명해 드리려고 합니다.



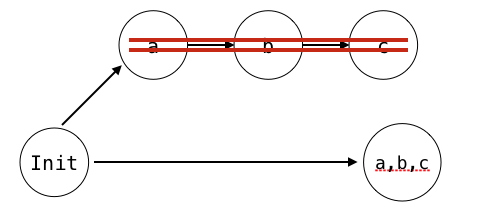
## 그래프로

각 머지를 그래프로 표현하면 아래와 같습니다.

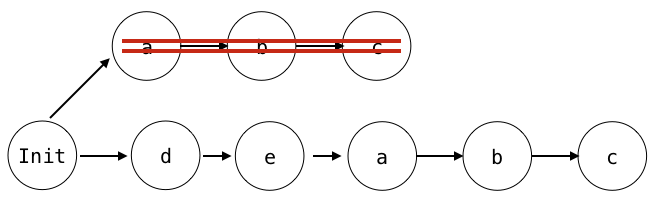
* Merge (a, b, c 를 refer 하는 m 커밋 노드 생성, m은 parent로 Init, c 를 가짐)



* Squash and Merge (a, b, c 를 합쳐서 새로운 커밋으로 만들고, 머지 대상 브렌치에 추가, 'a,b,c' 커밋은 parent를 Init 하나만 가짐.)



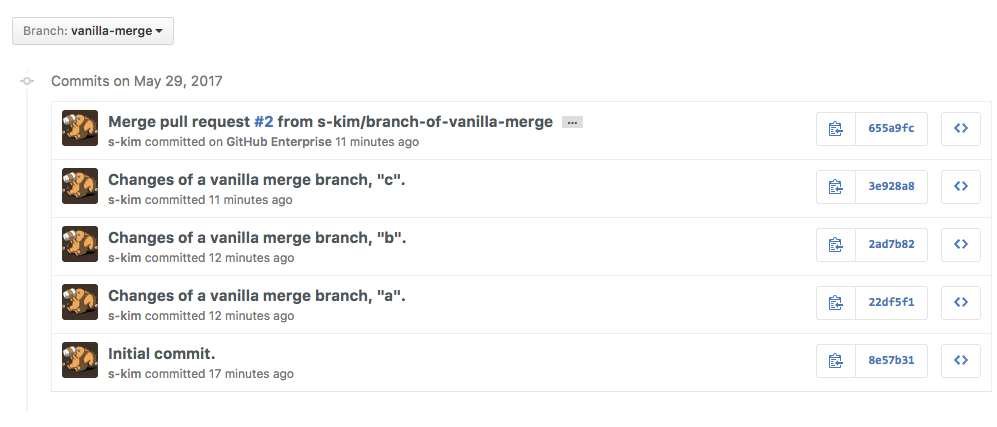
* Rebase and Merge (a, b, c 를 심리스하게 머지 대상 브렌치로 추가, 각 커밋들은 모두 parent를 하나씩만 가짐.)



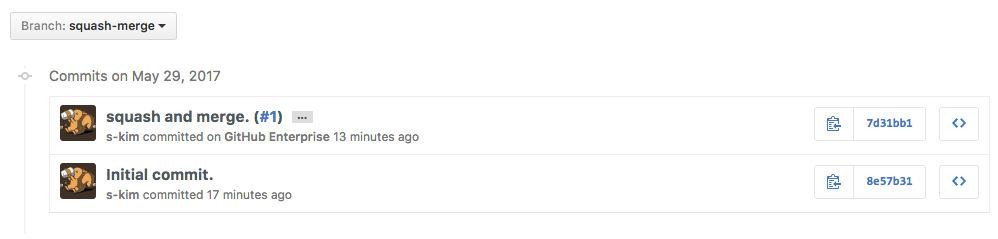
## 메인 브렌치의 관점에서

메인 브렌치의 커밋을 통해 머지된 커밋들이 어떤 형상을 가진지 비교해보면 아래와 같습니다.

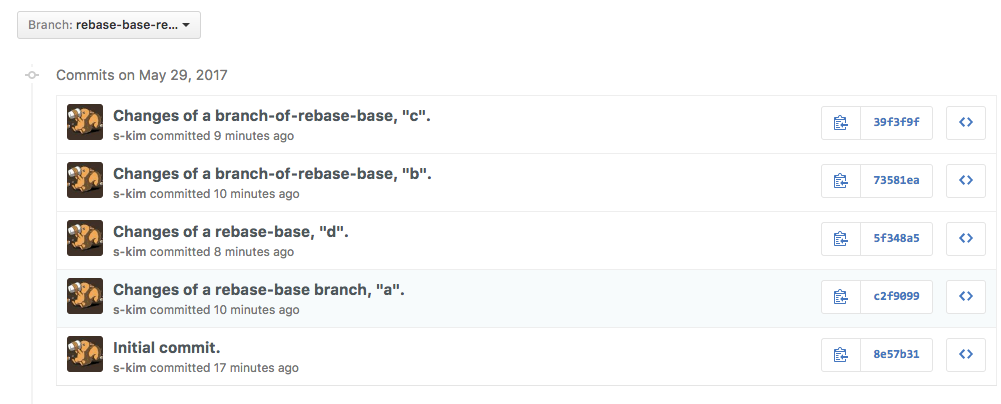
* Merge : 커밋 m에서부터 뒤로 되돌아가면서 부모를 모두 찾아 브렌치를 구성. 커밋 m은 부모로 c, Init을 가지고 있으며, c는 b, b는 a, a는 Init을 다시 부모로 가짐. 이 형상을 모두 backtrace 하여, Init -> a -> b -> c -> m이라는 구조를 만들고 이 구조가 모두 히스토리에 남음.



* Squash and Merge : 커밋 'a,b,c' 는 Init만을 부모로 가진 단일 커밋. 작업했던 브렌치의 a, b, c 커밋들은 머지 후의 메인 브렌치 커밋 Init, 'a,b,c' 와 아무런 연관을 가지지 않음.



* Rebase and Merge : 커밋 a, b, c 의 관계를 그대로 유지한 채, 메인 브렌치에 그대로 추가. 커밋 a는 부모로 커밋 e를 가짐. Rebase and Merge 작업 후에는, 작업했던 브렌치의 a, b, c 커밋들은 머지 후의 메인 브렌치의 Init, d, e, a, b, c 커밋들과 연관 관계를 가지지 않음.



## 사용 예시

Git Flow 를 따른다고 했을 때, 아래와 같이 정리할 수 있습니다.

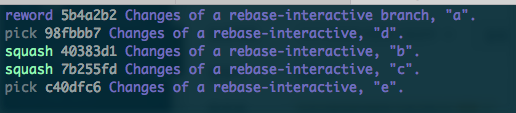
* develop - feature 브렌치간 머지 : Squash and Merge가 유용합니다. feature의 복잡하고 지저분한 커밋 히스토리를 모두 묶어 완전 새로운 커밋으로 develop 브렌치에 추가하여, develop 브렌치에서 독자적으로 관리할 수 있기 때문입니다. 일반적으로 머지 후에 feature 브렌치를 삭제해버리는 점을 떠올려 보면, feature 브렌치의 커밋 히스토리를 모두 develop 브렌치에 직접 연관 지어 남길 필요가 없습니다.
* master - develop 브렌치간 머지 : Rebase and Merge가 유용합니다. develop의 내용을 master에 추가할 때에는 별도의 새로운 커밋을 생성할 이유가 없기 때문입니다.
* hotfix - develop, hotfix - master 브렌치간 머지 : Merge 또는 Squash and Merge 모두 유용합니다. 때에 따라 골라 사용하면 좋을 것 같습니다. hotfix 브렌치 작업의 각 커밋 히스토리가 모두 남아야 하는 경우 Merge, 필요 없는 경우 Squash and Merge를 사용하면 됩니다.

## 보너스

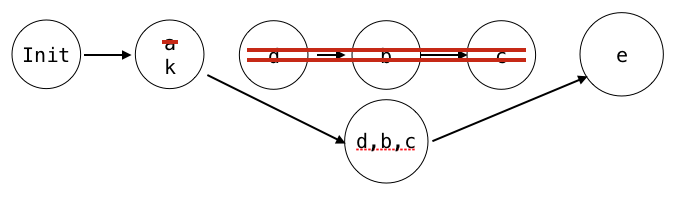
두 브렌치간 Rebase는 위와 같은 관점으로 사용할 수 있지만, cli의 interactive rebase (ex, git rebase -i HEAD~5)를 사용하면 단일 브렌치 내에서 rebase를 사용하여 커밋 히스토리를 정리할 수 있습니다.

* reword 키워드를 사용하면 커밋 메시지를 변경할 수 있습니다.
* squash 키워드를 사용하면 여러 커밋을 하나로 묶을 수 있습니다.

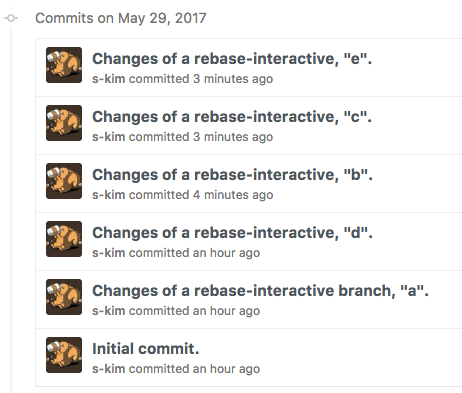
예를 들어, 아래와 같이 cli에서 작업 대상 브렌치에서 git rebase -i HEAD~5를 입력하고, 아래처럼 reword, squash를 적절히 사용하면,



그래프로 표현하면 아래와 같습니다.



기존의 커밋 히스토리인 아래 히스토리가



아래처럼 변경됩니다. (reword, squash 모두 커밋 메시지를 임의로 변경할 수 있습니다. 자세히 적진 않았지만, 아래 reword, squash 작업 후의 커밋 메시지들은 모두 제가 임의로 작성한 것입니다.)



즉, rebase를 다른 브렌치 간의 머지하는 용도 이외에도, 단일 브렌치를 정리하는데에도 사용할 수 있습니다.