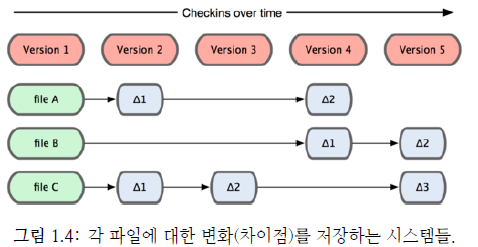
http://msysgit.github.io/

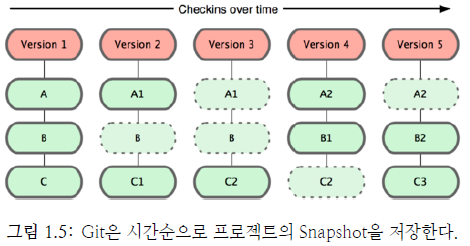
기존 svn과 cvs는 시스템이 관리하는 정보는 파일들의 목록이다.

파일의 집합으로 정보를 관리(파일의 변화를 관리) 한다



하지만 Git은 이런식으로 취급하지 않는다.

Commit하거나 프로제긑의 상태를 저장할때마다 파일이 존재하는 그순간을 중요하게 여긴다



거의모든 명령어를 로컬에서 한다. 로컬자체에서 서버관리들이 이루어진다.

따라서 히스토리등 인터넷이 연결이되지않아도 확인 및 커밋을 할수있다.

Git무결성

Git은 모든 데이터를 저장히기 전에 체크섬 (Checksum또는Hash) 을 구하고 그 체크섬으로 데이터를 관리한다.

체크섬없이는 그어떤 파일이나 디렉토리도 변경할수 없다. 체크섬은 Git에서 사용하는 가장 기본적인 단위인것이다. Git은 SHA-1 Hash를 사용하여 체크섬을 만든다 “24b9as54ds5f5fer8fg4df8g4dsf0awer8”...

Git도 Commit을 하지않으면 변경사항을 잃어버릴수 있다.

하지만 일단 Snapshot을 Commit 하고나면 그 데이터를 잃어버기기는 어렵다.

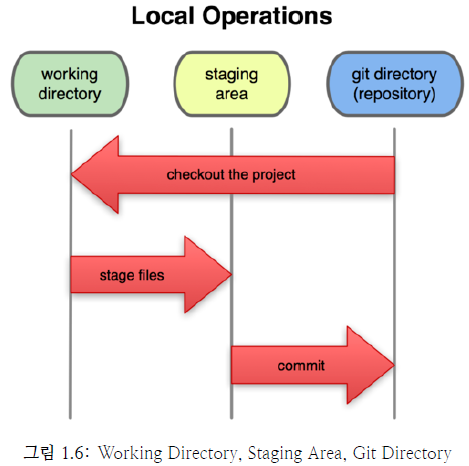
Git의 3가지상태

Commited, Modified, Staged

|  |  |
| --- | --- |
| Commited | 데이터가 로컬 데이터 베이스에 안전하게 저장됐다는것을 의미  **(Git Directory 에있는 파일들은 Committed 상태이다)** |
| Modified | 수정한 파일을 아직 로컬 데이터 베이스에 Commit하지 않은것  **(Checkout하고 나서 수정은했지만 아직 Staging Area에 추가하지않았다면 Modified상태이다)** |
| Staged | 현재 수정한 파일을 곧 Commit할것이라고 표시한 상태.  **(파일을 수정하고 Staging Area에 추가했다면 Staged상태이다)** |

위3가지 상태는 Git 프로젝트의 세가지 단계와 연결돼있다..

Git Directory, Working Directory, Staging Area 이 3가지 단계를 이해해야한다.



|  |  |
| --- | --- |
| Git Directory | Git의 프로젝트의 메타데이터와 객체 데이터 베이스를 저장하는 곳을 말한다.  다른 프로젝트를 Clone할때 Git Directory가 만들어진다. |
| Working Directory | 특정 버전을 Checkout 한것이다.  Git Directory는 지금 작업하는 디스크에 있고 그 Directory에 압축된 데이터베이스에서 파일을 가져와 Working Directory를 만든다. |
| Staging Area | Git Directory에 속하며 단순한 파일이고 곧 Commit할 파일에 대한 정보를 저장한다 종종 인덱스라고 불리기도.. |
| 1. Working Directory에서 파일을 수정한다. 2. Staging Area에 파일을 Stage해서 Commit할 Snapshot을 만든다. 3. Staging Area에 있는 파일들을 Commit해서 Git Directory에 영구적인 Snapshot으로 저장한다. | |

Git 최초설정

git config라는 도구로 설정 내용을 확인하고 변경할수 있다. 설정파일은 3가지나된다.

|  |  |
| --- | --- |
| /etc/gitconfig | 시스템의 모든 사용자와 모든 저장소에 적용되는 설정이다 git config --system 옵션으로 이파일을 읽고 쓸수있다. |
| ~/.gitconfig | 특정 사용자에게만 적용되는 설정이다 git config --global 옵션으로 이 파일을 읽고 쓸수있다. |
| .git/config | 이파일은 Git Directory에 있고 특정 저장소 (혹은 현재 작업중인 프로젝트)에만 적용된다. 역순으로 우선시된다 |

윈도우에서는 Gitdms $HOME Directory c:\\documents and setting\user.. 에있는 .gifconfig파일을 찾는다.

1. Git시작

사용자 정보

Git은 Commit할때 마다 이정보를 사용한다 한번 Commit한후에는 정보를 변경할수 없다.

다시말해 --global 옵션으로 설정한 것은 딱 한 번만 하면된다. 만약 프로젝트마다 다른 이름과 이메일 주소를 사용하고 싶으면 --global 옵션을 빼고 하면된다

|  |
| --- |
|  |
| $ git config --global user.name "visualkhh"  $ git config --global user.email "visualkhh@gmail.com"  $ git config --global user.name  visualkhh  $ git config --global user.email  visualkhh@gmail.com |

편집기

$ git config --global core.editor vi

diff도구 merge툴셋팅

$ git config --global merge.tool vimdiff

설정확인

|  |
| --- |
|  |
| Administrator@ADMIN-PC /C/Program Files/Git/bin  **$ git config --list**  core.symlinks=false  core.autocrlf=true  color.diff=auto  color.status=auto  color.branch=auto  color.interactive=true  pack.packsizelimit=2g  help.format=html  http.sslcainfo=/bin/curl-ca-bundle.crt  sendemail.smtpserver=/bin/msmtp.exe  diff.astextplain.textconv=astextplain  rebase.autosquash=true  user.name=visualkhh  user.email=visualkhh@gmail.com  core.editor=vi  merge.tool=vimdiff |

도움말

git help <verb>

git <verb> --help

man git-<verb>

**Git 저장소만들기**

두가지 방법이 있다

1. 기존에있는 프로젝트를 Git저장소로 만들기

2. 다른서버에 있는 저장소를 Clone 하는것

**1. 이미있는 디렉토리에서 새저장소 만들기**

|  |
| --- |
|  |
| Administrator@ADMIN-PC /D/git/document-visualkhh  $ git init  Initialized empty Git repository in d:/git/document-visualkhh/.git/ |
| 하위디렉토리.git 폴더를 만들고 이것이 저장소를 이루는 뼈대가 된다. |

파일추가 및 커밋

|  |
| --- |
|  |
| $ git add README.md  $ git commit -m "hi visualkhh world" |

**2. 이미있는 저장소를 Clone하기**

$ git clone <https://github.com/visualkhh/python.git>

**수정하고 저장소에 저장하기**

파일을 수정하다가 저장하고 싶으면 Snapshot을 커밋해야한다.

모든 파일은 크게 Tracked(관리대상임)와 Untracked(관리대상아님)로 나뉜다.

|  |  |  |
| --- | --- | --- |
| Snapshot 포함 -> | Tracked 파일 | Unmodified (수정하지않음) |
| Modified (수정함) |
| Staged (커밋을 하면 현재 수정사항을 저장소에 기록함) |
| Snapshot 미포함->  Stage Area 미포함-> | Untracked 파일 |  |
|  |

|  |
| --- |
|  |
| 1. 처음에 프로젝트를 Clone하고 난뒤 모든파일은 Tracked이면서 Unmodified 상태이다.  2. 이상태에서 어떤파일을 수정하게되면 Git은 즉시 Modified상태로 인식한다.  3. 이수정한 파일을 Stage하고  4. Staged 상태인 파일들을 커밋한다. |

**파일 상태 확인하기**

|  |
| --- |
|  |
| $git status  새로운파일 test.txt는 새로만들어졌기때문에 아직 untracked 상태에 속해있다.  수정된파일 README.md는 수정이되어있기때문에 modified 상태에 속해있다. |

**새로운 파일 추적하기**

Untracked -> Tracked 만들기

|  |
| --- |
|  |
| $git add filename |
| add와 동시에 파일이 Staged 상태로 바로 들어간다. |

**수정된 Modified상태의 파일을 Stage하기 -> Staged상태만들기**

|  |
| --- |
|  |
| 파일을 Tracked시킬때와 동일하게  git add 명령어를 통해서 Modified 파일을 Stage하여 Staged 상태로 변경할수 있다. |

**Stage상태 파일을 다시 수정하게되면..?**

|  |
| --- |
|  |
| Staged상태이면서 동시에 Unstaged상태로 돌아온다.? 다시 git add를 통해서 다시 Staged상태로 돌리면  마지막 git add한 상태를 마지막으로 반영한다. |
| ` |
| 마지막으로 git add 한상태를 커밋에 반영한다. |

**Staged 와 Unstaged 상태의 변경 내용보기**

|  |
| --- |
|  |
| git diff  이명령은 작업 디렉토리에 있는 것과 Staging Area에 있는것을 비교한다.  만약 Staging Area에 넣은 파일의 변경 부분을 보고싶으면 git diff --cached 옵션을 사용한다.  git diff는 Unstaged 상태에있는것들만 보여준다. 즉 Staged 상태로 들어가있는것은 diff를 해도 나오지않는다. |

**Staged 상태 파일들 Commit하기(변경사항 커밋하기)**

수정할것들을 커밋하기위해 Staging Area에 파일정리를 했다.

Unstaged상태의 파일은 커밋되지 않는다는것을 기억해라.

|  |
| --- |
|  |
| git commit –m “메세지” |

Staging Area생량하고 커밋도할수있다..

|  |
| --- |
|  |
| git commit –a –m “메세지”  -a옵션추가. |

**파일을 삭제하기**

|  |
| --- |
|  |
| 디렉토리에서 파일삭제후 status를 확인해보면 .. deleted 상태로나온다. 위에서는 test.txt 디렉토리에서삭제  Unstage상태로 들어가있는것을 staged상태로 돌리기위해서 git add를 사용하는것이 아니라  get rm 명령어를 사용해서 처리해야한다. |
| 바로 git rm 을통해 파일삭제하면 staged 위치에서 deleted상태로 들어간다.  위에서는 README.md 파일을 해보았다, |

위내용이 커밋이되면 파일이 삭제되고, 더이상 git은 추적을하지 않는다.

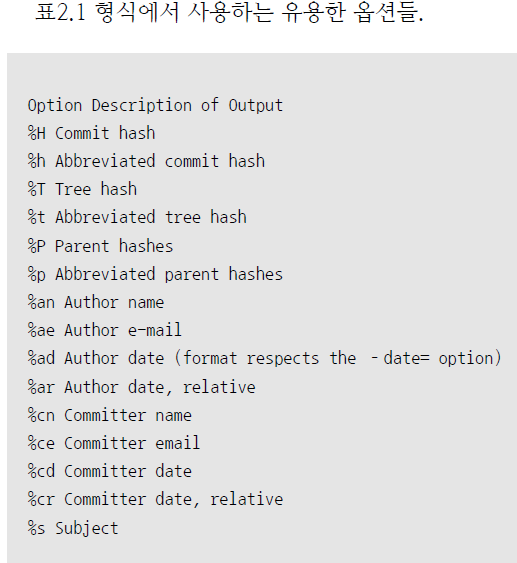
또한 Staging Area에서만 제거ㅓ되고 작업 디렉토리에 있는 파일은 지우지 않고 남겨둘수있는데. 추적만 안하게할수있다. --cached 옵션을 사용하여 명령어를 진행하면된다

**파일이름 변경**

|  |
| --- |
|  |
| git mv old new  안쪽내용을 확인해보면..  git rm ...  git add ..  이런식으로 두번해주는거뿐이다. |

**커밋 히스토리 조회하기**

|  |
| --- |
|  |
| git log |
| -p 는 diff내용을 보여준다.. -2는 최근2개만 보여주는것이다. |
| --stat 옵션으로 각 커밋의 통계 정보를 조회할수 있다. |
| git log –pertty=format: “%h %s” –graph...등으로 메시지 포맷을 정할수있다. |
| 조건제한조건 : git log –since=2.weeks 등..  -(n) 최근 n개 커밋만조회  --since, --after명시한 날짜 이후의 커밋만검색  --until, --before 명시한 날짜 이ㅣ전의 커밋만조회  --author 입력한 저자의 커밋만  --committer 입력한 커미터의 커밋만 보여준다. |

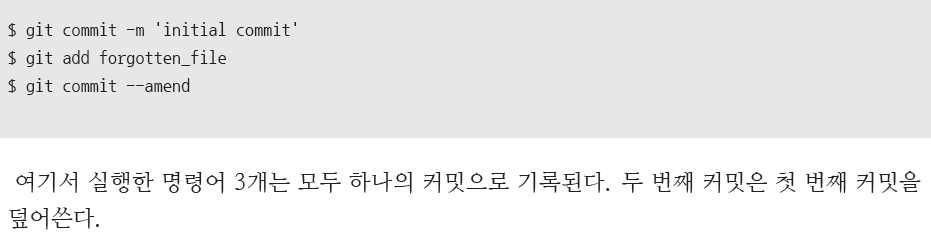


**되돌리기(Undo)**

너무 일찍 커밋을 했거나 어떤파일을 빼먹었을때 그리고 커밋 메시지를 잘못했을때 다시 커밋하고 싶으면

**커밋수정하기**

git commit --amend 명령어를 쓴다.



**파일상태를 Unstage로 변경하기**

잘못해서 git add \* 라고 실수를 했다면..

하나씩 git reset Head filename..할수있다

|  |
| --- |
|  |
| git reset HEAD filename 으로 처리할수있다.. |
|  |

**Modified 파일 되돌리기**

수정된 파일을 다시 되돌릴수 없을까? 특정 위치로 되될리는 방법이 없을까..?

|  |
| --- |
|  |
| git driectory에있는것을 체크아웃해온다. |

**리모트 알아보기**

리모트 리스트 git remote

리모트리스트 상세 git remote –v

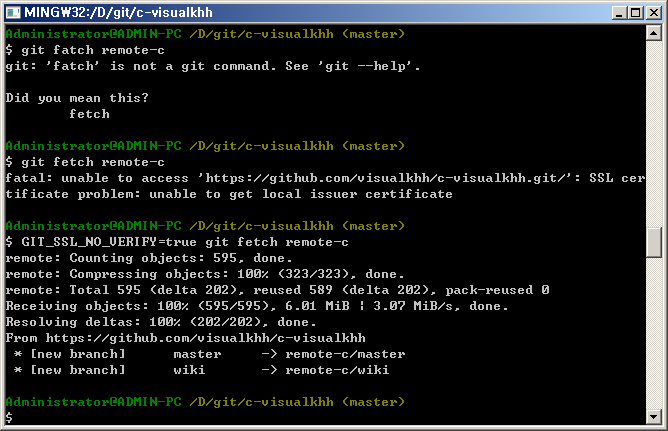
리모티 알리어스된거 보기 git remote show origin

리모트 저장소 추가하기

git remote add name git://ggg.com.git

이러면 위쪽에서 추가한 alias시킨이름으로 fetch 시킬수있다 (가져오기)

git fetch name



하지만 아직파일을 가져오진못했다.

Branch만 가져왔다. git branch –r 확인

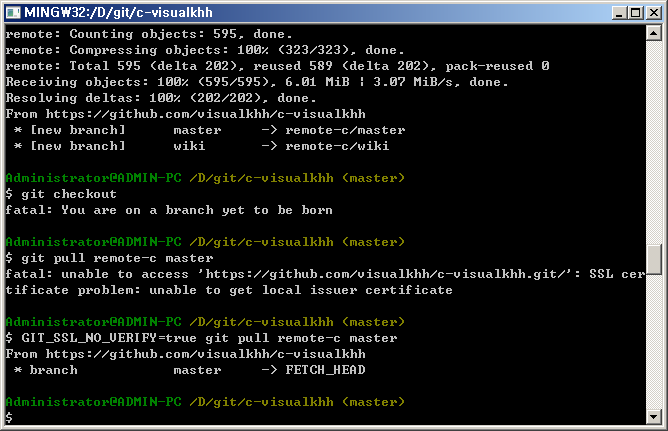
git branch -r을 하면 원격저장소의 브랜치 리스트를 볼 수 있고 git branch -a를 하면 모든 브랜치의 리스트를 볼 수 있습니다.

원격에 dev-env-refactoring이라는 브랜치가 있는데 이걸 로컬로 가지고 오고 싶다면..

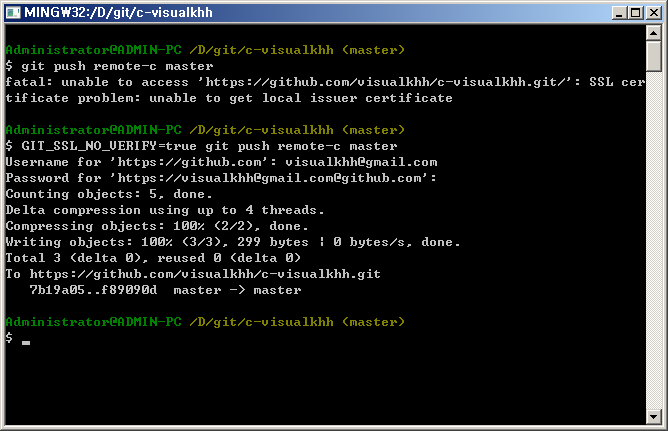
$ git checkout -b env origin/dev-env-refactoring

env는 알리어스다 안해도됨

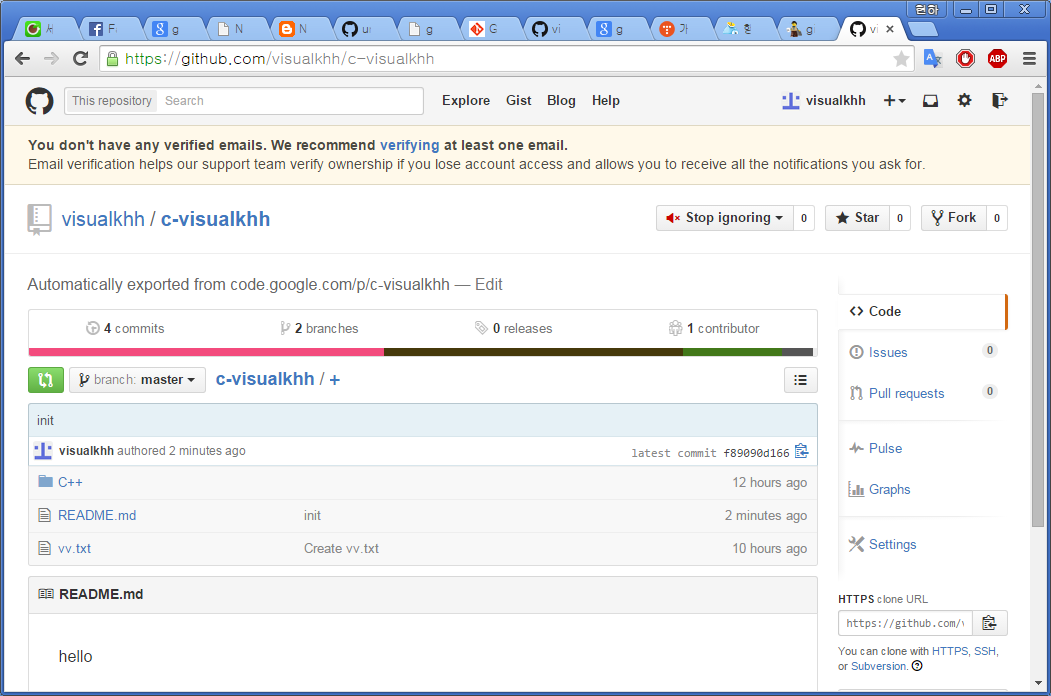
pull을해서 가져온다



add하고 commit하고



git push remote-c master



리모트쪽에 push하기

프로젝트를 공유하고 싶을 때 리모트 저장소에 push할수있다.

git push [리모트 저장소이름] [브랜치 이름] 단순하다.

**파일무시하기 gitignore**

|  |
| --- |
|  |
| 프로젝트 폴더에 .gitignore 파일을만들고 그 안쪽에 무시할 파일 이름패턴을 적어주면된다.  그런뒤 status 해보면 .gitignore 설정파일만 있을분 무시될파일은 포함되지않는다.    표준 Global패턴을 사용한다.  디렉토리는 슬래시 (/)를 끝에 사용하는것을 표현한다.  아무것도 없는 줄이나, #으로 시작하는 줄은 무시한다.  느낌표!로 시작하는 패턴은 해당 패턴의 파일을 무시한다. |

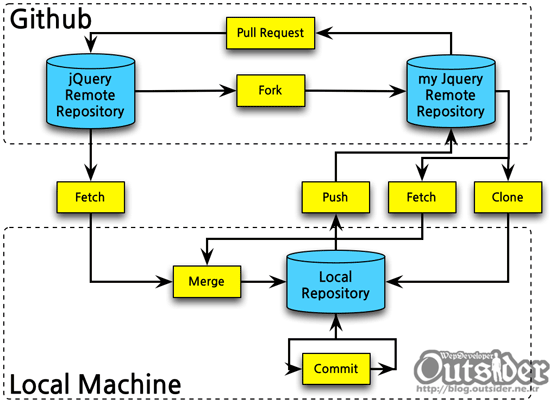
# http://blog.outsider.ne.kr/865

# [Github를 이용하는 전체 흐름 이해하기 #1](http://blog.outsider.ne.kr/865)

좀 어렵기는 하지만 Git은 정말 좋은 도구이다. Git을 쓰기 시작하면서 SVN이 얼마나 불편하고 구린지를 이해할 수 있다. 어쨌든 [Github](https://github.com/)의 엄청난 성장아래 이제는 대부분의 오프소스 프로젝트들이 Github로 이전하면서 회사에서 Git을 사용하지 않더라도 Git을 사용하지 않으면 안되는 때가 왔다. 하지만 Git은 상당히 어려운 도구이고([git 홈페이지](http://git-scm.com/)에 나온 easy to learn은 홍보성 문구로 거짓말이다. 어렵다!!) 그룹스터디를 할 때 Github을 사용하는 경우가 꽤 많았는데 사람들이 숙제를 해도 제출을 못하는 사태가 자주 발생하면서 각각의 명령어에 대한 사용법에 대한 설명도 중요하지만 전체적인 사용방식을 좀 설명할 필요가 있다고 느껴졌다.  
  
내가 생각하기에 사람들이 Git을 이해하는데 어려워하는 부분이 몇가지 있는데

* Github를 설명할 때 Git의 기능과 Github의 기능을 명확하게 구분해 주지 않는다.
* 대부분의 개발자들이 가진 Subversion에 대한 이해가 오히려 Git을 이해하는데 방해가 된다. Git은 DVCS이고 SVN은 VCS이다. 앞에 Distribute가 붙은건 마케팅적으로 괜히 붙힌게 아니라 분산 저장소이기 때문이다. Git에서 이 분산저장소라는 것은 무척 중요하다.

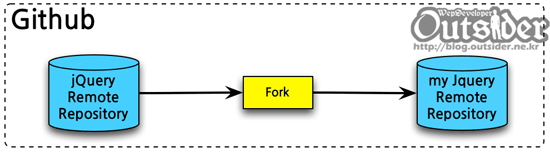
그래서 Github을 예시로 fork해서 소스를 수정하고 적용까지 하는 전체적인 흐름 과정을 이해하면 각각의 명령어를 이해하는데 좀더 도움이 되지 않을까 한다. 예제의 설명을 위해서 [jQuery](https://github.com/jquery/jquery)에서 어떤 버그를 발견해서 소스를 수정한뒤에 다시 jQuery에 적용요청을 하는 일련의 과정을 예시로 들어보자. git의 각 명령어를 일일이 다 설명하는 것은 내용이 너무 많기 때문에(모르는 명령어도 많고.. ㅡㅡ;;) 흐름을 이해할 수 있는 수준에서만 명령어를 설명할 것이다. 필요하다면 각 명령어는 따로 찾아보고 익혀야 한다.



이 과정은 도식화 하면 위처럼 된다. 좀 복잡해 보일 수 있는데 하나씩 살펴보자.  
  
  
**Fork**  
Github의 저장소를 보면 다음 화면처럼 3가지의 저장소 주소를 제공하고 있다.(Git에서 여러가지 프로토콜을 사용해서 저장소를 열어줄 수 있지만 Github가 제공하는 저장소가 이렇다.)

jQuery git 저장소의 주소 화면 

jQuery 저장소에는 쓰기 권한을 가지고 있지 않으므로(아무나 쓰기를 할 수 있다면 저장소는 당연히 엉망일 될 것이다.) 우측에 보듯이 Read-Only 권한만 있음을 알 수 있다. 만약에 지금 시나리오처럼 소스를 수정해서 적용할 것이 아니고 그냥 혼자 소스를 살펴보기만 할 것이라면 이 주소를 사용해서 git clone를 하면 로컬에 저장소를 내려받을 수 있다. 하지만 우리는 소스를 수정할 수 있는 저장소를 관리해야 하므로 Fork를 해야 한다. 저장소 관리자와 친분이 있다면 요청을 해서 해당 저장소에 Push할 수 있는 권한을 받을 수도 있겠지만 일반적으로는 Fork한 뒤에 나중에 Pull Request하는 방식으로 진행이 된다.

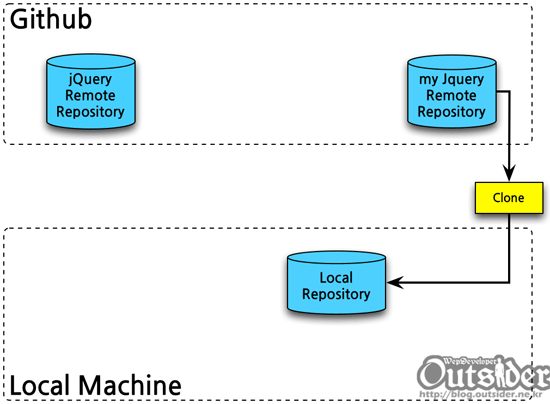


앞에서 본 다이어그램에서 위의 부분이다. jQuery의 저장소를 Fork받아서 내 저장소로 복사본을 만든다. 저장소의 우측상단을 보면 다음처런 Fork버튼을 볼 수 있다.

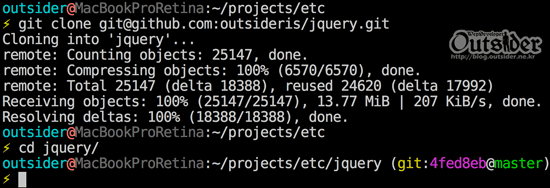
이 Fork버튼을 누르면 jQuery 저장소의 지금 상태 그대로를 복사해서 자신의 Github계정에 jQuery 저장소를 생성한다. 이 Fork는 git의 기능이 아니라 Github가 git의 기능을 추상화해서 제공하는 기능이므로 git fork같은 명령어는 없다. 간단히 생각하면 git clone을 github내에서 진행했다고 생각하면 된다.(내부 구현은 알 수 없지만....)



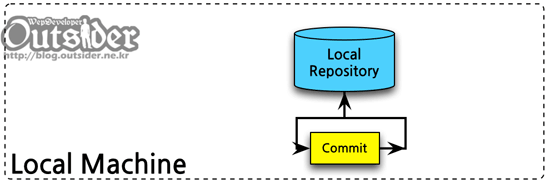
이제 자신의 계정에 jQuery저장소가 생겼고 저장소 주소에 Read+Write 권한이 있는 것을 볼 수 있다. 이제 이 저장소에는 소스를 수정해서 푸시를 할 수 있다. 이 저장소는 jQuery의 원래 저장소와 완전히 동일한(주소만 다른) 저장소이다. 분산저장소이기 때문에 저장소가 또 하나 생긴 것이고 다른 사람이 jQuery 원래 저장소 대신 내 저장소를 똑같이 Fork 받아서 수정하는 것도 당연히 가능하다 각 저장소는 모두 권한외에는 모두 동일한 기능을 가진다.  
  
  
**Clone**  
원격에서는 소스를 수정할 수 없으므로 이 저장소를 작업할 로컬 머신에 내려받아야 하는데 이 과정이 Clone이다.



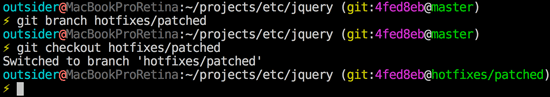
Clone는 git을 잘 몰라도 대부분 알고 있는 git clone git@github.com:outsideris/jquery.git 명령어로 수행한다. SSH 주소를 사용한 것은 git 프로토콜이 HTTPS보다 훨씬 빠르고 Github에 SSH키를 등록해 놓으면 푸시할때 암호를 입력하지 않아도 되기 때문이다.



Clone을 받아오면 로컬에 jquery 폴더가 생기고 jquery 폴더안에 git 저장소를 내려받은 것을 볼 수 있다. 저장소 이름이 jquery이기 때문에 jquery 폴더가 생겼고 다름 폴더명을 사용하려면 git clone git@github.com:outsideris/jquery.git NEW\_NAME과 같이 이름을 지정하면 된다. Fork가 Github내에서 저장소를 복사한 것이라면 clone은 원격 저장소를 로컬로 복사한 것이다.  
  
  
**Commit**  
이제 저장소를 로컬에 가져왔으므로 소스를 수정하는 작업을 해야한다. git에서는 관례적으로 메인 브랜치로 master 브랜치를 사용하지만 필요하다면 다른 브랜치를 메인으로 사용해도 아무런 문제가 없다. master는 그냥 관례적으로 약속한 브랜치일 뿐이다.



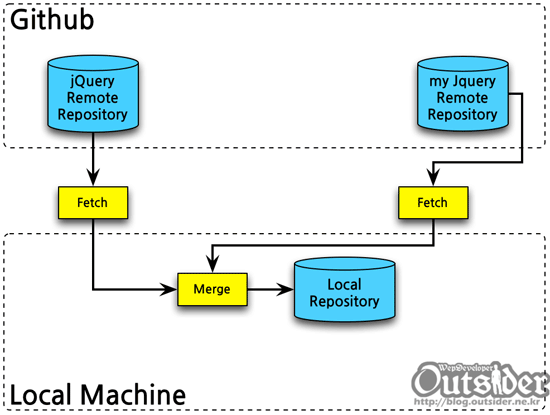
jQuery의 소스를 수정하기 위해서 master  브랜치를 수정해야 하는데 Subversion과는 다르게 Git에서는 브랜치를 생성하는 것을 권장하고 있고 일반적으로 master 브랜치에서 다른 브랜치를 생성해서 작업하는 것이 일반적이다. 여기서는 patched라는 브랜치를 생성한다.



git branch 명령어를 사용해서 현 브랜치에서 새로운 브랜치를 생성하고 해당 브랜치로 사용하는 브랜치를 바꾸었다. 여기서 hotfixes/patched라고 브랜치를 생성한 이유는 [git-flow](http://nvie.com/posts/a-successful-git-branching-model/)의 관례이고 그냥 브랜치명이 hotfixes/patched라고 지은것이고 계층적으로 구분하기 위해서 생성했다고 보면 된다. 위의 두 과정대신 git checkout -b hotfixes/patched를 사용하면 명령어 하나로 브랜치를 생성해서 바로 이동까지 할 수 있다. 사실 git을 사용하는 대부분의 과정은 이 단계에서 이루어지는데 기본적으로 소스를 수정하고 커밋할 소스를 git add한 후에 git commit을 한다. commit은 현재의 git 저장소에 소스를 적용해서 히스토리를 남긴 것으로 SVN의 commit과 다르게 원격저장소에 적용되는 것은 아니다.  
  
  
이 포스팅은 [Github를 이용하는 전체 흐름 이해하기 #2](http://blog.outsider.ne.kr/866)로 이어진다.

# [Github를 이용하는 전체 흐름 이해하기 #2](http://blog.outsider.ne.kr/866)

이 포스팅은 [Github를 이용하는 전체 흐름 이해하기 #1](http://blog.outsider.ne.kr/865)에서 이어진 글이다.  
  
  
**Fetch & Merge**



로컬에서 작업을 하다보면 원격저장소에 변경사항이 생긴다. 클론받은 이후에 원격저장소에 누군가 소스를 푸시하면 이 변경사항을 다시 로컬로 가져와야 하는데 이 과정을 fetch로 원격저장소의 변경사항을 로컬로 가져온 뒤에 로컬의 브랜치에merge하는 과정으로 이루어진다. 이 과정은 일반적으로는 git pull이라는 명령어를 통해서 한방으로 이루어지는데 git pull보다는 git fetch후에 git merge로 나누어서 작업하는 것을 보통 더 권장한다.(권장하는 이유는 여러가지가 있지만 conflict가 생겼을 때 대처가 훨씬 쉽다던지  merge를 훨씬 자유롭게 할 수 있다.)

[?](http://blog.outsider.ne.kr/866)

|  |  |
| --- | --- |
| 1  2  3 | $ git remote -v  origin    git@github.com:outsideris/jquery.git (fetch)  origin    git@github.com:outsideris/jquery.git (push) |

위 와 같이 로컬 저장소에 등록된 원격저장소 목록을 보면 clone을 받아온 대상인 원격 저장소가 origin이라는 이름으로 등록되어 있다. github에서 저장소를 새로 생성하면(fork로 생성하는 경우외에) git remote add origin git@github.com:outsideris/jquery.git와 같이 저장소를 추가하도록 안내를 하는데 이는 관례에 따라 메인 원격저장소를 origin으로 등록한 것이다. 그리고 이어서 소스를 git push -u origin master로 푸시하도록 안내하고 있는데 여기서 -u 옵션은 설정파일에 현재의 master 브랜치를 origin의(여기서는 fork받은 자신의 원격저장소) master 브랜치로 연결해 주어 다음부터는 자동으로 master브랜치에서 git push를 하면 origin의 master브랜치로 푸시가 되고 git pull을 하면 origin의 master를 fetch해서 로컬의 master로 merge하도록 설정하는 것이다. 그래서 .git/config파일의 내용을 보면 다음과 같이 master브랜치의 원격저장소와 머지할 브랜치가 설정되어 있는 것을 볼 수 있다.(clone을 받아오면 이 설정이 자동으로 추가된다.)

[?](http://blog.outsider.ne.kr/866)

|  |  |
| --- | --- |
| 1  2  3 | [branch "master"]      remote = origin      merge = refs/heads/master |

그래서 master가 아닌 [다른 브랜치를 원격저장소에 푸시하려면 대상 저장소와 브랜치를 직접 지정](http://blog.outsider.ne.kr/644)해 주어야한다.(매번하기 싫으면 위처럼 설정파일에 지정하면 된다.) 여기서는 fetch와 merge에 대해서 설명하는 단계이므로 이 부분에 대해서만 얘기하면 git pull하면 자동으로 master에 머지되도록 설정이 되어 있는 것이고 이를 다시 말하면 반드시 master에만 머지해야 하는 것은 당연히 아니다.   
  
지금까지 origin만 살펴보았는데 이 시나리오에서는 우리에게 필요한 원격저장소가 하나 더 있다. 즉, fork받은 자신의 저장소 외에 원본인 jQuery의 원격저장소가 있다. 이 원격저장소에도 누군가 계속 소스를 수정할 것이므로 jQuery를 기반으로 계속 작업을 할 것이라면 원본 jQuery저장소의 변경사항도 로컬로 가져와야 한다. 처음 git을 배울 때(정확히는 hg로 DVCS를 배웠지만...) 가장 궁금했던 것이 fork한 저장소는 fork한 시점으로 멈춰있으므로 변경사항을 가져오려면 어떻게 하는가?였다. 그래서 매번 새로운 내용이 필요할 때마다 github의 저장소를 삭제하고 다시 fork로 새 저장소를 만들어서 사용했다.(ㅡㅡ;;) 이 당시에는 git을 제대로 이해못했으므로(지금도 ㅠㅠ) 동기화기능을 github에서 제공해야 하는 것이 아닌가 하고 생각했지만 이러한 부분은 git이 자유롭게 다룰 수 있으므로 github에서 제공할 이유가 없는 기능이다.  
  
이 문제를 해결하려면 그냥 jQuery의 원본저장소도 원격저장소로 추가해주면 된다.

[?](http://blog.outsider.ne.kr/866)

|  |  |
| --- | --- |
| 1  2  3  4 | $ git remote add jquery git@github.com:jquery/jquery.git  $ git remote  jquery  origin |

앞에서 origin이라는 이름으로 원격저장소를 추가한것처럼 이번에는 jquery라는 이름으로(원하는 대로 지정하면 된다.) jquery의 원본 원격저장소의 주소를 원격저장소로 등록하고 git remote 명령어로 확인해 보면 원격저장소가 2개 등록되어 있는 것을 볼 수 있다. jquery 원본 저장소에는 push할 권한은 없지만 읽기권한은 있으므로 변경사항을 가져오는데는 아무런 문제가 없다. 이러한 점이 git이 DVCS이기 때문에 갖는 강점인데 원격저장소는 필요한대로 추가할 수 있고 어디서나 가져와서 머지할 수 있다.

[?](http://blog.outsider.ne.kr/866)

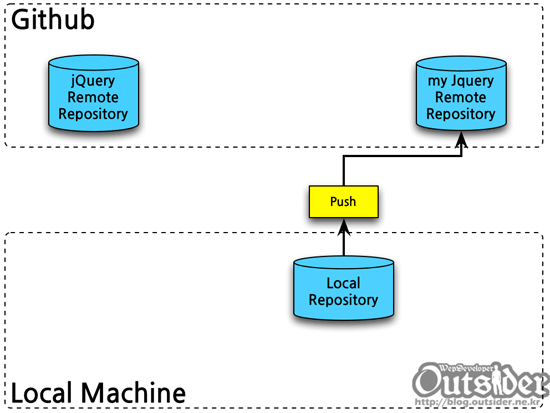
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | $ git fetch jquery  From github.com:jquery/jquery   \* [new branch]      1.8-stable -> jquery/1.8-stable   \* [new branch]      master     -> jquery/master   \* [new branch]      promisea   -> jquery/promisea  $ git merge jquery/master  Already up-to-date. |

이제 jquery저장소(이름을 그렇게 주었으므로)의 내용을 fetch받아오면 위처럼 새로운 브랜치들이 생기고 이 내용은 jquery/master라는 이름으로 로컬에서 접근할 수 있다.(다른 브랜치도 몇 개 가져온 것을 볼 수 있다.) git은 SVN과는 다르게 히스토리를 검색하기 위해서 원격 저장소에 접근하지 않아도 되고 로컬에 전체 히스토리를 모두 내려온다. 그래서 실제로 origin과 jquery 아래 내려받은 저장소의 브랜치가 존재하고 있다.

[?](http://blog.outsider.ne.kr/866)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | $ git branch -r    jquery/1.8-stable    jquery/master    jquery/promisea    origin/1.8-stable    origin/HEAD -> origin/master    origin/master    origin/promisea |

git branch -r은 원격저장소의 브랜치를 보는 명령어인데 위처럼 현재의 원격 브랜치를 모두 볼 수 있다. 이 원격 브랜치는 원격에서 받아온 브랜치를 의미하지만 실제로 그 내용은 모두 로컬에 내려받은 것이다. 하지만 이 브랜치에는 커밋을 하는 등의 수정은 할 수 없고 오로지 읽기 전용이다. 그래서 이 원격브랜치들에서 새로운 브랜치를 생성하거나 기존의 브랜치에 머지를 하는 것이 가능하다. 앞에서 본 것처럼 이 원격 저장소의 변경사항을 업데이트하려면 git fetch jquery처럼 원격저장소별로 업데이트를 하고 브랜치별로 업데이트를 하진 않는다. 앞에서 fetch로 업데이트를 한 후에 현재 브랜치(master)에 머지를 했는데 꼭 master에만 머지해야 하는 것은 아니다. 즉, 필요에 따라서는 작업을 master에다 하고 원격 저장소의 내용을 추적하는 다른 브랜치를 만들어서 머지할 수도 있다.  
  
앞에서 commit과정을 설명할 때 일반적으로 작업을 master에서 하지 않고 별도의 브랜치를 만들어서 한다고 설명했는데 여기서 설명하고 있는 시나리오처럼 jQuery 소스를 수정하는 경우 특별한 이유가 없다면 자신이 작업하는 master브랜치를jquery/master 브랜치와 동일하게 유지하는 것이 좋다. 그래서 merge할 때 fast-forward되도록 master를 유지하는 것이 히스토리를 좀 더 이쁘게 할 수 있는 방법이다.(fast-forward가 되지 않으면 merge했다는 새로운 커밋이 하나 생기게 된다.)  
  
  
**Push**  
이제 로컬에서 수정작업을 어느정도 완료했다면 자신의 원격저장소에 수정한 커밋들을 푸시해야 한다.

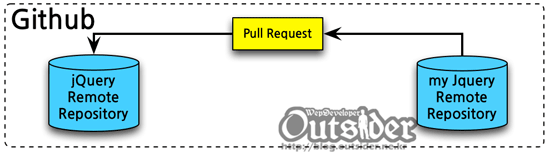


이 과정은 로컬저장소에 커밋한 내역을 원격저장소에 적용하는 것으로 git push origin master처럼 푸시할 원격저장소명과 브랜치를 지정해야 한다.(앞에서 처럼 -u옵션으로 설정했다면 master에선 git push만 해도 된다.)지정해야 한다는 것은 권한만 있다면 어떤 원격저장소에도 또는 다른 브랜치에도 푸시할 수 있다는 것을 의미한다. 대부분의 git 저장소는 fast-forward할 수 있는 경우에만 푸시할 수 있도록 하고 있고 이는 Github도 마찬가지이다.   
  
앞에서 봤듯이 jQuery의 원본저장소와 자신의 원격저장소의 master 브랜치를 계속 싱크하려면 jquery 원본저장소를 fetch받아서 master에 머지한 뒤에 자신의 원격저장소의 master로 푸시하면 된다. 이 과정을 계속 반복하면 항상 두 master 브랜치의 동기화를 할 수 있다. commit과정에서 수정사항을 보통 별도의 브랜치를 만들어서 작업한다고 했는데 [git에서 원격저장소에 branch와 tag를 push하기](http://blog.outsider.ne.kr/644)에서 설명했듯이git push 원격저장소명 로컬브랜치명:원격브랜치명으로 원격저장소에 새로운 브랜치를 생성하면서 푸시해야 한다.

[?](http://blog.outsider.ne.kr/866)

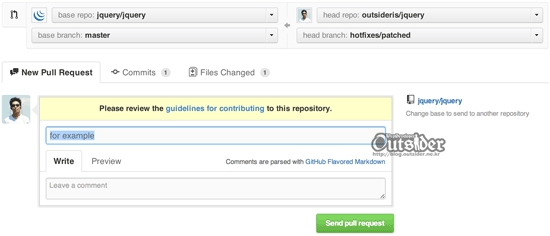
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | $ git push origin hotfixes/patched  Counting objects: 5, done.  Delta compression using up to 8 threads.  Compressing objects: 100% (3/3), done.  Writing objects: 100% (3/3), 309 bytes, done.  Total 3 (delta 2), reused 0 (delta 0)  To git@github.com:outsideris/jquery.git   \* [new branch]      hotfixes/patched -> hotfixes/patched |

이처럼 푸시를 하면 원격에 새로운 브랜치가 생성된 것을 볼 수 있다.  
  
  
**Pull Request**



이제 원격에 올린 변경사항을 원본 jQuery 저장소에 적용하기 위해서 Pull Request를 보내야 한다. Pull Request도 git 자체에 있는 기능은 아니고 Github에서 제공하는 기능이라고 할 수 있다.(써본 적은 없지만 Github를 쓰지 않는다면 패치내용을 메일로 보내면 적용하는 방식을 취했었다.) Pull Request를 사용하는 이유는 jQuery 원본저장소에는 쓰기권한이 없기 때문에 수정한 내용을 Pull Request로 보내면 jQuery 저장소의 커미터들이 내용을 확인한뒤에 승인을 하면 Pull Request의 내용이 jQuery 원본저장소에 merge가 된다.(물론 승인하지 않고 거절할 수도 있다.)

이제 Github의 자신의 저장소에서 변경사항을 적용한 브랜치페이지로 이동해서 상단의 Pull Request버튼을 누르면 된다. 최근에 변경한 내용은 중간에도 Pull Request 버튼이 나온다. 이 버튼을 클릭하면 다음과 같이 Pull Request를 보내는 화면을 볼 수 있다.



상단에 Pull Request를 보내는 자신의 브랜치와 대상이 되는 브랜치를 지정할 수 있고 그 아래 Pull Request에 대한 내용을 볼 수 있다. jQuery의 경우CONTRIBUTING.md 파일이 존재하기 때문에 위에 해당 부분에 대한 안내가 나온다. 오픈소스 프로젝트마다 Pull Request를 받아주는 약속이 다르므로 보내기 전에 이를 먼저 확인해야 한다. 소스 수정이 잘 되었더라도 이 약속을 제대로 지키지 않으면 받아주지 않는다. 탬에서 Commits나 Files Changed를 클릭하면 Pull Request를 보내는 커밋과 변경사항이 제대로 되었는지 확인할 수 있다.   
  
  
  
물론 다른 사람의 소스를 Fork해서 나중에 Pull Request까지 보내는 경우가 많진 않지만(이정도까지 하는 사람은 대부분 이미 Git에 익숙해져있다고 생각한다.) 직접 저장소를 생성해서 혼자 사용한다고 하더라도 이 전체 흐름의 일부로 포함되기 때문에 이 흐름에서 해당 부분만 참고해서 사용하면 된다.

## **링크 & 자료**

### **그래픽 클라이언트**

* [GitX (L) (OS X용, 오픈 소스 소프트웨어)](http://gitx.laullon.com/)
* [Tower (OS X용)](http://www.git-tower.com/)
* [Source Tree (OS X용, 무료)](http://www.sourcetreeapp.com/)
* [GitHub for Mac (OS X용, 무료)](http://mac.github.com/)
* [Gitbox (OS X용, App Store)](https://itunes.apple.com/kr/app/gitbox/id403388357?mt=12)

### **한글 안내서**

* [Pro Git](http://git-scm.com/book/ko)
* [A Visual Git Guide](http://marklodato.github.io/visual-git-guide/index-ko.html)
* [Git 작업 흐름](http://dalinaum-kr.tumblr.com/post/15516936704/git-work-flow)
* [Git 브랜치 배우기](http://learnbranch.urigit.com/)

### **영문 안내서**

* [Pro Git](http://git-scm.com/book/)
* [Think like a git](http://think-like-a-git.net/)
* [GitHub Help](http://help.github.com/)
* [A Visual Git Guide](http://marklodato.github.io/visual-git-guide/index-en.html)

문제점 해결

[**[Git for Windows] Protocol https not supported or disabled in libcurl**](http://prkyaong.tistory.com/13)

2014/04/07 15:50

[[트러블슈팅](http://prkyaong.tistory.com/category/%ED%8A%B8%EB%9F%AC%EB%B8%94%EC%8A%88%ED%8C%85)]

C:\Program Files\Git\bin 로부터 libcurl.dll 을 가져와, C:\Windows\System32 로 복사하면 해결된다.

**1. 원격지 Repogitory Clone 또는 Push 시 libcurl 관련 에러**  
  
C:\dev\git>git clone https://github.com/JungYeongGeun/spring-boot.git  
Cloning into 'spring-boot'...  
fatal: unable to access 'https://github.com/JungYeongGeun/spring-boot.git/': Pro  
tocol https not supported or disabled in libcurl  
  
**Solution >> c:\windows\system32\libcurl.dll 을 삭제 한다.**  
  
**2. (1)번을 해결하면 아래와 같은 SSL 문제가 발생하는데**  
  
C:\dev\git>git clone https://github.com/JungYeongGeun/spring-boot.git  
Cloning into 'spring-boot'...  
fatal: unable to access 'https://github.com/JungYeongGeun/spring-boot.git/': SSL  
 certificate problem: unable to get local issuer certificate  
  
**Solution >> 명령어 앞에 env GIT\_SSL\_NO\_VERIFY=true 을 사용한다.**  
  
C:\dev\git>env GIT\_SSL\_NO\_VERIFY=true git clone https://github.com/JungYeongGeun  
/spring-boot.git  
Cloning into 'spring-boot'...  
remote: Counting objects: 3, done.  
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0  
Unpacking objects: 100% (3/3), done.  
Checking connectivity... done.  
  
**정상처리 되었다.**

## http://www.gliderwiki.org/wiki/246

## 03. git clone, push,full, fetch를 이용한 소스 공유

## Table of Contents

* [git clone 을 통해 코드 가져오기](http://www.gliderwiki.org/wiki/246#git clone 을 통해 코드 가져오기)
* [git clone 으로 원격의 저장소 코드를 가져오기](http://www.gliderwiki.org/wiki/246#git clone 으로 원격의 저장소 코드를 가져오기)
* [git fetch, git pull, git push](http://www.gliderwiki.org/wiki/246#git fetch, git pull, git push)
* [git pull , push 의 동작 방식](http://www.gliderwiki.org/wiki/246#git pull , push 의 동작 방식)
* [리모트 저장소 확인하기](http://www.gliderwiki.org/wiki/246#리모트 저장소 확인하기)

## git clone 을 통해 코드 가져오기

우리는 지난 시간에 로컬 저장소를 git init 이라는 명령어로 지정해주었고 파일을 생성하여 로컬 저장소에 add 해주었다.   
git 에 만든 파일을 수정하여 git commit -m 을 통해 로컬 저장소의 index에 추가 하였고 원격 저장소를 추가하여 실제 수정한 파일을 원격 저장소로 push 하였다.   
  
다시 명령어를 상기해보자. 

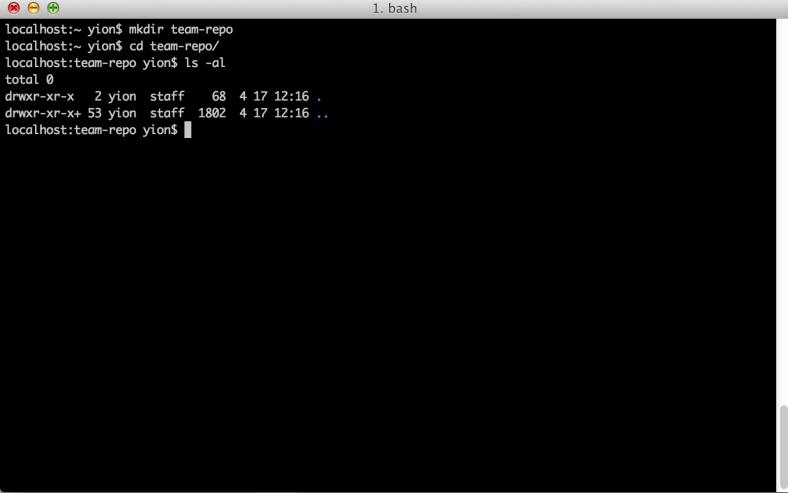
[?](http://www.gliderwiki.org/wiki/246)

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | git init  git add README.md  git status  git commit -m "first commit"  git remote add origin https://github.com/gliderwiki/git-sample-project.git  git push -u origin master |

이 명령들이 익숙해져야 한다. 이해가 되지 않는다면 이전 강좌를 다시 한번 꼼꼼히 살펴보고 넘오오길 바란다.   
  
이제, git clone 을 통해서 원격 저장소의 특정 프로젝트를 로컬에 가져와서 개발을 한 후 push, pull, fetch, merge 등의 명령어를 통해 소스를 관리하고 공유해보는 것을 알아보도록 하자. 

### git clone 으로 원격의 저장소 코드를 가져오기

보통 팀작업을 할 때 원격의 저장소를 내 로컬에 땡겨와서 수정하게 될것이다.   
여러명이 수정하면 소스의 충돌도 날것이고, 그걸 merge 해야 한다거나, 다른 사람 소스도 그때 그때 내려받거나, 내 소스를 올리는 작업이 버전관리의 기본이다.   
  
아래와 같이 원격 리파지토리의 팀 소스를 내려받기 위해 폴더를 하나 생성한 후 git clone 으로 땡겨오도록 한다. 

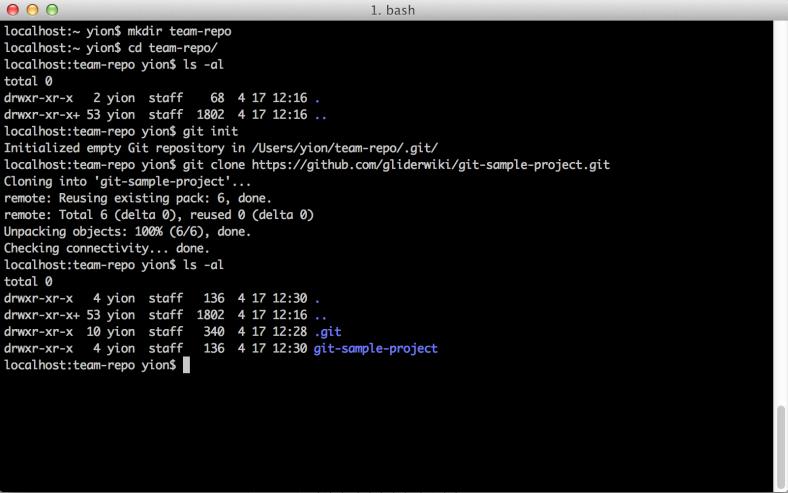


이제 git 작업 디렉토리라는 것을 명시하기 위해 git init 을 추가 한다.   
  
원격지 (여기서는 이전 강좌에서 등록한 git의 리파지토리 - git-sample-project) 소스를 내려받도록 하자. 

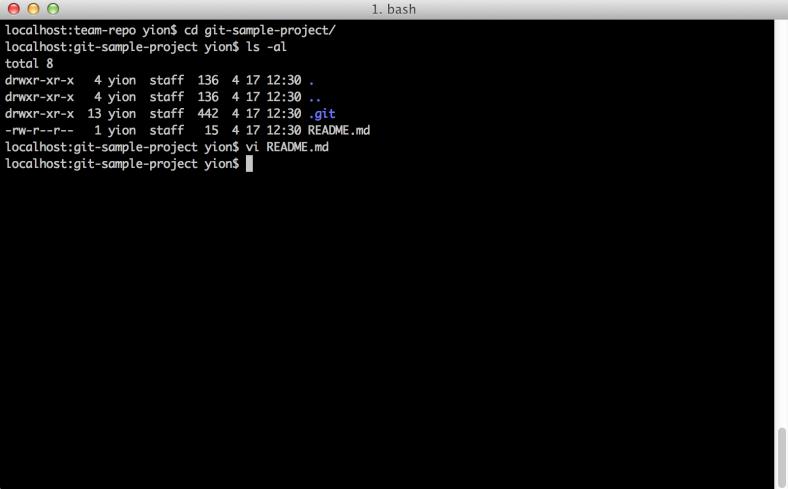
[?](http://www.gliderwiki.org/wiki/246)

|  |  |
| --- | --- |
| 1  2 | git init  git clone https://github.com/gliderwiki/git-sample-project.git |

$ ls -al 을 입력하여 로컬에 소스가 땡겨져있는지 확인해보자. 



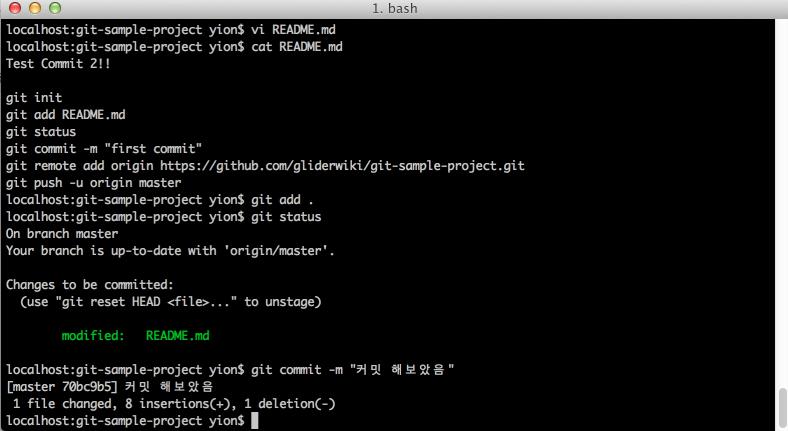
다시 내가 수정하고자 하는 파일을 열어서 수정하고 서버에 push 해보자.   
  
아래와 같이 프로젝트 폴더로 이동하여 vi 로 README.md 파일을 수정한다. 



애초에 입력되어 있던 Test Commit 2!! 구문 밑에 아래와 같이 소스를 수정해주었다. 

[?](http://www.gliderwiki.org/wiki/246)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | Test Commit 2!!    git init  git add README.md  git status  git commit -m "first commit"  git remote add origin https://github.com/gliderwiki/git-sample-project.git  git push -u origin master |



수정한 파일을 로컬 리파지토리에 추가한 후 커밋해보자. 

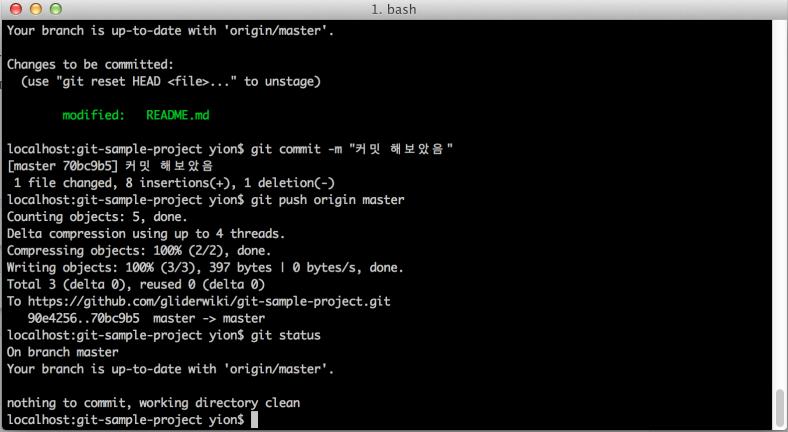
[?](http://www.gliderwiki.org/wiki/246)

|  |  |
| --- | --- |
| 1  2  3 | git add .  git status  git commit -m "커밋 해보았음" |

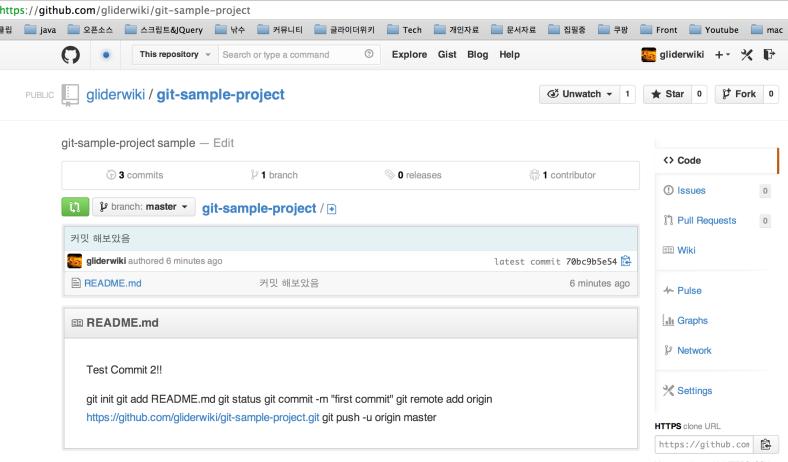
git status 로 현재 상태를 보면 master 브랜치에 README.md 파일이 수정된것을 확인 할 수 있다.   
  
여기서 중요한 것은 commit 명령어만으로는 서버에 올라가지 않는다는 것이다. (왜 인지는 이전 강좌에서 설명했다)   
수정된 파일을 서버에 추가해주기 위해서는 push를 이용해야 한다. 

[?](http://www.gliderwiki.org/wiki/246)

|  |  |
| --- | --- |
| 1 | git push origin master |



git 사이트에 수정된게 반영된것을 확인할 수 있다. 

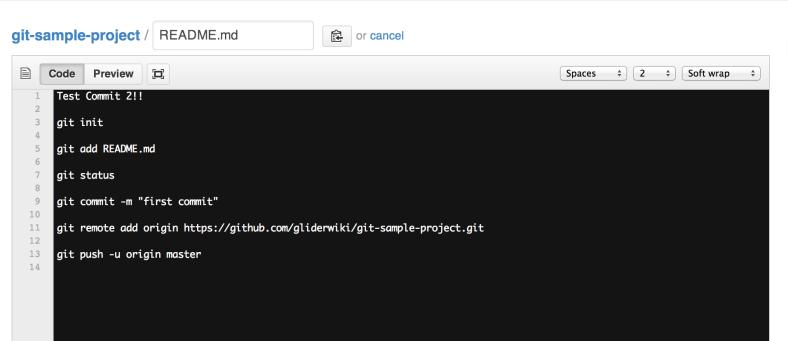


이제 팀 작업이라는 가정을 하고 다른 사람이 README.md를 수정한 경우 어떤일이 벌어질까 생각해보자.   
  
일단 버전관리 특성상 다른 사람이 해당 파일을 고치고 반영했다면, 내가 다시 그 파일을 수정하기 위해서는 그 파일을 새로 받아야 버전간의 충돌이 나지 않을 것이다.   
  
github 사이트에서 README.md 파일을 직접 수정해보고 (다른 사람이 소스를 수정했다고 가정하고) 내 버전과의 소스 관리가 어떻게 되는지 살펴보도록 한다. 

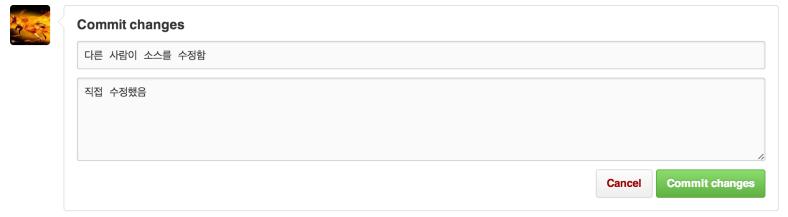
### git fetch, git pull, git push

github 에서 README 파일을 클릭하여 들어간 후 edit 버튼을 클릭하여 파일을 수정해보자 

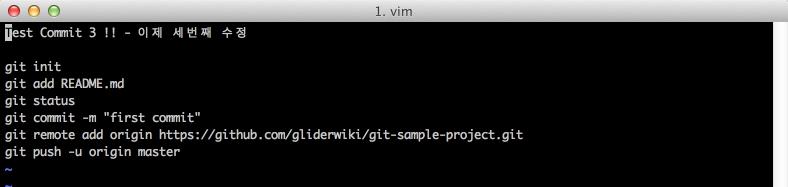




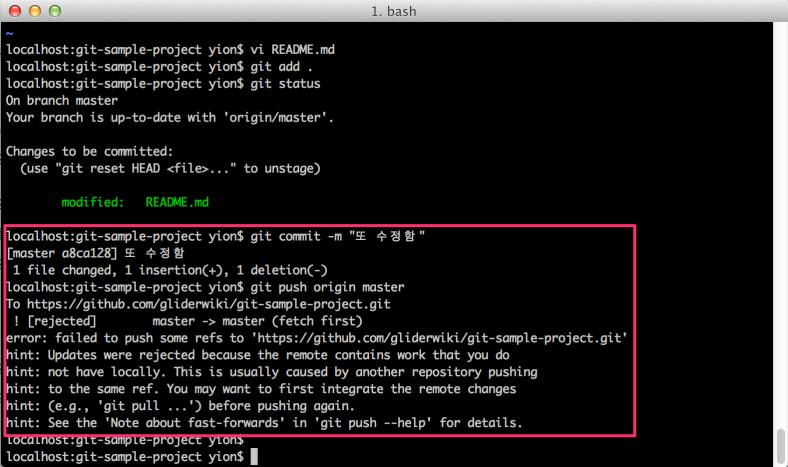
개행 처리 몇줄을 하고 하단에 수정버튼을 클릭해서 직접 수정한것을 반영하였다. 



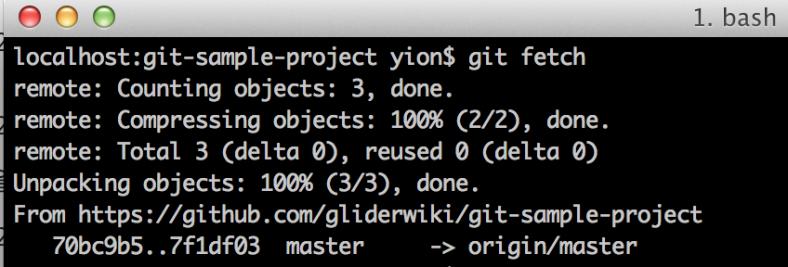
해당 파일이 다른 누군가에 의해서 수정된 상황인데, 이제 내 로컬에서 다시 파일을 수정해보면 당연히 충돌이 나게 될 것이다.   
vi 로 해당 파일을 수정한다.   
  
$ vi README.md 



세번째 수정이라는 문구를 소스에 적용한 후 저장, 추가, 커밋 해보도록 한다. 



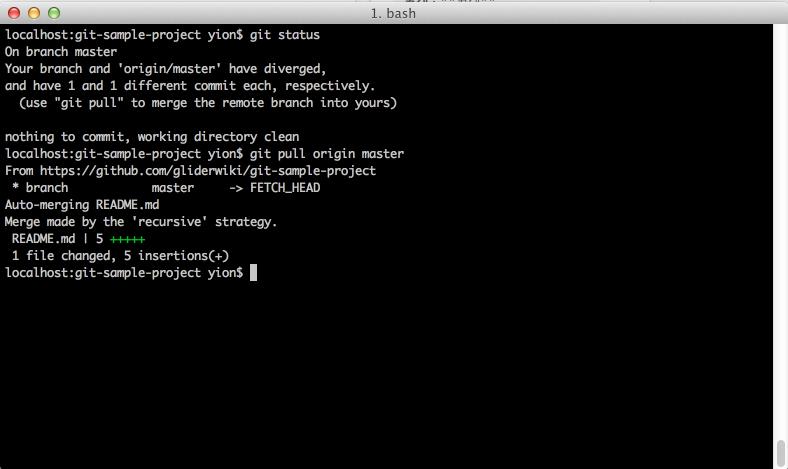
위와 같이 git push origin master 명령어로 서버에 반영하였으나 rejected 되었다고 나온다.   
왜냐면 **다른 사람이 내 로컬의 최종 버전 README.md를 한번 더 수정해서 서버에 반영 해 놨는데 내 소스는 최종본이 아니기 때문에 (수정한것과 버전차이가 나기 때문에) 최종 버전을 다시 한번 내려받아야 버전간의 충돌이 없어질 것**이다.   
  
따라서, 틈틈히 서버의 소스 변경 부분을 로컬에 반영해주어야 하는데 이때 쓰는것이 git fetch 이다. 



fetch를 통해 서버의 최신 버전을 받아왔다.   
  
git status 를 확인해보면 소스는 받아왔지만 내 로컬에 반영되진 않은 것을 확인할 수 있다.   
이때 git pull 을 이용해 원격 브랜치의 소스를 로컬에 머지하라고 친절하게 메시지가 출력된다. git merge 의 경우 좀 더 복잡한 설명이 추가 되어야 하므로 차후에 Branch 추가하는 부분에서 상세하게 설명하도록 하겠다. 

[?](http://www.gliderwiki.org/wiki/246)

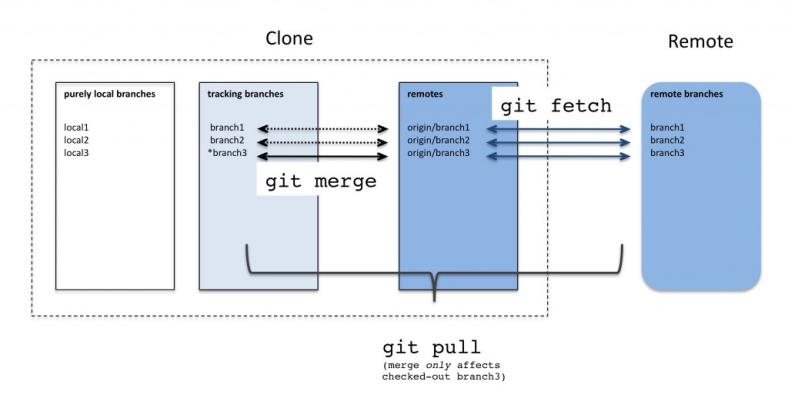
|  |  |
| --- | --- |
| 1 | git pull origin master |



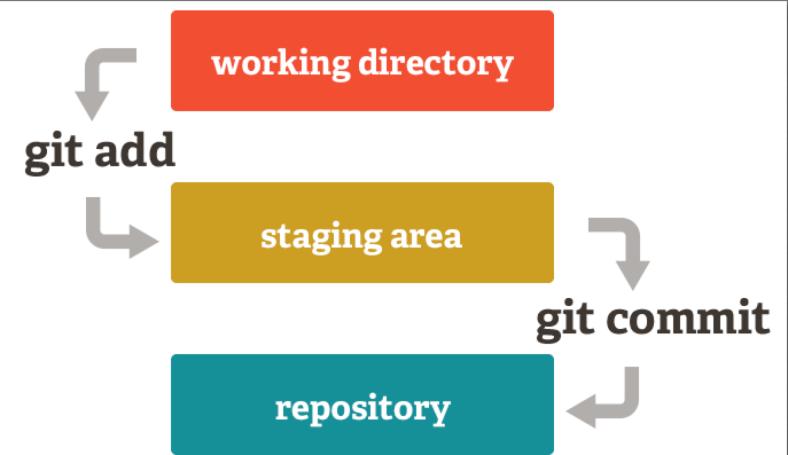
자동으로 머지 되는것을 확인할 수 있다. (에디터 등으로 열어서 실제 소스 변경분이 로컬에 적용 된건지 확인해보자) 

### git pull , push 의 동작 방식

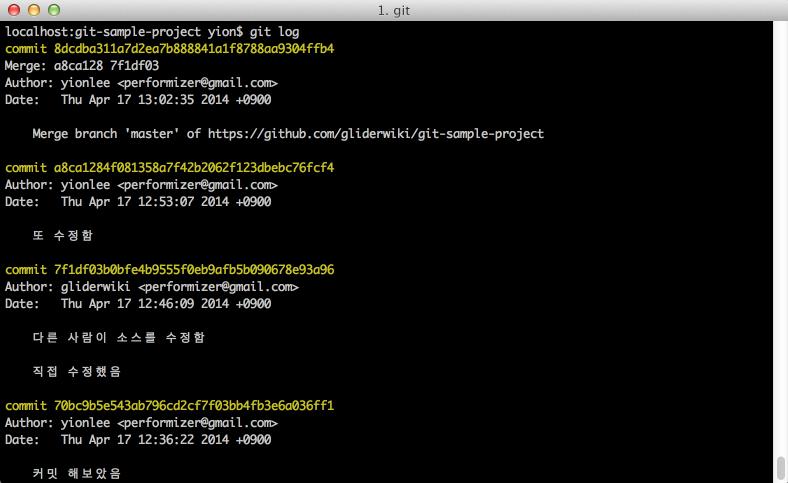
원격 저장소의 수정 파일들을 git fetch 로 땡겨오면, 실제 git 의 remotes 영역(repository) 에 파일이 올라온다.   
로컬 저장소에서는 fetch 된 목록을 git merge 를 통해 반영할 수 있는데, 이 두가지를 동시에 수행하는 것이 git pull 이다. 



다시 설명하자면, fetch 와 pull은 아래와 같은 차이가 있다.   
  
Fetch : 원격 저장소의 소스를 로컬 저장소로 가져오지만 로컬 소스와 merge 를 하지는 않는다.   
Pull : 원격 저장소의 소스를 로컬 저장소로 가져와서 현재 작업중인 소스들의 Merge 작업까지 통합하여 수행한다   
  
실제 내 작업 디렉토리(working directory) 에 파일을 수정하고 git add 를 통해 staging area 에 파일을 추가하면 git commit 을 통해 로컬의 repository 영역에 파일이 추가 되는 것이 로컬에서 최종적으로 작업한 파일들을 저장하는 개념으로 보면 git push 는 작업된 모든 repository 의 목록을 원격 저장소로 업로드 하는 것이다. 



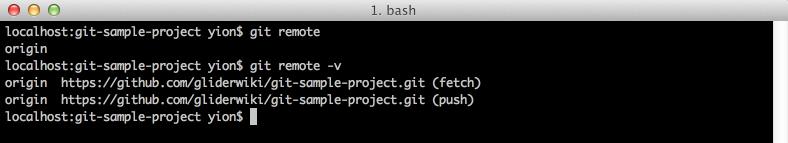
지금까지의 작업 내역은 git log를 통해 확인할 수 있다. 



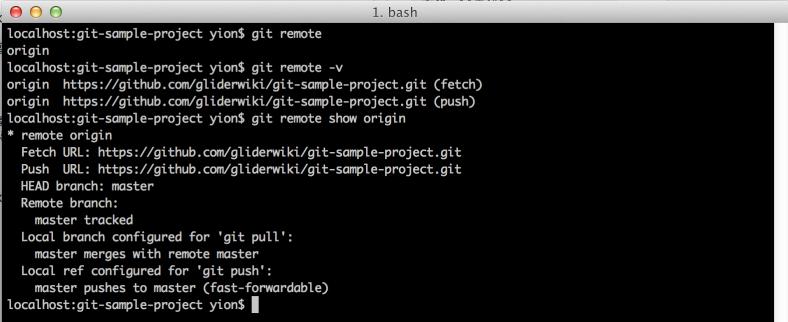
또한 git status를 통해 현재 어떤 상태로 작업하고 있는지 확인할 수 있다. 

## 리모트 저장소 확인하기

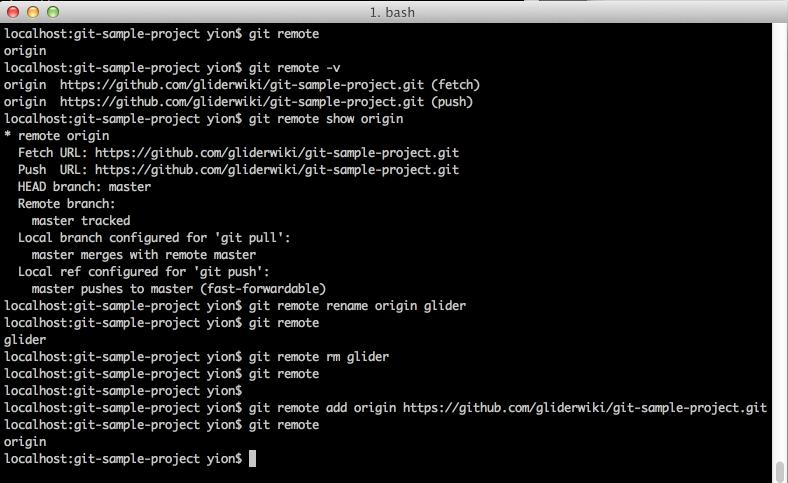
git remote 명령으로 현재 프로젝트에 등록된 리모트 저장소를 확인할 수 있다. 이 명령은 리모트 저장소의 단축 이름을 보여준다. 저장소를 Clone하면 origin이라는 리모트 저장소가 자동으로 등록되기 때문에 origin이라는 이름을 볼 수 있다:   
  
원격 저장소는 git remote 를 통해 확인할 수 있다. 



-v옵션을 주어 단축이름과 URL을 함께 볼 수 있다   
또한 show 명령어로 좀 더 상세한 정보를 살펴볼 수 있다.   
  
$ git remote show [리모트 저장소 이름] 



원격 저장소에 추가된 파일을 반영하려면 **git push [리모트 저장소 이름] [브랜치 이름]**으로 추가해줌으로써 작업을 반영한다고 이전에 설명한바 있다.   
  
$ git push origin master   
  
더 상세하게 파고 들어서, 만약 원격 저장소의 origin을이 아닌 다른 이름으로 변경하고자 한다면 git remote rename origin [변경이름] 옵션으로 적용할 수 있다.   
  
$ git remote rename origin glider   
  
이렇게 되면 리모트 저장소의 브랜치 이름도 바뀐다. 여태까지 origin/master로 리모트 저장소 브랜치를 사용했으면 이제는 glider/master라고 사용해야 한다.   
  
생성된 리모트 저장소를 삭제해야 한다면 git remote rm 명령을 사용한다.   
  
$ git remote rm glider   
  
작업 영역을 삭제하였기 push 할때 어디에 해야 할지 몰라 에러를 뱉어내는데 (origin을 glider로 수정하고 glider 를 삭제했으므로 더 이상의 작업영역이 존재 하지 않는다) 이때는 이전 강좌에서 정리한데로 git remote add [단축이름] [url] 를 이용하여 저장소를 만들어주어야 한다. 



이제까지 작업한 것들을 다시 한번 반복하는 의미에서 README.md파일을 수정해보고 새로 추가한 remote 영역에 다시 반영해보도록 하자.   
  
아래와 같이 네번째 수정본이 반영된것을 확인해보고 git 의 기본 사항을 기억해두길 바란다. 

