## You have an empty repository

To get started you will need to run these commands in your terminal.

New to Git? [Learn the basic Git commands](http://docs.atlassian.com/bitbucketserver/docs-0414/Basic+Git+commands?utm_campaign=in-app-help&utm_medium=in-app-help&utm_source=stash)

### **Configure Git for the first time**

git config --global user.name "Kyung-Tak LEE"

git config --global user.email "kyung.lee@amadeus.com"

### **Working with your local repository**

#### I just want to clone this repository

If you want to simply clone this empty repository then run this command in your terminal.

git clone https://kyung.lee@rndwww.nce.amadeus.net/git/scm/mqh/mqhcv-v2.git

#### My code is ready to be pushed

If you already have code ready to be pushed to this repository then run this in your terminal.

cd existing-project

git init

git add --all

git commit -m "Initial Commit"

git remote add origin https://kyung.lee@rndwww.nce.amadeus.net/git/scm/mqh/mqhcv-v2.git

git push -u origin master

#### My code is already tracked by Git

If your code is already tracked by Git then set this repository as your "origin" to push to.

cd existing-project

git remote set-url origin https://kyung.lee@rndwww.nce.amadeus.net/git/scm/mqh/mqhcv-v2.git

git push -u origin master

#### All done with the commands?

$ git add -all

error: did you mean `--all` (with two dashes ?)

$ git add [ list of files, --all, **\***.txt, docs/**\***.txt, docs/, “**\***.txt”, **\*** ]

kyung.lee@LOND6628 MINGW64 /d/eclipse\_kellper/worksp\_mqhcv/git\_repository/mqhcv (master)

**$ git add \***

warning: LF will be replaced by CRLF in mqhcv/trunk/IRhistory/.gitignore.

The file will have its original line endings in your working directory.

warning: LF will be replaced by CRLF in mqhcv/trunk/aircode/.gitignore.

The file will have its original line endings in your working directory.

warning: LF will be replaced by CRLF in mqhcv/trunk/mqhcv/classes/.gitignore.

The file will have its original line endings in your working directory.

warning: LF will be replaced by CRLF in mqhcv/trunk/mqhcv/exploded-archives/mqhcv.war/home.xhtml.

The file will have its original line endings in your working directory.

warning: LF will be replaced by CRLF in mqhcv/trunk/mqhcv/exploded-archives/mqhcv.war/layout/menu.xhtml.

The file will have its original line endings in your working directory.

warning: LF will be replaced by CRLF in mqhcv/trunk/mqhcv/exploded-archives/mqhcv.war/layout/template.xhtml.

The file will have its original line endings in your working directory.

warning: LF will be replaced by CRLF in mqhcv/trunk/mqhcv/exploded-archives/mqhcv.war/stylesheet/theme.css.

The file will have its original line endings in your working directory.

kyung.lee@LOND6628 MINGW64 /d/eclipse\_kellper/worksp\_mqhcv/git\_repository/mqhcv (master)

**$ git commit -m "mqhcv.eqr rebuilt for new boxes"**

[master warning: LF will be replaced by CRLF in mqhcv/trunk/IRhistory/.gitignore.

The file will have its original line endings in your working directory.

warning: LF will be replaced by CRLF in mqhcv/trunk/aircode/.gitignore.

The file will have its original line endings in your working directory.

warning: LF will be replaced by CRLF in mqhcv/trunk/mqhcv/classes/.gitignore.

The file will have its original line endings in your working directory.

35dae63] mqhcv.eqr rebuilt for new boxes

warning: LF will be replaced by CRLF in mqhcv/trunk/IRhistory/.gitignore.

The file will have its original line endings in your working directory.

warning: LF will be replaced by CRLF in mqhcv/trunk/aircode/.gitignore.

The file will have its original line endings in your working directory.

warning: LF will be replaced by CRLF in mqhcv/trunk/mqhcv/classes/.gitignore.

The file will have its original line endings in your working directory.

12 files changed, 19 insertions(+), 3 deletions(-)

rewrite mqhcv/trunk/mqhcv/exploded-archives/mqhcv.jar/com/amadeus/etc/mqhcv/SearchIcpItem.class (67%)

rewrite mqhcv/trunk/mqhcv/exploded-archives/mqhcv.jar/com/amadeus/etc/mqhcv/SystemEnum.class (79%)

rewrite mqhcv/trunk/mqhcv/exploded-archives/mqhcv.jar/com/amadeus/etc/mqhcv/TxQueryParserEnum.class (96%)

kyung.lee@LOND6628 MINGW64 /d/eclipse\_kellper/worksp\_mqhcv/git\_repository/mqhcv (master)

**$ git push origin master**

Counting objects: 37, done.

Delta compression using up to 8 threads.

Compressing objects: 100% (25/25), done.

Writing objects: 100% (37/37), 1006.49 KiB | 467.00 KiB/s, done.

Total 37 (delta 13), reused 0 (delta 0)

To https://rndwww.nce.amadeus.net/git/scm/mqh/mqhcv.git

9d8fa33..35dae63 master -> master

kyung.lee@LOND6628 MINGW64 /d/eclipse\_kellper/worksp\_mqhcv/git\_repository/mqhcv (master)

**$ git log**

commit 35dae63b1e75eb6047d2a0f809cef385267d71e8

Author: ktlee88 <ktlee88@yahoo.com>

Date: Mon Nov 13 09:57:33 2017 +0000

mqhcv.eqr rebuilt for new boxes

commit 9d8fa33c3e8ecaec9b99f4c1336795e81887231d

Author: ktlee88 <ktlee88@yahoo.com>

Date: Tue Jul 18 16:07:55 2017 +0100

Initial commit

commit 32c503f58a77e04ac432a7cbf51ef8652720687a

Author: Peter Cardoe <pcardoe@amadeus.com>

Date: Fri Apr 8 15:12:05 2016 +0100

Initial Commit

git 시작하기

## 새로운 저장소 만들기, 폴더를 하나 만들고, 그 안에서 아래 명령을 실행하세요. git init ,새로운 git 저장소가 만들어집니다.

## 저장소 받아오기,

## 로컬 저장소를 복제(clone)하려면 아래 명령을 실행하세요. git clone /로컬/저장소/경로

**원격 서버의 저장소를 복제 하려면 아래 명령을 실행하세요.**  
git clone ~~(사용자명@호스트):~~/원격/저장소/경로

<https://rndwww.nce.amadeus.net/git/scm/mqh/mqhcv.git>

## 작업의 흐름

여러분의 로컬 저장소는 git이 관리하는 세 그루의 나무로 구성돼있어요.

첫번째 나무인 Working directory (작업 디렉토리)는 실제 파일들로 이루어져있고,

두번째 나무인 Index (인덱스)는 준비 영역(staging area)의 역할을 하며,

마지막 나무인 HEAD는 최종 확정본(commit)을 나타내요.



## 추가와 확정(commit)

변경된 파일은 아래 명령어로 (**인덱스**에) 추가할 수 있어요.  
git add <파일 이름>  
git add \*  
이것이 바로 git의 기본 작업 흐름에서 첫 단계에 해당돼요.  
하지만 실제로 변경 내용을 확정하려면 아래 명령을 내려야 한답니다.

git commit -m "이번 확정본에 대한 설명"  
자, 이제 변경된 파일이 **HEAD**에 반영됐어요.  
하지만, 원격 저장소에는 아직 반영이 안 됐답니다.

## 변경 내용 발행(push)하기

현재의 변경 내용은 아직 로컬 저장소의 **HEAD** 안에 머물고 있어요.  
이제 이 변경 내용을 원격 서버로 올려봅시다. 아래 명령을 실행하세요.  
git push origin master  
(다른 가지를 발행하려면 *master*를 원하는 가지 이름으로 바꿔주세요.)   
  
만약 기존에 있던 원격 저장소를 복제한 것이 아니라면,  
원격 서버의 주소를 git에게 알려줘야 해요.  
git remote add origin <원격 서버 주소>  
이제 변경 내용을 원격 서버로 발행할 수 있어요.

## 가지(branch)치기

가지는 안전하게 격리된 상태에서 무언가를 만들 때 사용해요.  
여러분이 저장소를 새로 만들면 기본으로 *master 가지*가 만들어집니다.  
이제 다른 가지를 이용해서 개발을 진행하고, 나중에 개발이 완료되면  
master 가지로 돌아와 병합하면 돼요.



아래 명령으로 "feature\_x"라는 이름의 가지를 만들고 갈아탑니다.  
git checkout -b feature\_x  
아래 명령으로 master 가지로 돌아올 수 있어요.  
git checkout master  
아래 명령으로는 가지를 삭제할 수 있어요.  
git branch -d feature\_x  
여러분이 새로 만든 가지를 원격 저장소로 전송하기 전까지는  
*다른 사람들이 접근할 수 없어요*.  
git push origin <가지 이름>

## 갱신과 병합(merge)

여러분의 로컬 저장소를 원격 저장소에 맞춰 갱신하려면  
아래 명령을 실행하세요.  
git pull  
이렇게 하면 원격 저장소의 변경 내용이 로컬 작업 디렉토리에  
*받아지고(fetch)*, *병합(merge)*된답니다.  
다른 가지에 있는 변경 내용을 현재 가지(예를 들면, master 가지)에  
병합하려면 아래 명령을 실행하세요.  
git merge <가지 이름>  
첫번째 명령이든 두번째 명령이든, git은  
자동으로 변경 내용을 병합하려고 시도해요.  
문제는, 항상 성공하는 게 아니라 가끔  
*충돌(conflicts)*이 일어나기도 한다는 거예요.  
이렇게 충돌이 발생하면, git이 알려주는 파일의 충돌 부분을  
여러분이 직접 수정해서 병합이 가능하도록 해야 하죠.  
충돌을 해결했다면, 아래 명령으로 git에게  
아까의 파일을 병합하라고 알려주세요.  
git add <파일 이름>  
변경 내용을 병합하기 전에, 어떻게 바뀌었는지 비교해볼 수도 있어요.  
git diff <원래 가지> <비교 대상 가지>

## 꼬리표(tag) 달기

소프트웨어의 새 버전을 발표할 때마다 꼬리표를 달아놓으면 좋아요.  
(물론 꼬리표는 SVN 등에 이미 존재하는 기능이지요.)  
아래 명령을 실행하면 새로운 꼬리표인 *1.0.0*을 달 수 있어요.  
git tag 1.0.0 1b2e1d63ff  
위 명령에서 *1b2e1d63ff* 부분은 꼬리표가 가리킬 확정본 식별자입니다.  
아래 명령으로 확정본 식별자를 얻을 수 있어요.  
git log  
확정본 식별자의 앞부분 일부만 입력해도 꼬리표를 붙일 수 있지만,  
그 일부분이 반드시 고유하다는 조건이 필요해요.

## 로컬 변경 내용 되돌리기

만약 여러분이 (물론 그럴 일은 없겠지만 ;) 실수로 무언가 잘못한 경우,  
아래 명령으로 로컬의 변경 내용을 되돌릴 수 있어요.  
git checkout -- <파일 이름>  
위 명령은 로컬의 변경 내용을 변경 전 상태(HEAD)로 되돌려줘요.  
다만, 이미 인덱스에 추가된 변경 내용과  
새로 생성한 파일은 그대로 남는답니다.

만약, 로컬에 있는 모든 변경 내용과 확정본을 포기하려면,  
아래 명령으로 원격 저장소의 최신 이력을 가져오고,  
로컬 master 가지가 저 이력을 가리키도록 할 수 있어요.  
git fetch origin  
git reset --hard origin/master

## 유용한 힌트

git의 내장 GUI  
gitk  
콘솔에서 git output을 컬러로 출력하기  
git config color.ui true  
이력(log)에서 확정본 1개를 딱 한 줄로만 표시하기  
git config format.pretty oneline  
파일을 추가할 때 대화식으로 추가하기  
git add -i

# **소스 코드 관리와 git**

소프트웨어 개발에 있어서, 소스코드의 관리는 중요한 포인트 중의 하나이다. 다양한 버전과 변경 관리, 협업을 위해서는 소스코드를 저장 및 관리할 수 있는 시스템이 필요하고, 이를 VCS (Version Control System) 또는 SCM (Source Code Management) System이라고 한다.

## 분산 형상 관리 시스템 Git

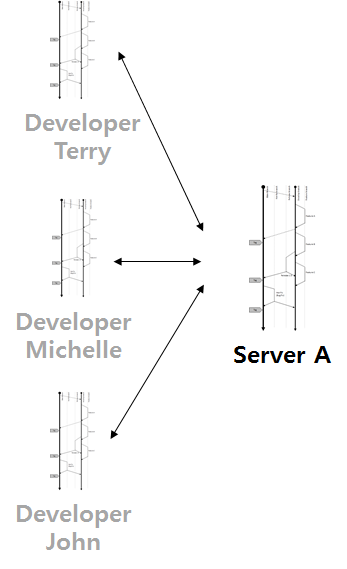
그러면 근래들어 많이 사용되고 있는 VCS 시스템인 Git에 대해서 알아보도록 하자.

Git는 기본적으로 분산 소스코드 관리 시스템이다. 모든 개발자가 중앙 저장소에 붙어서 작업을 하는 것이 아니라. 소스 코드가 여러개의 서버에 다른 복제본으로 존재할 수 있다. 먼저 이 분산 저장소의 개념부터 이해 해보도록 하자

### 중앙 집중형 저장소 (Centralized Version Control System)

중앙 집중형 저장소는, 코드가 저장 서버 단 한군데만 저장된다. 개발자가, 코드를 받아서 수정하고 저장하면, 그 내용이 바로 중앙 저장소에 반영된다 (당연 하지만). 즉 서버에는 항상 마스터 버전(최신 버전)의 소스코드가 저장되어 있다.

  그리고, 서버가 다운 되거나, 네트워크에 접속할 수 없다면 당연히 코드를 commit하거나 최신 코드를 내려받을 수 없다.

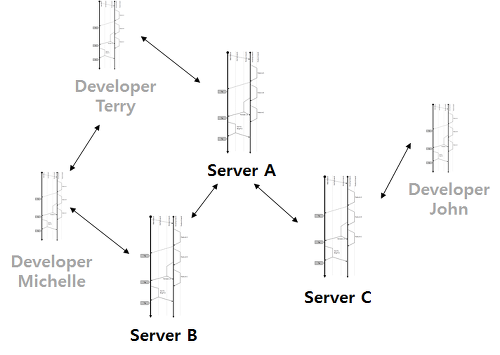


### 분산형 저장소 (Distributed Version Control System)

분산 저장소는 말 그대로, 소스코드가 하나의 중앙 서버가 아니라, 여러개의 서버나 여러개의 개발자 PC에 저장될 수 있으며, 각각이 소스 코드 저장소 (source repository)가 된다. 각 저장소에 저장되는 소스 코드는 같은 버전의 코드가 아니라 제각기 다른 브렌치 코드가 저장된다. 즉 서버 A에는 branch A,B,C 버전이, 서버 B에는 branch A,C,D버전과 같이 다른 브렌치 버전을 저장할 수 있다. 즉 각 저장소에 브렌치 버전이 모두 틀리고, 소스 코드를 access해서 가지고 오는 장소도 모두 다르기 때문에, 시스템 자체에서는 마스터 버전 (최신 버전이 항상 어느 곳에 저장되어 있는가)의 개념이 없다. 예를 들어 설명하자. 아래 그림과 같이 Developer Terry는 Server A에서 코드를 내려 받아서, 데이타 베이스 관련 모듈을 개발하고 있고, Michelle은 Server B에서 UI 관련 모듈을 개발하고 있다. 각자는 개발을 진행하면서, 수시로 각자 Server A와 Server B에 Commit을 하고 있다고 가정하자. Server A는 전체 시스템에서 데이타 베이스 모듈 부분은 가장 최시일테고, Server B에는 UI 모듈의 가장 최신 버전의 코드가 들어가 있을 것이다.

대신 각 모듈의 개발이 끝나면, Server A와 Server B의 코드를 merge하여, 개발 내용을 병합(합칠 수)할 수 있다.

즉 전체 시스템의 최신 소스 코드가 명시적으로 어느 한곳에 저장되어 있지 않는 구조이다.



이러한 형태로, 코드의 여러 버전을 여러 저장소에 분산해서 저장할 수 있기 때문에, 몇가지 장점이 올 수 가 있는데, 팀 단위나 기능 단위로 저장소를 분리해서 개발하거나 릴리즈 버전 단위로 저장소를 분리해서 개발할 수 있는 등 소스코드 버전관리에 많은 유연성을 가지고 있다.

또한 중앙 저장소의 개념이 없기 때문에, 특정 VCS 시스템이 장애가 나더라도, 내가 사용하고 있는 VCS만 문제가 없다면 개발을 계속할 수 있고,앞에서 언급한바와 같이 개발자의 Local PC에 VCS를 설치하여 네트워크 연결이 없는 상태에도 개발을 지속할 수 있다.

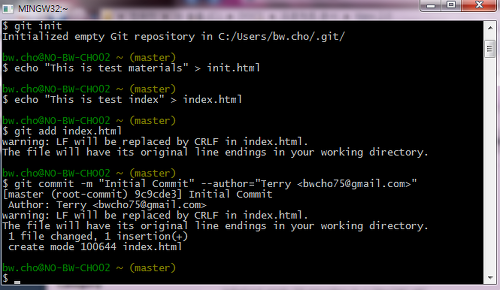
그리고 소스코드가 중앙 서버만이 아니라 여러 서버와 PC에 분산 되서 저장되기 때문에 서버 장애로 저장소가 손상된다고 해도, 다른 서버나 다른PC에서 소스코드와 History들을 모두 저장하고 있기 때문에, 중앙 서버 방식에 비해서 복구가 쉽다.

### Git

DVCS는 Mercurial, Bazzar등 여러가지 제품이 있으나, 근래에는 git 가 가장 많이 사용되고 있다. Linux Kernel 프로젝트나, 안드로이드, Gnome, Ruby on Rails등이 Git를 사용하는 대표적인 프로젝트이다.

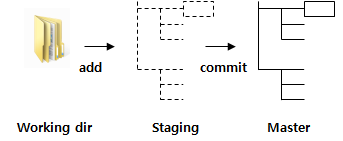
Git는 기존의 VCS에 비해서 설치가 쉽고, 속도가 매우 빠르다. 그리고, branch와 merge가 매우 빠르고 사용이 쉽다. 특히 merge의 경우 누가? 언제? 무엇을 어떤 부분을 merge했는지 까지 상세하게 추적이 가능하기 때문에, 오픈 소스와 같이 대규모 개발자가 동시에 개발을 진행하는 환경에서는 매우 유용하다. 오픈소스 개발자들이 network가 연결되어 있지 않은 상황에서도 언제 어디서나 개발을 할 수 있다. 그리고, 자기만의 개발branch로 개발을 하다가 main branch로 merge도 쉽다. 아마 이런 장점이 git를 major VCS로 만든게 아닌가 싶다.

그러면 간단하게, git의 간략한 기본 사용법을 알아보자.



1)     **Repository 생성 (init과 clone)** - git에서 init 명령은 local에 새롭게 저장소를 만드는 명령이다. 이를 통해서 새롭게 git저장소를 만들 수 있다.  
만약에 원격에 있는 서버의 저장소를 복제해서 로컬에 만들려면  
*git clone 사용자명@서버주소:"저장소 경로"*  
명령어를 사용하면 된다.

2)     **파일 추가**  
다음으로 파일을 추가해보자. 위의 예제에서 echo 명령을 사용하여, init.html과, index.html 파일을 생성하였다. 그리고git에 이 파일이 git에 commit이 될 예정이라고 mark를 해놓는다. 위의 예제에서는 index.html 파일만 먼저 commit을 하도록 한다.  
git에서는 add의 개념을 이해하려면 먼저 staging area라는 개념을 이해해야 한다.



작업 디렉토리 (working dir)에서 작업을 한것은 내 local pc에만 반영된 내용이다. 이 내용을 저장소로 올리기 전에, git는staging(git에서는 index라는 이름으로 사용한다.) 이라는 개념을 제공한다. 소스 코드를 저장소에 최종 반영하기 전에 두 단계를 거치는 two-phase commit을 사용한다.

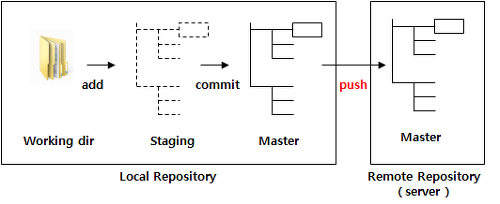
staging이라는 개념은 작업 디렉토리에서 작업한 내용을 반영하기 전에, 최종으로 확인하는 중간 단계 정도로 생각하면 된다. 작업 디렉토리에서 작업한 내용 중 commit할 내용을 미리 "add" 명령어를 통해서 stage에 반영한 후에, stage에 있는 내용을 commit전에, 저장소내의 코드와 비교(diff)하면서, commit을 할 수 있다.

작업 영역과 (working directory)와 stage영역간의 비교는   
*"git diff" 명령어를 통해서 가능하고*  stage와 master 버전에 저장된 코드의 변경 사항은"*git diff --cached"  (또는 git diff --staged)*명령을 이용해서 비교가 가능하다.

작업 디렉토리에서 바뀐 내용을 몽땅 한꺼번에 commit하는 것이 아니라, feature별이나 특정 그룹 (기능이나 FIX별 또는 모듈별)staging에 이동 시킨후, 하나하나 검증하면서 그룹별로 commit이 가능하다.

*3)*     **저장소에 반영 (commit)**  
add를 통해서, 변경본 반영 리스트를 작성하고, diff등을 통해서 확인이 끝나면, 이를 저장소에 반영해야 한다. 반영은commit 명령어를 사용하며, commit에 대한 comment (변경 내용)을 같이 넣는다.  
*git commit -m "변경 내용 description"*

4)     **변경 내용을 원격 저장소에 반영 (push)**  
앞 단계 까지 끝났으면, 소스코드의 변경 내용은 내 local pc에 있는 git 로컬 저장소에 반영되었다. git는 앞서도 설명했지만 분산저장소이기 때문에, commit을 한다고 해서, 서버에 코드가 저장되지 않는다. 기본적으로 commit은 로컬 저장소에 반영이 되지만, 서버의 원격 저장소에 코드를 반영하려면 별도의 반영 작업이 필요하다. 반영은 push 명령어를 사용한다.다음과 같이 push 명령을 사용하여, 원격 서버에 반영한다.



*git push origin "브렌치명"*예) git push origin master (master branch로 push하는 명령)  
*※단 이 경우는 처음에 저장소를 만들때, git clone을 통해서 원격 저장소로 부터 코드를 읽어와서 로컬 저장소를 만들었을 경우이다.*

만약에 원격 저장소로 부터 clone을 해서 만든 경우가 아닐때, 원격 저장소로 code를 밀어 놓고자 한다면, 원격 저장소를 정의해줘야 한다. 원격 저장소를 정의 하는 방법은  
*git remote add "원격저장소명" "원격저장소주소"*  
예) git remote add zipkin https://github.com/twitter/zipkin.git   
zipkin이라는 이름으로 https://github.com/twitter/zipkin.git URL에 있는 원격 저장소를 등록하였다. 여기에 push를 하려면, git push zipkin master (zipkin 원격 저장소 master branch에 push)

분산 저장 VCS 답게, 원격 저장소는 하나가 아닌 여러 개를 git remote add 명령을 통해서 추가 할 수 있고, 등록되어 있는 원격 저장소는 git remote -v 명령을 통해서 조회해볼 수 있다.

5)     **브렌치 관리**  
소스코드 브렌치에 대한 개념은 앞에서 설명하였기 때문에 별도로 설명하지 않고, 명령어 사용법만 설명한다. 현재 코드에서 브렌치를 생성하려면

*git branch "브렌치명" 을 사용한다..*  
예) git branch bugFix  
bugFix라는 이름으로, 현재 코드에서 branch 만들기

그리고 현재 작업중인 브렌치를 이동하려면  
*git checkout "브렌치명" 을 사용한다.*  
예) git checkout master  
master 브렌치로 이동

**6)**   **merge**

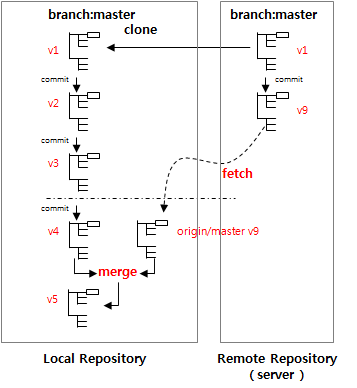
merge는 다른 브렌치의 내용을 현재 작업 중인 브렌치로 합쳐 오는 작업이다.  
예를 들어 내가 master 브렌치에서 작업을 하고 있을때, 예전에 버그 수정을 위해 만들었던 bugFix라는 브렌치의 내용을 현재 브렌치에 반영하고 싶을 경우, master 브렌치에서 "git merge bugFix" 라는 명령을 사용하면 bugFix 브렌치의 변경 내용을 master 브렌치에 반영하게 된다.

이 때, 서로 변경 내용이 다르거나, 같은 코드 라인을 수정하였을 경우 충돌 (conflict)이 발생하는데, 충돌이 발생한 경우,직접 충돌 부분을 수정한 후, git add를 통해서 수정한 파일을 넣고, git commit을 통해서 최종 반영한다.

**7)**   **원격 저장소의 변경 내용을 읽어오기 (pull & fetch)**

반대로, 원격 저장소에서 다른 사람들이 작업했던 내용을 내 로컬저장소로 가지고 오려면 pull 과 fetch 라는 두 가지 방법이 있다.

먼저 fetch ("git fetch")의 경우, 원격 저장소의 업데이트 된 내용을 별도의 브렌치로 읽어 온다. 실제 내가 로컬에서 작업중인 로컬 저장소에는 반영되지 않는다. 반영을 하려면 원격 저장소에서 읽어온 내용을 merge를 통해서 내 작업 영역에 반영해야 한다 이해를 돕기 위해서 아래 그림을 보자.

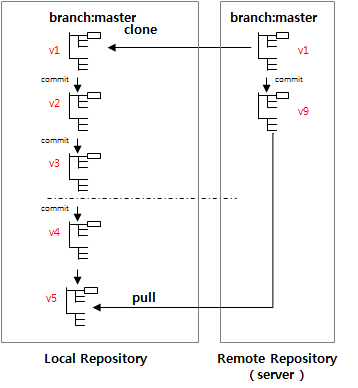


원격 저장소의 v1 버전에서 clone을 받아서 로컬 저장소에서 개발을 시작하였다. 로컬에서 여러번의 commit을 통해서, v4버전까지 개발을 진행하였다.

그 상태에서, 원격 저장소의 변경 내용을 업데이트 하기 위해서 fetch를 하면, 원래 clone을 하였던 원격 저장소의 브렌치(master)의 최신 코드를 로컬로 복사해서 origin/master라는 이름의 브렌치에 업데이트를 한다. 내 작업 영역은 여전히 v4이고, 원격 저장소의 변경 내용은 반영되지 않았다.

이를 반영하려면 "git merge origin/master"를 해주면 merge를 통해서 내 작업 영역에 반영된다. (v5 버전상태)

pull ("git pull")은 한마디로, fetch + merge다. pull 명령어 하나로 자동으로 fetch와 merge를 한꺼번에 해주는 명령어 이다. 아래 그림을 보자, 처음에 원격 저장소에서 v1 버전을 clone으로 내려 받아서 개발을 하다가, v4 버전에서 원격 저장소의 코드를 pull 해주면, 원격 저장소의 clone을 한 원본 저장소와, 로컬의 저장소의 현재 브렌치를 merge하여 새로운 버전으로 만들어 준다.



**8)**   **태깅**

git에서 태깅은 매우 간편하다.

*git tag -a "태그명" -m "태그 설명"*

예) git tag -a "build1109" -m "July.15 빌드 태그".

태깅한 버전으로 이동하려면, 브렌치 이동과 마찬가지로 checkout을 사용하면 된다.

*git checkout "태그명"*

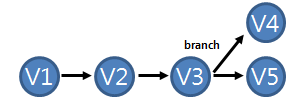
만약 그 태그 버전으로 부터 무엇인가를 작업하려면, 해당 태그로 이동한 후에, git branch를 이용하여, 그 버전에서 부터branch를 따서 작업을 하거나 clone등을 해서 작업을 하는 것이 좋다. 그리고, 태그를 다른 사람과 공유하려면, 브렌치나 코드와 마찬가지로, 태그를 서버로 push해줘야 한다.

*git push origin "태그명"*

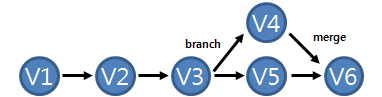
**9)**   **Rebase**

git의 rebase는 git만의 고유 기능이다. merge와 유사한 기능이지만, 코드 변경 History를 조금더 깔끔하게 정리해주는 기능이라고 생각하면 된다.

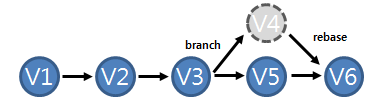
다음과 같은 브렌치가 있다고 하자. V3에서 브렌치를 생성하여 V4를 만들어내었고, master에서는 commit을 진행하여 V5로 진행되었다고 하자. 이 상태에서 V4의 변경 내용을 V5로 합치고자 한다.



일반적인 경우에는 merge를 한다. 아래와 같이 merge를 하는 경우에는 V6 버전이 새로 생기고, V4 branch와 내용과history가 모두 유지 된다.



반면 rebase 명령을 수행할 경우, merge와 마찬가지로, 코드를 master branch로 합치지만, 기존의 V4 branch는 없어지고, V4에 작업된 내용 (변경 History)는 마치 master branch의 변경 history인 것 처럼 하나로 합쳐진다.



**유용한 git 관련 도구들**

git의 경우, 요즘 널리 사용되는 만큼 지원되는 도구들도 많다. 여기서 몇몇 유용한 git 지원 도구들에 대해서 살펴보도록 하자.

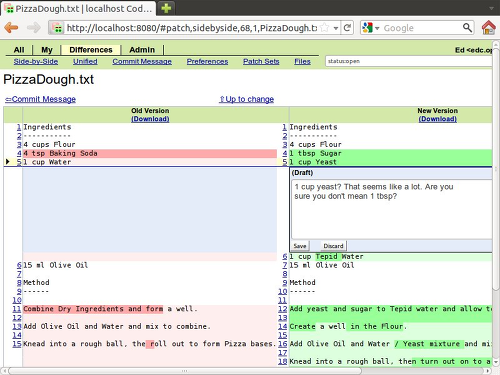
**1)**   **git-gui & gitk**

이 두가지 툴을 gui 툴로, git를 설치하면 기본적으로 설치 된다. git-gui는 git에서 commit을 하는 것을 도와주고, gitk는git 저장소를 gui 기반으로 browsing할 수 있게 해준다.

**2)**   **gerrit**

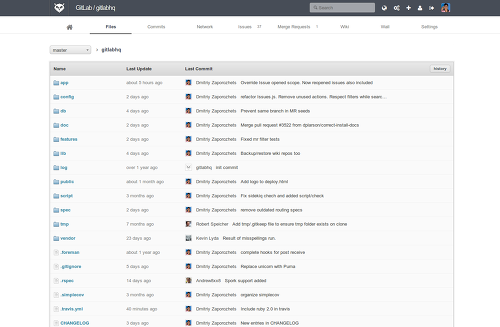
gerrit는 오픈소스 코드리뷰 도구로 (<https://code.google.com/p/gerrit/>) 에서 다운로드 바을 수 있으며, git를 사용시, 웹 기반으로 코드 리뷰를 가능하도록 해준다.

master branch와 dev branch를 갖는 상태에서, dev branch에서 개발을 한 후, dev branch로 push를 하면, jenkins등과 같은 자동 빌드 도구에서 컴파일 및 테스트를 수행한 후에, gerrit을 통해서 코드 리뷰 요청을 생성하고, 코드 리뷰어가 리뷰를 끝내면, 코드를 master로 push하는 시나리오를 만들 수 있다. UI는 좀 떨어지긴 하지만, 저렴한 비용으로 코드리뷰 도구를 사용할 수 있다.



**3)**   **gitLab**

git lab 역시 오픈 소스이다. 코드 리뷰뿐만 아니라, git 저장소에 대한 browsing, 프로젝트 관리, 이슈 관리등 다양한 기능을 제공한다.



지금까지 git에 대해서 가장 기본적인 기능만을 소개하였다. 이외에도 상당히 많은 기능이 있기 때문에, git의 경우에는 꼭 별도의 서적을 참고해 보기를 권장한다.   
※ Pragmatic Version Control Using Git, 저자 Travis Swicegood , ISBN-10: 1934356158  
※ 참고할만한 글

   git에 대해서 간략하게 설명해놓은 문서<http://rogerdudler.github.io/git-guide/index.ko.html>

   git에 대해서 설치 없이 간단하게 배울 수 있는 온라인 튜토리얼   
<http://pcottle.github.io/learnGitBranching/>

사실 git를 사용해보면,그 속도도 빠르고 branch 생성이나 merge도 쉬워서 참 유연한 툴이구나 생각하게 된다. 반대로 생각해보면 유연하다는 것은 사용할 수 있는 용도가 많다는 것이고, 잘 모르면 어떤 방향으로 사용해야 할 지 정하기가 매우 애매하다는 이야기가 된다. svn과 같은 경우는 일반적으로 사용하는 branch 전략이 정해져 있어서 learning curve도 낮아서 쉽게 사용할 수 있지만, svn과 같은 centralized VCS에 익숙해져 있는 사람은 git를 접하면 개념을 잡는데 한참 시간이 소요되다가, 결국은 centralized VCS와 같은 형태로 사용하는 경우가 종종 있다.

 git가 되었건, svn이 되었건 간에, 툴의 사용법 보다는 소스코드의 버전 관리의 본질을 제대로 이해하고, 팀과 프로젝트 성격과 회사내의 릴리즈 정책에 맞는 VCS를 선택하고 그에 맞는 브렌치 전략을 제대로 수립하고 적용해 나가는 것이 더욱 더 중요하다.  특히 조직의 규모가 커지면 커질 수 록, 그리고 릴리즈하는 버전의 다양성이 많을 수 록, 효과적인 브렌치 구조의 정의는 개발 환경의 효율성에 아주 지대한 영향을 미치게 된다.

또한 이 글을 쓰는 내내 머릿속에 멤도는 생각은 수년전만 해도 개발자가 사실 이러한 VCS나 branch 전략에 대해서 크게 신경 쓸 필요가 없었다.일반적인 팀 단위 프로젝트에 들어가서 Check out, update, commit, merge 정도만 잘하면 되었으니까는, 그러나 앞서서 git에서 살펴보았듯이.이미 소스 코드 관리 전략은 오픈소스등의 영향을 많이 받아서 분산형이 대세다. 분산을 하다보니, 상대적으로 다른 개발자나 개발팀의 간섭을 덜 받으면서 별도의 브렌치에서 작업을 해서 코드 작업을 할 수 있다는 장점도 있지만, 그만큼 개발자가 이제는 코드 관리에 대한 깊숙한 이해를 가져야 한다는 것이다.

# **1.5 시작하기 - Git 최초 설정**

### [설정 확인](http://git-scm.com/book/ko/v1/%EC%8B%9C%EC%9E%91%ED%95%98%EA%B8%B0-Git-%EC%B5%9C%EC%B4%88-%EC%84%A4%EC%A0%95#설정-확인)

$ git config --list 명령을 실행하면 설정한 모든 것을 보여준다:

Git은 같은 키를 여러 파일(/etc/gitconfig와 ~/.gitconfig 같은)에서 읽기 때문에 같은 키가 여러개 있을 수도 있다. 이러면 Git은 나중 값을 사용한다.

git config {key} 명령으로 Git이 특정 Key에 대해 어떤 값을 사용하는지 확인할 수 있다:

$ git config user.name

Scott Chacon

도움말 보기

명령어에 대한 도움말이 필요할 때 도움말을 보는 방법은 세 가지다:

$ git help <verb>

$ git <verb> --help

$ man git-<verb>

# **2.1 Git의 기초 - Git 저장소 만들기**

Git을 사용하는 방법을 알고 싶은데 한 챕터밖에 읽을 시간이 없다면 이번 챕터를 읽어야 한다. Git에서 자주 사용하는 명령어는 모두 2장에 등장한다. 2장을 다 읽으면 저장소를 만들고 설정하는 방법, 파일을 추적하거나(Track) 추적을 그만두는 방법, 변경 내용을 Stage 하고 커밋하는 방법을 알게 된다. 파일이나 파일 패턴을 무시하도록 Git을 설정하는 방법, 실수를 쉽고 빠르게 만회하는 방법, 프로젝트 히스토리를 조회하고 커밋을 비교하는 방법, 리모트 저장소에 Push 하고 Pull 하는 방법을 살펴본다.

## Git 저장소 만들기

Git 저장소를 만드는 방법은 두 가지다. 기존 프로젝트를 Git 저장소로 만드는 방법이 있고, 다른 서버에 있는 저장소를 Clone 하는 방법이 있다.

### [기존 디렉토리를 Git 저장소로 만들기](https://git-scm.com/book/ko/v2/Git%EC%9D%98-%EA%B8%B0%EC%B4%88-Git-%EC%A0%80%EC%9E%A5%EC%86%8C-%EB%A7%8C%EB%93%A4%EA%B8%B0#기존-디렉토리를-Git-저장소로-만들기)

기존 프로젝트를 Git으로 관리하고 싶을 때, 프로젝트의 디렉토리로 이동해서 아래과 같은 명령을 실행한다.

**$** git init

이 명령은 .git 이라는 하위 디렉토리를 만든다. .git 디렉토리에는 저장소에 필요한 뼈대 파일(Skeleton)이 들어 있다. 이 명령만으로는 아직 프로젝트의 어떤 파일도 관리하지 않는다. (.git 디렉토리가 막 만들어진 직후에 정확히 어떤 파일이 있는지에 대한 내용은 [Chapter 10](https://git-scm.com/book/ko/v2/ch10/_git_internals)에서 다룬다)

저장소를 초기화 하기 전에 여러분의 현재 작업 디렉토리가 어디인지 꼭 확인하세요. 맥OS나 Linux라면 PWD, 윈도우라면 CD 명령어를 실행하면 알 수 있을 거에요. 반드시 해당 폴더에서 해야합니다.

$ git init

Initialized empty Git repository in ~/djangogirls/.git/

$ git config --global user.name "ktlee88"

$ git config --global user.email ktlee88@yahoo.com

git 저장소 초기화는 프로젝트를 시작할 때 딱 한번만 필요합니다. (또한 두 번째, 세 번째 명령인 username과 email 등록 명령은 계속 적용되기 때문에 이 계정에서는 다시 입력할 필요가 없습니다.)

\*note -- error: could not lock config file C:\/.gitconfig: No such file or directory

To fix it, You can simply remove the home from environment variables (My Computer (right-click) > Properties > Advanced (tab) > Environment Variables (under System Variables)

Git이 파일을 관리하게 하려면 저장소에 파일을 추가하고 커밋해야 한다. git add 명령으로 파일을 추가하고 git commit 명령으로 커밋한다:

**$** git add \*.c

**$** git add LICENSE

**$** git commit -m 'initial project version'

명령어를 몇개로 순식간에 Git 저장소를 만들고 파일이 관리되게 했다.

### [기존 저장소를 Clone 하기](https://git-scm.com/book/ko/v2/Git%EC%9D%98-%EA%B8%B0%EC%B4%88-Git-%EC%A0%80%EC%9E%A5%EC%86%8C-%EB%A7%8C%EB%93%A4%EA%B8%B0#기존-저장소를-Clone-하기)

다른 프로젝트에 참여하려거나(Contribute) Git 저장소를 복사하고 싶을 때 git clone 명령을 사용한다. 이미 Subversion 같은 VCS에 익숙한 사용자에게는 “checkout” 이 아니라 “clone” 이라는 점이 도드라져 보일 것이다. Git이 Subversion과 다른 가장 큰 차이점은 서버에 있는 거의 모든 데이터를 복사한다는 것이다. git clone을 실행하면 프로젝트 히스토리를 전부 받아온다. 실제로 서버의 디스크가 망가져도 클라이언트 저장소 중에서 아무거나 하나 가져다가 복구하면 된다(서버에만 적용했던 설정은 복구하지 못하지만 모든 데이터는 복구된다 - [“서버에 Git 설치하기”](https://git-scm.com/book/ko/v2/ch04/_git_on_the_server)에서 좀 더 자세히 다룬다).

git clone [url] 명령으로 저장소를 Clone 한다. libgit2 라이브러리 소스코드를 Clone 하려면 아래과 같이 실행한다.

**$** git clone https://github.com/libgit2/libgit2

이 명령은 “libgit2”이라는 디렉토리를 만들고 그 안에 .git 디렉토리를 만든다. 그리고 저장소의 데이터를 모두 가져와서 자동으로 가장 최신 버전을 Checkout 해 놓는다. libgit2 디렉토리로 이동하면 Checkout으로 생성한 파일을 볼 수 있고 당장 하고자 하는 일을 시작할 수 있다. 아래과 같은 명령을 사용하여 저장소를 Clone 하면 “libgit2”이 아니라 다른 디렉토리 이름으로 Clone 할 수 있다.

**$** git clone https://github.com/libgit2/libgit2 mylibgit

디렉토리 이름이 mylibgit 이라는 것만 빼면 이 명령의 결과와 앞선 명령의 결과는 같다.

Git은 다양한 프로토콜을 지원한다. 이제까지는 https:// 프로토콜을 사용했지만 git://를 사용할 수도 있고 user@server:path/to/repo.git 처럼 SSH 프로토콜을 사용할 수도 있다. 자세한 내용은 [“서버에 Git 설치하기”](https://git-scm.com/book/ko/v2/ch04/_git_on_the_server)에서 다루며 각 프로토콜의 장단점과 Git 저장소에 접근하는 방법을 설명한다.

# **2.2 Git의 기초 - 수정하고 저장소에 저장하기**

## 수정하고 저장소에 저장하기

만질 수 있는 Git 저장소를 하나 만들었고 워킹 디렉토리에 Checkout도 했다. 이제는 파일을 수정하고 파일의 스냅샷을 커밋해 보자. 파일을 수정하다가 저장하고 싶으면 스냅샷을 커밋한다.

워킹 디렉토리의 모든 파일은 크게 Tracked(관리대상임)와 Untracked(관리대상이 아님)로 나눈다. Tracked파일은 이미 스냅샷에 포함돼 있던 파일이다. Tracked 파일은 또 Unmodified(수정하지 않음)와 Modified(수정함) 그리고 Staged(커밋하면 저장소에 기록되는) 상태 중 하나이다. 그리고 나머지 파일은 모두 Untracked파일이다. Untracked 파일은 워킹 디렉토리에 있는 모든 파일이 스냅샷에 포함돼 있는 것은 아니고 Staging Area에 있는 것도 아니다. 처음 저장소를 Clone하면 모든 파일은 Tracked이면서 Unmodified 상태가 된다. 파일을 Checkout하고 나서 아무것도 수정하지 않았기 때문에 그렇다.

마지막 커밋 이후 아직 아무것도 수정하지 않은 상태에서 어떤 파일이 수정되면 Git은 그 즉시 파일을Modified 상태로 인식한다. 그리고 이 수정한 파일을 Stage하고 Staged 상태인 파일을 커밋한다. 이 라이프사이클을 그림 2-1처럼 계속 반복한다.



그림 2-1. 파일의 라이프사이클

### [파일의 상태 확인하기](https://git-scm.com/book/ko/v1/Git%EC%9D%98-%EA%B8%B0%EC%B4%88-%EC%88%98%EC%A0%95%ED%95%98%EA%B3%A0-%EC%A0%80%EC%9E%A5%EC%86%8C%EC%97%90-%EC%A0%80%EC%9E%A5%ED%95%98%EA%B8%B0#파일의-상태-확인하기)

파일의 상태를 확인하려면 보통 git status 명령을 사용한다. Clone한 후에 바로 이 명령을 실행하면 아래과 같은 메시지를 볼 수 있다:

$ git status

On branch master

nothing to commit, working directory clean

위의 내용은 파일을 하나도 수정하지 않았다는 것을 말해준다. Tracked나 Modified 상태인 파일이 없다는 의미다. Untracked 파일은 아직 없어서 목록에 나타나지 않는다. 그리고 현재 작업 중인 브랜치를 알려준다. 기본 브랜치가 master이기 때문에 현재 master로 나오는 것이다. 브랜치 관련 내용은 차차 알아가자. 다음 장에서 브랜치와 레퍼런스에 대해 자세히 다룬다.

프로젝트에 README 파일을 만들어보자. README 파일은 새로 만든 파일이기 때문에 git status를 실행하면 'Untracked files'에 들어 있다:

$ vim README

$ git status

On branch master

Untracked files:

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

README

nothing added to commit but untracked files present (use "git add" to track)

README 파일은 Untracked files 부분에 속해 있는데 이것은 README 파일이 Untracked 상태라는 것을 말한다. Git은 Untracked 파일을 아직 스냅샷(커밋)에 넣어지지 않은 파일이라고 본다. 파일이 Tracked 상태가 되기 전까지는 Git은 절대 그 파일을 커밋하지 않는다. 그래서 일하면서 생성하는 바이너리 파일 같은 것을 커밋하는 실수는 하지 않게 된다. README 파일을 추가해서 직접 Tracked 상태로 만들어 보자.

### [파일을 새로 추적하기](https://git-scm.com/book/ko/v1/Git%EC%9D%98-%EA%B8%B0%EC%B4%88-%EC%88%98%EC%A0%95%ED%95%98%EA%B3%A0-%EC%A0%80%EC%9E%A5%EC%86%8C%EC%97%90-%EC%A0%80%EC%9E%A5%ED%95%98%EA%B8%B0#파일을-새로-추적하기)

git add 명령으로 파일을 새로 추적할 수 있다. 아래 명령을 실행하면 Git은 README 파일을 추적한다:

$ git add README

git status 명령을 다시 실행하면 README 파일이 Tracked 상태이면서 Staged 상태라는 것을 확인할 수 있다:

$ git status

On branch master

Changes to be committed:

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

new file: README

'Changes to be committed' 에 들어 있는 파일은 Staged 상태라는 것을 의미한다. 커밋하면 git add를 실행한 시점의 파일이 커밋되어 저장소 히스토리에 남는다. 앞에서 git init 명령을 실행했을 때, 그 다음 git add (files) 명령을 실행했던 걸 기억할 것이다. 이것은 작업 디렉토리에 있는 파일들을 추적하기 시작하게 하였다. git add 명령은 파일 또는 디렉토리의 경로명을 아규먼트로 받는다; 만일 디렉토리를 아규먼트로 줄 경우, 그 디렉토리 아래에 있는 모든 파일들을 재귀적으로 추가한다.

### [Modified 상태의 파일을 Stage하기](https://git-scm.com/book/ko/v1/Git%EC%9D%98-%EA%B8%B0%EC%B4%88-%EC%88%98%EC%A0%95%ED%95%98%EA%B3%A0-%EC%A0%80%EC%9E%A5%EC%86%8C%EC%97%90-%EC%A0%80%EC%9E%A5%ED%95%98%EA%B8%B0#Modified-상태의-파일을-Stage하기)

이미 Tracked 상태인 파일을 수정하는 법을 알아보자. benchmarks.rb라는 파일을 수정하고 나서 git status 명령을 다시 실행하면 결과는 아래와 같다:

$ git status

On branch master

Changes to be committed:

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

new file: README

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: benchmarks.rb

이 benchmarks.rb 파일은 Changes not staged for commit에 있다. 이것은 수정한 파일이 Tracked 상태이지만 아직 Staged 상태는 아니라는 것이다. Staged 상태로 만들려면 git add 명령을 실행해야 한다.git add는 파일을 새로 추적할 때도 사용하고 수정한 파일을 Staged 상태로 만들 때도 사용한다. git add를 실행하여 benchmarks.rb 파일을 Staged 상태로 만들고 git status 명령으로 결과를 확인해보자:

$ git add benchmarks.rb

$ git status

On branch master

Changes to be committed:

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

new file: README

modified: benchmarks.rb

두 파일 모두 Staged 상태이므로 다음 커밋에 포함된다. 하지만, 아직 더 수정해야 한다는 것을 알게 되어 바로 커밋하지 못하는 상황이 되었다고 하자. 이 상황에서 benchmark.rb 파일을 열고 수정한다. 아마 당신은 커밋할 준비가 다 됐다고 생각할 테지만, Git은 그렇지 않다. git status 명령으로 파일의 상태를 다시 확인해보자:

$ vim benchmarks.rb

$ git status

On branch master

Changes to be committed:

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

new file: README

modified: benchmarks.rb

Changes not staged for commit:

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

modified: benchmarks.rb

헉! benchmarks.rb가 Staged 상태이면서 동시에 Unstaged 상태로 나온다. 어떻게 이런 일이 가능할까? git add 명령을 실행하면 Git은 파일을 바로 Staged 상태로 만든다. 지금 이 시점에서 커밋을 하면 git commit명령을 실행하는 시점의 버전이 커밋되는 것이 아니라 마지막으로 git add 명령을 실행했을 때의 버전이 커밋된다. 그러니까 git add 명령을 실행한 후에 또 파일을 수정하면 git add 명령을 다시 실행해서 최신 버전을 Staged 상태로 만들어야 한다:

$ git add benchmarks.rb

$ git status

On branch master

Changes to be committed:

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

new file: README

modified: benchmarks.rb

### [파일 무시하기](https://git-scm.com/book/ko/v1/Git%EC%9D%98-%EA%B8%B0%EC%B4%88-%EC%88%98%EC%A0%95%ED%95%98%EA%B3%A0-%EC%A0%80%EC%9E%A5%EC%86%8C%EC%97%90-%EC%A0%80%EC%9E%A5%ED%95%98%EA%B8%B0#파일-무시하기)

어떤 파일은 Git이 자동으로 추가하거나 Untracked 파일이라고 보여줄 필요가 없다. 보통 로그 파일이나 빌드 시스템이 자동으로 생성한 파일이 그렇다. 그런 파일을 무시하려면 .gitignore 파일을 만들고 그 안에 무시할 파일 패턴을 적는다. 아래는 .gitignore 파일의 예이다:

$ cat .gitignore

\*.[oa]

\*~

첫번째 줄은 확장자가 .o 나 .a인 파일을 Git이 무시하라는 것이고 둘째 줄은 ~로 끝나는 모든 파일을 무시하라는 것이다. .o와 .a는 각각 빌드 시스템이 만들어내는 오브젝트와 아카이브 파일이고 ~로 끝나는 파일은 Emacs나 VI 같은 텍스트 편집기가 임시로 만들어내는 파일이다. 또 log, tmp, pid 같은 디렉토리나, 자동으로 생성하는 문서 같은 것들도 추가할 수 있다. .gitignore 파일은 보통 처음에 만들어 두는 것이 편리하다. 그래서 Git 저장소에 커밋하고 싶지 않은 파일을 실수로 커밋하는 일을 방지할 수 있다.

.gitignore 파일에 입력하는 패턴은 아래 규칙을 따른다:

* 아무것도 없는 줄이나, #로 시작하는 줄은 무시한다.
* 표준 Glob 패턴을 사용한다.
* 디렉토리는 슬래시(/)를 끝에 사용하는 것으로 표현한다.
* 느낌표(!)로 시작하는 패턴의 파일은 무시하지 않는다.

Glob 패턴은 정규표현식을 단순하게 만든 것으로 생각하면 되고 보통 쉘에서 많이 사용한다. 애스터리스크(\*)는 문자가 하나도 없거나 하나 이상을 의미하고, [abc]는 중괄호 안에 있는 문자 중 하나를 의미한다(그러니까 이 경우에는 a, b, c). 물음표(?)는 문자 하나를 말하고, [0-9]처럼 중괄호 안의 캐릭터 사이에 하이픈(-)을 사용하면 그 캐릭터 사이에 있는 문자 하나를 말한다.

다음은 .gitignore 파일의 예이다:

# a comment - 이 줄은 무시한다.

# 확장자가 .a인 파일 무시

\*.a

# 윗 줄에서 확장자가 .a인 파일은 무시하게 했지만 lib.a는 무시하지 않는다.

!lib.a

# 루트 디렉토리에 있는 TODO파일은 무시하고 subdir/TODO처럼 하위디렉토리에 있는 파일은 무시하지 않는다.

/TODO

# build/ 디렉토리에 있는 모든 파일은 무시한다.

build/

# `doc/notes.txt`같은 파일은 무시하고 doc/server/arch.txt같은 파일은 무시하지 않는다.

doc/\*.txt

# `doc` 디렉토리 아래의 모든 .txt 파일을 무시한다.

doc/\*\*/\*.txt

\*\*/ 스타일의 문법은 Git 1.8.2 버전부터 사용할 수 있다.

### [Staged와 Unstaged 상태의 변경 내용을 보기](https://git-scm.com/book/ko/v1/Git%EC%9D%98-%EA%B8%B0%EC%B4%88-%EC%88%98%EC%A0%95%ED%95%98%EA%B3%A0-%EC%A0%80%EC%9E%A5%EC%86%8C%EC%97%90-%EC%A0%80%EC%9E%A5%ED%95%98%EA%B8%B0#Staged와-Unstaged-상태의-변경-내용을-보기)

단순히 파일이 변경됐다는 사실이 아니라 어떤 내용이 변경됐는지 살펴보기엔 git status 명령이 아니라git diff 명령을 사용해야 한다. 보통우리는 '수정했지만, 아직 Staged 파일이 아닌것?'과 '어떤 파일이 Staged 상태인지?'가 궁금하기 때문에 git status 명령으로도 충분하다. git diff는 Patch처럼 어떤 라인을 추가했고 삭제했는지가 궁금할 때에 사용한다. git diff는 나중에 더 자세히 다룬다.

README 파일을 수정해서 Staged 상태로 만들고 benchmarks.rb 파일은 그냥 수정만 해둔다. 이 상태에서git status 명령을 실행하면 아래와 같은 메시지를 볼 수 있다:

$ git status

On branch master

Changes to be committed:

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

new file: README

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: benchmarks.rb

git diff 명령을 실행하면 수정했지만 아직 staged 상태가 아닌 파일을 비교해 볼 수 있다:

$ git diff

diff --git a/benchmarks.rb b/benchmarks.rb

index 3cb747f..da65585 100644

--- a/benchmarks.rb

+++ b/benchmarks.rb

@@ -36,6 +36,10 @@ def main

@commit.parents[0].parents[0].parents[0]

end

+ run\_code(x, 'commits 1') do

+ git.commits.size

+ end

+

run\_code(x, 'commits 2') do

log = git.commits('master', 15)

log.size

이 명령은 워킹 디렉토리에 있는 것과 Staging Area에 있는 것을 비교한다. 그래서 수정하고 아직 Stage하지 않은 것을 보여준다.

만약 커밋하려고 Staging Area에 넣은 파일의 변경 부분을 보고 싶으면 git diff --cached 옵션을 사용한다(Git 버전 1.6.1부터는 좀 더 기억하기 쉽게 git diff --staged로도 사용할 수 있다). 이 명령은 저장소에 커밋한 것과 Staging Area에 있는 것을 비교한다:

$ git diff --cached

diff --git a/README b/README

new file mode 100644

index 0000000..03902a1

--- /dev/null

+++ b/README2

@@ -0,0 +1,5 @@

+grit

+ by Tom Preston-Werner, Chris Wanstrath

+ http://github.com/mojombo/grit

+

+Grit is a Ruby library for extracting information from a Git repository

**꼭 잊지 말아야 할 것이 있는데 git diff 명령은 마지막으로 커밋한 후에 수정한 것들 전부를 보여주지 않는다. git diff는 Unstaged 상태인 것들만 보여준다. 이 부분이 조금 헷갈릴 수 있다. 수정한 파일을 모두 Staging Area에 넣었다면 git diff 명령은 아무것도 출력하지 않는다.**

benchmarks.rb 파일을 Stage한 후에 다시 수정해도 git diff 명령을 사용할 수 있다. 이때는 Staged 상태인 것과 Unstaged 상태인 것을 비교한다:

$ git add benchmarks.rb

$ echo '# test line' >> benchmarks.rb

$ git status

On branch master

Changes to be committed:

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

modified: benchmarks.rb

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: benchmarks.rb

git diff 명령으로 Unstaged 상태인 변경 부분을 확인해 볼 수 있다:

$ git diff

diff --git a/benchmarks.rb b/benchmarks.rb

index e445e28..86b2f7c 100644

--- a/benchmarks.rb

+++ b/benchmarks.rb

@@ -127,3 +127,4 @@ end

main()

##pp Grit::GitRuby.cache\_client.stats

+# test line

Staged 상태인 파일은 git diff --cached 옵션으로 확인한다:

$ git diff --cached

diff --git a/benchmarks.rb b/benchmarks.rb

index 3cb747f..e445e28 100644

--- a/benchmarks.rb

+++ b/benchmarks.rb

@@ -36,6 +36,10 @@ def main

@commit.parents[0].parents[0].parents[0]

end

+ run\_code(x, 'commits 1') do

+ git.commits.size

+ end

+

run\_code(x, 'commits 2') do

log = git.commits('master', 15)

log.size

### [변경사항 커밋하기](https://git-scm.com/book/ko/v1/Git%EC%9D%98-%EA%B8%B0%EC%B4%88-%EC%88%98%EC%A0%95%ED%95%98%EA%B3%A0-%EC%A0%80%EC%9E%A5%EC%86%8C%EC%97%90-%EC%A0%80%EC%9E%A5%ED%95%98%EA%B8%B0#변경사항-커밋하기)

수정한 것을 커밋하기 위해 Staging Area에 파일을 정리했다. Unstaged 상태의 파일은 커밋되지 않는다는 것을 기억해야 한다. Git은 생성하거나 수정하고 나서 git add 명령으로 추가하지 않은 파일은 커밋하지 않는다. 그 파일은 여전히 Modified 상태로 남아 있다. 커밋하기 전에 git status 명령으로 모든 것이 Staged 상태인지 확인할 수 있다. 그리고 git commit을 실행하여 커밋한다:

$ git commit

Git 설정에 지정된 편집기가 실행되고, 아래와 같은 텍스트가 자동으로 포함된다(아래 예제는 Vim 편집기의 화면이다). 이 편집기는 쉘의 $EDITOR 환경 변수에 등록된 편집기이고 보통은 Vim이나 Emacs을 사용한다. 또 1장에서 설명했듯이 git config --global core.editor 명령으로 어떤 편집기를 사용할지 설정할 수 있다:

편집기는 아래와 같은 내용을 표시한다(아래 예제는 Vim 편집기):

# Please enter the commit message for your changes. Lines starting

# with '#' will be ignored, and an empty message aborts the commit.

# On branch master

# Changes to be committed:

# new file: README

# modified: benchmarks.rb

#

~

~

~

".git/COMMIT\_EDITMSG" 10L, 283C

자동으로 생성되는 커밋 메시지의 첫 줄은 비어 있고 둘째 줄부터 git status 명령의 결과가 채워진다. 커밋한 내용을 쉽게 기억할 수 있도록 이 메시지를 포함할 수도 있고 메시지를 전부 지우고 새로 작성할 수 있다(수정한 내용을 좀 더 구체적으로 남겨 둘 수 있다. git commit에 -v 옵션을 추가하면 편집기에 diff 메시지도 추가된다).

메시지를 인라인으로 첨부할 수도 있다. commit 명령을 실행할 때 아래와 같이 -m 옵션을 사용한다:

$ git commit -m "Story 182: Fix benchmarks for speed"

[master 463dc4f] Story 182: Fix benchmarks for speed

2 files changed, 3 insertions(+)

create mode 100644 README

commit 명령은 몇 가지 정보를 출력하는데 위 예제는 master 브랜치에 커밋했고 체크섬은 463dc4f이라고 알려준다. 그리고 수정한 파일이 몇 개이고 삭제됐거나 추가된 줄이 몇 줄인지 알려준다.

Git은 Staging Area에 속한 스냅샷을 커밋한다는 것을 기억해야 한다. 수정은 했지만, 아직 Staging Area에 넣지 않은 것은 다음에 커밋할 수 있다. 커밋할 때마다 프로젝트의 스냅샷을 기록하기 때문에 나중에 스냅샷끼리 비교하거나 예전 스냅샷으로 되돌릴 수 있다.

### [Staging Area 생략하기](https://git-scm.com/book/ko/v1/Git%EC%9D%98-%EA%B8%B0%EC%B4%88-%EC%88%98%EC%A0%95%ED%95%98%EA%B3%A0-%EC%A0%80%EC%9E%A5%EC%86%8C%EC%97%90-%EC%A0%80%EC%9E%A5%ED%95%98%EA%B8%B0#Staging-Area-생략하기)

Staging Area는 커밋할 파일을 정리한다는 점에서 매우 유용하지만 복잡하기만 하고 필요하지 않은 때도 있다. 아주 쉽게 Staging Area를 생략할 수 있다. git commit 명령을 실행할 때 -a 옵션을 추가하면 Git은 Tracked 상태의 파일을 자동으로 Staging Area에 넣는다. 그래서 git add 명령을 실행하는 수고를 덜 수 있다:

$ 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: benchmarks.rb

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

$ git commit -a -m 'added new benchmarks'

[master 83e38c7] added new benchmarks

1 files changed, 5 insertions(+)

이 예제에서는 커밋하기 전에 git add 명령으로 benchmarks.rb 파일을 추가하지 않았다는 점을 눈여겨보자.

### [파일을 삭제하기](https://git-scm.com/book/ko/v1/Git%EC%9D%98-%EA%B8%B0%EC%B4%88-%EC%88%98%EC%A0%95%ED%95%98%EA%B3%A0-%EC%A0%80%EC%9E%A5%EC%86%8C%EC%97%90-%EC%A0%80%EC%9E%A5%ED%95%98%EA%B8%B0#파일을-삭제하기)

Git에서 파일을 제거하려면 git rm 명령으로 Tracked 상태의 파일을 삭제한 후에(정확하게는 Staging Area에서 삭제하는 것) 커밋해야 한다. 이 명령은 워킹 디렉토리에 있는 파일도 삭제하기 때문에 실제로 지워진다.

만약 Git없이 그냥 파일을 삭제하고 git status 명령으로 상태를 확인하면 Changes not staged for commit(즉, Unstaged) 에 속한다는 것을 확인할 수 있다:

$ rm grit.gemspec

$ git status

On branch master

Changes not staged for commit:

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

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

deleted: grit.gemspec

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

그리고 git rm 명령을 실행하면 삭제한 파일은 staged 상태가 된다:

$ git rm grit.gemspec

rm 'grit.gemspec'

$ git status

On branch master

Changes to be committed:

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

deleted: grit.gemspec

커밋하면 파일은 삭제되고 Git은 이 파일을 더는 추적하지 않는다. 이미 파일을 수정했거나 Index에(역주, Staging Area을 Git Index라고도 부른다) 추가했다면 -f옵션을 주어 강제로 삭제해야 한다. 이 점은 실수로 데이터를 삭제하지 못하도록 하는 안전장치다. 한 번도 커밋한적 없는 데이터는 Git으로 복구할 수 없다.

또 Staging Area에서만 제거하고 워킹 디렉토리에 있는 파일은 지우지 않고 남겨둘 수 있다. 다시 말해서 하드디스크에 있는 파일은 그대로 두고 Git만 추적하지 않게 한다. 이것은 .gitignore 파일에 추가하는 것을 빼먹었거나 대용량 로그 파일이나 컴파일된 파일인 .a 파일 같은 것을 실수로 추가했을 때 쓴다. --cached 옵션을 사용하여 명령을 실행한다:

$ git rm --cached readme.txt

여러 개의 파일이나 디렉토리를 한꺼번에 삭제할 수도 있다. 아래와 같이 git rm 명령에 file-glob 패턴을 사용한다:

$ git rm log/\\*.log

\*앞에 \을 사용한 것을 기억하자. 파일명 확장 기능은 쉘에만 있는 것이 아니라 Git 자체에도 있기 때문에 필요하다. Windows 기본 쉘을 쓸 때는 \ 기호를 붙이지 않는다. 이 명령은 log/ 디렉토리에 있는 .log 파일을 모두 삭제한다. 아래의 예제처럼 할 수도 있다:

$ git rm \\*~

이 명령은 ~로 끝나는 파일을 모두 삭제한다.

### [파일 이름 변경하기](https://git-scm.com/book/ko/v1/Git%EC%9D%98-%EA%B8%B0%EC%B4%88-%EC%88%98%EC%A0%95%ED%95%98%EA%B3%A0-%EC%A0%80%EC%9E%A5%EC%86%8C%EC%97%90-%EC%A0%80%EC%9E%A5%ED%95%98%EA%B8%B0#파일-이름-변경하기)

Git은 다른 VCS 시스템과는 달리 파일 이름의 변경이나 파일의 이동을 명시적으로 관리하지 않는다. 다시 말해서 파일 이름이 변경됐다는 별도의 정보를 저장하지 않는다. Git은 똑똑해서 굳이 파일 이름이 변경되었다는 것을 추적하지 않아도 아는 방법이 있다. 파일의 이름이 변경된 것을 Git이 어떻게 알아내는지 살펴보자.

이렇게 말하고 Git에 mv 명령이 있는 게 좀 이상하겠지만, 아래와 같이 파일이름을 변경할 수 있다:

$ git mv file\_from file\_to

잘 동작한다. 이 명령을 실행하고 Git의 상태를 확인해보면 Git은 이름이 바뀐 사실을 알고 있다:

$ git mv README.txt README

$ git status

On branch master

Changes to be committed:

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

renamed: README.txt -> README

사실 git mv 명령은 아래 명령어들을 수행한 것과 완전히 똑같다:

$ mv README.txt README

$ git rm README.txt

$ git add README

git mv는 일종의 단축 명령어이다. 이 명령으로 파일이름을 바꿔도 되고 mv 명령으로 파일이름을 직접 바꿔도 된다. 단지 Git의 mv명령은 편리하게 명령을 세 번 실행해주는 것뿐이다. 어떤 도구로 이름을 바꿔도 상관없다. 중요한 것은 이름을 변경하고 나서 꼭 rm/add 명령을 실행해야 한다는 것뿐이다.

# **2.4 Git의 기초 - 되돌리기**

## 되돌리기

일을 하다보면 모든 단계에서 어떤 것은 되돌리고(Undo) 싶을 때가 있다. 이번에는 우리가 한 일을 되돌리는 방법을 살펴볼 것이다. 한 번 되돌리면 복구할 수 없어서 주의해야 한다. Git을 사용하면 우리가 한 실수를 복구하지 못할 것은 거의 없지만 되돌리기는 복구할 수 없다.

### [커밋 수정하기](http://git-scm.com/book/ko/v1/Git%EC%9D%98-%EA%B8%B0%EC%B4%88-%EB%90%98%EB%8F%8C%EB%A6%AC%EA%B8%B0#커밋-수정하기)

종종 완료한 커밋을 수정해야 할 때가 있다. 너무 일찍 커밋했거나 어떤 파일을 빼먹었을 때 그리고 커밋 메시지를 잘못 적었을 때 하게 된다. 다시 커밋하고 싶으면 --amend 옵션을 사용한다:

$ git commit --amend

이 명령은 Staging Area를 사용하여 커밋한다. 만약 마지막으로 커밋하고 나서 수정한 것이 없다면(커밋하자마자 바로 이 명령을 실행하는 경우) 조금 전에 한 커밋과 모든 것이 같다. 이때는 커밋 메시지만 수정한다.

편집기가 실행되면 이전 커밋 메시지가 자동으로 포함된다. 메시지를 수정하지 않고 그대로 커밋해도 기존의 커밋을 덮어쓴다.

커밋을 했는데 Stage하는 것을 깜빡하고 빠트린 파일이 있으면 아래와 같이 고칠 수 있다:

$ git commit -m 'initial commit'

$ git add forgotten\_file

$ git commit --amend

여기서 실행한 명령어 3개는 모두 하나의 커밋으로 기록된다. 두 번째 커밋은 첫 번째 커밋을 덮어쓴다.

### [파일 상태를 Unstage로 변경하기](http://git-scm.com/book/ko/v1/Git%EC%9D%98-%EA%B8%B0%EC%B4%88-%EB%90%98%EB%8F%8C%EB%A6%AC%EA%B8%B0#파일-상태를-Unstage로-변경하기)

다음은 Staging Area와 워킹 디렉토리 사이를 넘나드는 방법을 설명한다. 두 영역의 상태를 확인할 때마다 변경된 상태를 되돌리는 방법을 알려주기 때문에 매우 편리하다. 예를 들어 파일을 두 개 수정하고서 따로따로 커밋하려고 했지만, 실수로 git add \* 라고 실행해 버렸다. 두 파일 모두 Staging Area에 들어 있다. 이제 둘 중 하나를 어떻게 꺼낼까? 우선 git status 명령으로 확인해보자:

$ git add .

$ git status

On branch master

Changes to be committed:

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

modified: README.txt

modified: benchmarks.rb

Changes to be commited 밑에 git reset HEAD <file>...이라는 문장을 볼 수 있다. 이 명령으로 Unstage 상태로 변경할 수 있다. benchmarks.rb 파일을 Unstage 상태로 변경해보자:

$ git reset HEAD benchmarks.rb

Unstaged changes after reset:

M benchmarks.rb

$ git status

On branch master

Changes to be committed:

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

modified: README.txt

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: benchmarks.rb

명령어가 낮설게 느껴질 수도 있지만 잘 동작한다. benchmarks.rb 파일은 Unstage 상태가 됐다.

### [Modified 파일 되돌리기](http://git-scm.com/book/ko/v1/Git%EC%9D%98-%EA%B8%B0%EC%B4%88-%EB%90%98%EB%8F%8C%EB%A6%AC%EA%B8%B0#Modified-파일-되돌리기)

어떻게 해야 benchmarks.rb 파일을 수정하고 나서 다시 되돌릴 수 있을까? 그러니까 최근 커밋된 버전으로(아니면 처음 Clone했을 때처럼 워킹 디렉토리에 처음 Checkout 한 그 내용으로) 되돌리는 방법이 무얼까?git status 명령이 친절하게 알려준다. 바로 위에 있는 예제에서 Unstaged 부분을 보자:

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: benchmarks.rb

위의 메시지는 수정한 파일을 되돌리는 방법을 꽤 정확하게 알려준다(적어도 Git 1.6.1이후 버전부터는 그렇다. 만약 예전 것을 아직 사용하고 있으면 업그레드하는 것이 좋다. 편의성이 많이 개선됐다). 알려주는 대로 한 번 해보자:

$ git checkout -- benchmarks.rb

$ git status

On branch master

Changes to be committed:

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

modified: README.txt

정상적으로 복원된 것을 알 수 있다. 하지만 이 명령은 꽤 위험한 명령이라는 것을 알아야 한다. 수정 이전의 파일로 덮어썼기 때문에 수정했던 내용은 전부 사라진다. 수정한 내용이 진짜 마음에 들지 않을 때에만 사용하자. 정말 이렇게 삭제해야 한다면 Stash와 Branch를 사용하자. 다음 장에서 다루는 이 방법들이 훨씬 낫다.

Git으로 커밋한 모든 것은 언제나 복구할 수 있다. 삭제한 브랜치에 있었던 것도 --amend 옵션으로 다시 커밋한 것도 복구할 수 있다(자세한 것은 9장에서 다룬다). 하지만, 커밋하지 않고 잃어버린 것은 절대로 되돌릴 수 없다.

# **2.5 Git의 기초 - 리모트 저장소**

## 리모트 저장소

리모트 저장소를 관리할 줄 알아야 다른 사람과 함께 일할 수 있다. 리모트 저장소는 인터넷이나 네트워크 어딘가에 있는 저장소를 말한다. 저장소는 여러 개가 있을 수 있는데 어떤 저장소는 읽고 쓰기 모두 할 수 있고 어떤 저장소는 읽기 권한만 있을 수도 있다. 간단히 말해서 다른 사람들과 함께 일한다는 것은 리모트 저장소를 관리하면서 데이터를 거기에 Push하고 Pull하는 것이다. 리모트 저장소를 관리한다는 것은 저장소를 추가, 삭제하는 것뿐만 아니라 브랜치를 관리하고 추적할지 말지 등을 관리하는 것을 말한다. 이번에는 리모트 저장소를 관리하는 방법에 대해 설명한다.

### [리모트 저장소 확인하기](http://git-scm.com/book/ko/v1/Git%EC%9D%98-%EA%B8%B0%EC%B4%88-%EB%A6%AC%EB%AA%A8%ED%8A%B8-%EC%A0%80%EC%9E%A5%EC%86%8C#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:

$ 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

-v옵션을 주어 단축이름과 URL을 함께 볼 수 있다:

$ git remote -v

origin git://github.com/schacon/ticgit.git (fetch)

origin git://github.com/schacon/ticgit.git (push)

리모트 저장소가 여러 개 있다면 이 명령은 전부 보여준다. 내 Grit 저장소에서 실행하면 아래와 같이 출력한다:

$ cd grit

$ git remote -v

bakkdoor git://github.com/bakkdoor/grit.git

cho45 git://github.com/cho45/grit.git

defunkt git://github.com/defunkt/grit.git

koke git://github.com/koke/grit.git

origin git@github.com:mojombo/grit.git

이렇게 리모트 저장소가 여러 개가 등록되어 있으면 다른 사람이 기여한 내용(Contributions)을 쉽게 가져올 수 있다. 그리고 origin만 SSH URL이기 때문에 origin에만 Push할 수 있다(4장에서 좀 더 자세히 다룬다).

### [리모트 저장소 추가하기](http://git-scm.com/book/ko/v1/Git%EC%9D%98-%EA%B8%B0%EC%B4%88-%EB%A6%AC%EB%AA%A8%ED%8A%B8-%EC%A0%80%EC%9E%A5%EC%86%8C#리모트-저장소-추가하기)

이전 절에서도 리모트 저장소를 추가하는 것에 대해 설명했었지만 수박 겉핥기식으로 살펴봤을 뿐이었다. 여기에서는 리모트 저장소를 추가하는 방법을 자세하게 설명한다. 쉽게 새 리모트 저장소를 추가할 수 있는데 git remote add [단축이름] [url] 명령을 실행한다:

$ git remote

origin

$ git remote add pb git://github.com/paulboone/ticgit.git

$ git remote -v

origin git://github.com/schacon/ticgit.git

pb git://github.com/paulboone/ticgit.git

이제 URL 대신에 스트링 pb를 사용할 수 있다. 예를 들어 로컬 저장소에는 없지만 Paul의 저장소에 있는 것을 가져오려면 아래과 같이 실행한다:

$ git fetch pb

remote: Counting objects: 58, done.

remote: Compressing objects: 100% (41/41), done.

remote: Total 44 (delta 24), reused 1 (delta 0)

Unpacking objects: 100% (44/44), done.

From git://github.com/paulboone/ticgit

\* [new branch] master -> pb/master

\* [new branch] ticgit -> pb/ticgit

로컬에서 pb/master가 Paul의 master 브랜치이다. 이것을 로컬 브랜치중 하나에 머지하거나 체크아웃하여 브랜치 내용을 자세히 확인할 수 있다.

### [리모트 저장소를 Pull 하거나 Fetch 하기](http://git-scm.com/book/ko/v1/Git%EC%9D%98-%EA%B8%B0%EC%B4%88-%EB%A6%AC%EB%AA%A8%ED%8A%B8-%EC%A0%80%EC%9E%A5%EC%86%8C#리모트-저장소를-Pull-하거나-Fetch-하기)

앞서 설명했듯이 리모트 저장소에서 데이터를 가져오려면 간단히 아래와 같이 실행한다:

$ git fetch [remote-name]

이 명령은 로컬에는 없지만, 리모트 저장소에는 있는 데이터를 모두 가져온다. 그리고 나면 리모트 저장소의 모든 브랜치를 로컬에서 접근할 수 있어서 언제든지 머지를 하거나 내용을 살펴볼 수 있다(우리는 3장에서 브랜치를 사용하는 방법에 대해 좀 더 자세히 설명할 것이다).

저장소를 Clone하면 명령은 자동으로 리모트 저장소를 origin이라는 이름으로 추가한다. 그래서 나중에git fetch origin을 실행하면 Clone한 이후에(혹은 마지막으로 가져온 이후에) 수정된 것을 모두 가져온다. fetch 명령은 리모트 저장소의 데이터를 모두 로컬로 가져오지만, 자동으로 머지하지 않는다. 그래서 당신이 로컬에서 하던 작업을 정리하고 나서 수동으로 머지해야 한다.

그냥 쉽게 git pull 명령으로 리모트 저장소 브랜치에서 데이터를 가져올 뿐만 아니라 자동으로 로컬 브랜치와 머지시킬 수 있다. 먼저 git clone 명령은 자동으로 로컬의 master 브랜치가 리모트 저장소의 master 브랜치를 추적하도록 한다(물론 리모트 저장소에 master 브랜치가 있다고 가정에서). 그리고 git pull 명령은 Clone한 서버에서 데이터를 가져오고 그 데이터를 자동으로 현재 작업하는 코드와 머지시킨다.

### [리모트 저장소에 Push하기](http://git-scm.com/book/ko/v1/Git%EC%9D%98-%EA%B8%B0%EC%B4%88-%EB%A6%AC%EB%AA%A8%ED%8A%B8-%EC%A0%80%EC%9E%A5%EC%86%8C#리모트-저장소에-Push하기)

프로젝트를 공유하고 싶을 때 리모트 저장소에 Push할 수 있다. 이 명령은 git push [리모트 저장소 이름] [브랜치 이름]으로 단순하다. master 브랜치를 origin 서버에 Push하려면(다시 말하지만 Clone하면 보통 자동으로 origin 이름이 생성된다) 아래와 같이 서버에 Push한다:

$ git push origin master

이 명령은 Clone한 리모트 저장소에 쓰기 권한이 있고, Clone하고 난 이후 아무도 리모트 저장소에 Push하지 않았을 때만 사용할 수 있다. 다시 말해서 Clone한 사람이 여러 명 있을 때, 다른 사람이 Push한 후에 Push하려고 하면 Push할 수 없다. 먼저 다른 사람이 작업한 것을 가져와서 머지한 후에 Push할 수 있다. 3장에서 서버에 Push하는 방법에 대해 자세히 설명할 것이다.

### [리모트 저장소 살펴보기](http://git-scm.com/book/ko/v1/Git%EC%9D%98-%EA%B8%B0%EC%B4%88-%EB%A6%AC%EB%AA%A8%ED%8A%B8-%EC%A0%80%EC%9E%A5%EC%86%8C#리모트-저장소-살펴보기)

(역주, 이 절은 최신 버전의 Git이 출력하는 메시지와 조금 다르다.)

git remote show [리모트 저장소 이름] 명령으로 리모트 저장소의 구체적인 정보를 확인할 수 있다.origin 같은 단축이름으로 이 명령을 실행하면 아래와 같은 정보를 볼 수 있다:

$ 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 저장소의 URL과 추적하는 브랜치를 출력한다. 이 명령은 git pull 명령을 실행할 때 master 브랜치와 머지할 브랜치가 무엇인지 보여 준다. git pull 명령은 리모트 저장소 브랜치의 데이터를 모두 가져오고 나서 자동으로 머지할 것이다. 그리고 가져온 모든 리모트 저장소 정보도 출력한다.

좀 더 Git을 열심히 사용하게 되면 git remote show 명령은 더 많은 정보를 보여줄 것이다. 여러분도 언젠가는 아래와 같은 메시지(역주, 다수의 브랜치를 사용하는 메시지)를 볼 날이 올 것이다.

$ git remote show origin

\* remote origin

URL: git@github.com:defunkt/github.git

Remote branch merged with 'git pull' while on branch issues

issues

Remote branch merged with 'git pull' while on branch master

master

New remote branches (next fetch will store in remotes/origin)

caching

Stale tracking branches (use 'git remote prune')

libwalker

walker2

Tracked remote branches

acl

apiv2

dashboard2

issues

master

postgres

Local branch pushed with 'git push'

master:master

브랜치명을 생략하고 git push 명령을 실행할 때 어떤 브랜치가 어떤 브랜치로 Push되는지 보여준다. 또 아직 로컬로 가져오지 않은 리모트 저장소의 브랜치는 어떤 것들이 있는지, 서버에서는 삭제됐지만 아직 가지고 있는 브랜치는 어떤 것인지, git pull 명령을 실행했을 때 자동으로 머지할 브랜치는 어떤 것이 있는지 보여준다.

### [리모트 저장소 이름을 바꾸거나 리모트 저장소를 삭제하기](http://git-scm.com/book/ko/v1/Git%EC%9D%98-%EA%B8%B0%EC%B4%88-%EB%A6%AC%EB%AA%A8%ED%8A%B8-%EC%A0%80%EC%9E%A5%EC%86%8C#리모트-저장소-이름을-바꾸거나-리모트-저장소를-삭제하기)

git remote rename 명령으로 리모트 저장소의 이름을 변경할 수 있다. 예를 들어 pb를 paul로 변경하려면 git remote rename 명령을 사용한다:

$ git remote rename pb paul

$ git remote

origin

paul

리모트 저장소의 브랜치 이름도 바뀐다. 여태까지 pb/master로 리모트 저장소 브랜치를 사용했으면 이제는 paul/master라고 사용해야 한다.

리모트 저장소를 삭제해야 한다면 git remote rm 명령을 사용한다. 서버 정보가 바뀌었을 때, 더는 별도의 미러가 필요하지 않을 때, 더는 기여자가 활동하지 않을 때 필요하다:

$ git remote rm paul

$ git remote

origin

# **2.6 Git의 기초 - 태그**

## 태그

다른 VCS처럼 Git도 태그를 지원한다. 사람들은 보통 릴리즈할 때 사용한다(v1.0, 등등). 이번에는 태그를 조회하고 생성하는 법과 태그의 종류를 설명한다.

### [태그 조회하기](http://git-scm.com/book/ko/v2/Git%EC%9D%98-%EA%B8%B0%EC%B4%88-%ED%83%9C%EA%B7%B8#태그-조회하기)

우선 git tag 명령으로 이미 만들어진 태그가 있는지 확인할 수 있다.

**$** git tag

v0.1

v1.3

이 명령은 알파벳 순서로 태그를 보여준다. 사실 순서는 별로 중요한 게 아니다.

검색 패턴을 사용하여 태그를 검색할 수 있다. Git 소스 저장소는 500여 개의 태그가 있다. 만약 1.8.5 버전의 태그들만 검색하고 싶으면 아래와 같이 실행한다.

**$** git tag -l 'v1.8.5\*'

v1.8.5

v1.8.5-rc0

v1.8.5-rc1

v1.8.5-rc2

v1.8.5-rc3

v1.8.5.1

v1.8.5.2

v1.8.5.3

v1.8.5.4

v1.8.5.5

### [태그 붙이기](http://git-scm.com/book/ko/v2/Git%EC%9D%98-%EA%B8%B0%EC%B4%88-%ED%83%9C%EA%B7%B8#태그-붙이기)

Git의 태그는 Lightweight 태그와 Annotated 태그로 두 종류가 있다.

Lightweight 태그는 브랜치와 비슷한데 브랜치처럼 가리키는 지점을 최신 커밋으로 이동시키지 않는다. 단순히 특정 커밋에 대한 포인터일 뿐이다.

한편 Annotated 태그는 Git 데이터베이스에 태그를 만든 사람의 이름, 이메일과 태그를 만든 날짜, 그리고 태그 메시지도 저장한다. GPG(GNU Privacy Guard)로 서명할 수도 있다. 이 모든 정보를 저장해둬야 할 때에만 Annotated 태그를 추천한다. 그냥 다른 정보를 저장하지 않는 단순한 태그가 필요하다면 Lightweight 태그를 사용하는 것이 좋다.

### [Annotated 태그](http://git-scm.com/book/ko/v2/Git%EC%9D%98-%EA%B8%B0%EC%B4%88-%ED%83%9C%EA%B7%B8#Annotated-태그)

Annotated 태그를 만드는 방법은 간단하다. tag 명령을 실행할 때 -a 옵션을 추가한다.

**$** git tag -a v1.4 -m 'my version 1.4'

**$** git tag

v0.1

v1.3

v1.4

-m 옵션으로 태그를 저장할 때 메시지를 함께 저장할 수 있다. 명령을 실행할 때 메시지를 입력하지 않으면 Git은 편집기를 실행시킨다.

git show 명령으로 태그 정보와 커밋 정보를 모두 확인할 수 있다.

**$** git show v1.4

tag v1.4

Tagger: Ben Straub <ben@straub.cc>

Date: Sat May 3 20:19:12 2014 -0700

my version 1.4

commit ca82a6dff817ec66f44342007202690a93763949

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

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

changed the version number

커밋 정보를 보여주기 전에 먼저 태그를 만든 사람이 누구인지, 언제 태그를 만들었는지, 그리고 태그 메시지가 무엇인지 보여준다.

### [Lightweight Tags](http://git-scm.com/book/ko/v2/Git%EC%9D%98-%EA%B8%B0%EC%B4%88-%ED%83%9C%EA%B7%B8#Lightweight-Tags)

Lightweight 태그는 기본적으로 파일에 커밋 체크섬을 저장하는 것뿐이다. 다른 정보는 저장하지 않는다. Lightweight 태그를 만들 때에는 -a, -s, -m 옵션을 사용하지 않는다.

**$** git tag v1.4-lw

**$** git tag

v0.1

v1.3

v1.4

v1.4-lw

v1.5

이 태그에 git show를 실행하면 별도의 태그 정보를 확인할 수 없다. 이 명령은 단순히 커밋 정보만을 보여준다.

**$** git show v1.4-lw

commit ca82a6dff817ec66f44342007202690a93763949

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

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

changed the version number

### [나중에 태그하기](http://git-scm.com/book/ko/v2/Git%EC%9D%98-%EA%B8%B0%EC%B4%88-%ED%83%9C%EA%B7%B8#나중에-태그하기)

예전 커밋에 대해서도 태그할 수 있다. 커밋 히스토리는 아래와 같다고 가정한다.

**$** git log --pretty=oneline

15027957951b64cf874c3557a0f3547bd83b3ff6 Merge branch 'experiment'

a6b4c97498bd301d84096da251c98a07c7723e65 beginning write support

0d52aaab4479697da7686c15f77a3d64d9165190 one more thing

6d52a271eda8725415634dd79daabbc4d9b6008e Merge branch 'experiment'

0b7434d86859cc7b8c3d5e1dddfed66ff742fcbc added a commit function

4682c3261057305bdd616e23b64b0857d832627b added a todo file

166ae0c4d3f420721acbb115cc33848dfcc2121a started write support

9fceb02d0ae598e95dc970b74767f19372d61af8 updated rakefile

964f16d36dfccde844893cac5b347e7b3d44abbc commit the todo

8a5cbc430f1a9c3d00faaeffd07798508422908a updated readme

“updated rakefile” 커밋을 v1.2로 태그하지 못했다고 해도 나중에 태그를 붙일 수 있다. 특정 커밋에 태그하기 위해서 명령의 끝에 커밋 체크섬을 명시한다(긴 체크섬을 전부 사용할 필요는 없다).

**$** git tag -a v1.2 9fceb02

이제 아래와 같이 만든 태그를 확인한다.

**$** git tag

v0.1

v1.2

v1.3

v1.4

v1.4-lw

v1.5

**$** git show v1.2

tag v1.2

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

Date: Mon Feb 9 15:32:16 2009 -0800

version 1.2

commit 9fceb02d0ae598e95dc970b74767f19372d61af8

Author: Magnus Chacon <mchacon@gee-mail.com>

Date: Sun Apr 27 20:43:35 2008 -0700

updated rakefile

...

### [태그 공유하기](http://git-scm.com/book/ko/v2/Git%EC%9D%98-%EA%B8%B0%EC%B4%88-%ED%83%9C%EA%B7%B8#태그-공유하기)

git push 명령은 자동으로 리모트 서버에 태그를 전송하지 않는다. 태그를 만들었으면 서버에 별도로 Push 해야 한다. 브랜치를 공유하는 것과 같은 방법으로 할 수 있다. git push origin [태그 이름]을 실행한다.

**$** git push origin v1.5

Counting objects: 14, done.

Delta compression using up to 8 threads.

Compressing objects: 100% (12/12), done.

Writing objects: 100% (14/14), 2.05 KiB | 0 bytes/s, done.

Total 14 (delta 3), reused 0 (delta 0)

To git@github.com:schacon/simplegit.git

\* [new tag] v1.5 -> v1.5

만약 한 번에 태그를 여러 개 Push 하고 싶으면 --tags 옵션을 추가하여 git push명령을 실행한다. 이 명령으로 리모트 서버에 없는 태그를 모두 전송할 수 있다.

**$** git push origin --tags

Counting objects: 1, done.

Writing objects: 100% (1/1), 160 bytes | 0 bytes/s, done.

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

To git@github.com:schacon/simplegit.git

\* [new tag] v1.4 -> v1.4

\* [new tag] v1.4-lw -> v1.4-lw

이제 누군가 저장소에서 Clone 하거나 Pull을 하면 모든 태그 정보도 함께 전송된다.

### [태그를 Checkout 하기](http://git-scm.com/book/ko/v2/Git%EC%9D%98-%EA%B8%B0%EC%B4%88-%ED%83%9C%EA%B7%B8#태그를-Checkout-하기)

태그는 브랜치와는 달리 가리키는 커밋을 바꿀 수 없는 이름이기 때문에 Checkout 해서 사용할 수 없다. 태그가 가리키는 특정 커밋 기반의 브랜치를 만들어 작업하려면 아래와 같이 새로 브랜치를 생성한다.

**$** git checkout -b version2 v2.0.0

Switched to a new branch 'version2'

물론 이렇게 브랜치를 만든 후에 version2 브랜치에 커밋하면 브랜치는 업데이트된다. 하지만, v2.0.0태그는 가리키는 커밋이 변하지 않았으므로 두 내용이 가리키는 커밋이 다르다는 것을 알 수 있다.

# **2.7 Git의 기초 - 팁과 트릭**

## 팁과 트릭

Git의 기초를 마치기 전에 Git을 좀 더 쉽고 편안하게 쓸 수 있게 만들어 줄 몇 가지 팁과 트릭도 설명한다. 이런 팁 없이 Git을 사용하는 사람들도 많다. 우리는 이 책에서 이 팁을 다시 거론하지 않고 이런 팁을 알고 있다고 가정한다. 그래서 알고 있는 것이 좋다.

### [자동완성](http://git-scm.com/book/ko/v1/Git%EC%9D%98-%EA%B8%B0%EC%B4%88-%ED%8C%81%EA%B3%BC-%ED%8A%B8%EB%A6%AD#자동완성)

Bash 쉘을 쓰고 있다면 멋진 자동완성(Auto-completion) 기능을 사용할 수 있다. https://github.com/git/git/blob/master/contrib/completion/git-completion.bash 에서 바로 다운받는다. 그 파일을 홈 디렉토리에 카피하고 .bashrc 파일에 아래와 같은 내용을 추가하자:

source ~/git-completion.bash

또 모든 사용자가 사용할 수 있게 설정할 수 있다. Mac 시스템이라면 이 스크립트를/opt/local/etc/bash\_completion.d 디렉토리에 복사하고 리눅스라면/etc/bash\_completion.d/에 복사한다. 이 디렉토리는 Bash가 자동완성을 지원하기 위해 사용하는 디렉토리다.

윈도에 msysGit을 설치해서 Git Bash를 사용하는 경우에는 자동완성이 미리 설정되어 있다.

Git 명령을 입력할 때 <Tab> 키를 누르면 Git이 제안하는 명령어가 출력된다:

$ git co<tab><tab>

commit config

이 경우 git co를 입력하고 Tab 키를 두번 누르면 commit과 config를 제안한다. 이 때 m<tab>을 입력하면 자동으로 git commit명령을 완성한다.

옵션에도 이 기능이 되고 더 유용하다. 예를 들어 git log명령을 실행하는데 옵션이 전혀 기억나지 않는다면 아래와 같이 입력하고 Tap 키를 누르면 아래와 같은 옵션을 제안한다:

$ git log --s<tab>

--shortstat --since= --src-prefix= --stat --summary

이건 상당히 멋진 팁이다. 아마 문서를 찾아보는 등의 시간을 절약해 줄 것이다.

### [Git Alias](http://git-scm.com/book/ko/v1/Git%EC%9D%98-%EA%B8%B0%EC%B4%88-%ED%8C%81%EA%B3%BC-%ED%8A%B8%EB%A6%AD#Git-Alias)

명령을 완벽하게 입력하지 않으면 Git은 알아듣지 못한다. Git의 명령을 전부 입력하는 것이 귀찮다면 git config를 사용하여 각 명령의 Alias을 쉽게 만들 수 있다. 아래는 Alias을 만드는 예이다:

$ git config --global alias.co checkout

$ git config --global alias.br branch

$ git config --global alias.ci commit

$ git config --global alias.st status

이제 git commit 대신 git ci만으로도 커밋할 수 있다. Git을 계속 사용한다면 다른 명령어도 자주 사용하게 될 것이다. 자주 사용하는 명령은 Alias을 만들어 편하게 사용한다.

이미 있는 명령을 편리하고 새로운 명령으로 만들어 사용할 수 있다. 예를 들어 파일을 Unstage 상태로 변경하는 명령을 만들어서 불편함을 덜 수 있다. 아래와 같이 unstage 라는 Alias을 만든다:

$ git config --global alias.unstage 'reset HEAD --'

아래 두 명령은 동일한 명령이다:

$ git unstage fileA

$ git reset HEAD fileA

한결 간결해졌다. 추가로 last 명령을 만들어 보자:

$ git config --global alias.last 'log -1 HEAD'

이제 최근 커밋을 좀 더 쉽게 확인할 수 있다:

$ git last

commit 66938dae3329c7aebe598c2246a8e6af90d04646

Author: Josh Goebel <dreamer3@example.com>

Date: Tue Aug 26 19:48:51 2008 +0800

test for current head

Signed-off-by: Scott Chacon <schacon@example.com>

이것으로 쉽게 새로운 명령을 만들 수 있다. 그리고 Git의 명령어뿐만 아니라 외부 명령어도 실행할 수 있다.!를 제일 앞에 추가하면 외부 명령을 실행한다. 아래 명령은 git visual이라고 입력하면 gitk가 실행된다:

$ git config --global alias.visual '!gitk'

# **Git 상황별 명령어 tips**

git의 경우 튜토리얼들과 설명들이 매우 많은편이지만, 처음 접하는 사람들은 명령어들의 숫자나 개념이 익숙하지 않아서 어떤 상황에 어떤 명령어를 넣어야하는지는 정작 힘들 경우가 많다. 그래서 이 포스트에서는 상황을 가정하고 해당 상황일때 어떻게 대처해야하는지를 시나리오별로 정리해보려고 한다. 시나리오들은 생각나는대로 계속 추가 예정이다.

* 참고1: git에대해 기본적인 내용을 공부하고싶으면 이 사이트를 추천한다.<http://learnbranch.urigit.com/> 인터랙티브한 방식으로 개념을 잘 설명하고있어서 처음 배울때 많은 도움이 된다.
* 참고2: <http://git-scm.com/book/ko> git에대한 기술적인 상세한 내용을 보려면 이곳을 참고할것.

마지막으로, master를 지금 트리가 변경된 부분으로 이동합니다. (편하신 방법으로 하세요)  
이 목표를 달성하기 위해서는 많은 방법이 있는데요(체리픽을 고민중이시죠?), 체리픽은 나중에 더 살펴보기로 하고, 우선은 위의 방법으로 해결해보세요.

## 커밋한 내용을 버리고 이전 커밋 상태로 돌리기

**상황 설명:**

코드를 작성하다가 실수로 잘못된 수정들을 많이 해서 문제가 생겼을경우, 이것들을 지난 커밋 상태로 돌려놓고 다시 코드를 테스트해보고 싶은 경우들이 많이 있다.

**해결하기:**

이때 보통 어떤부분을 수정했는지 사람이 완벽하게 다 기억하지는 못하기 때문에 손으로 일일히 복구하기 보단 git의 기능을 이용하여 지난 커밋으로 변경하는 것이 훨씬 빠르고 안정적이다.

1. 저장소 전체(모든파일) 하드리셋하기

하드리셋을 할 경우 해당 커밋으로 저장소를 돌려놓으면서 그 **이후의 commit 상태들은 다 지워지게됨에 주의**하자.(수정된 내용들뿐 아니라 커밋 기록들까지 아예 깨끗히 삭제된다.)

레포지토리의 마지막 커밋 상태(HEAD)로 돌아가려면 다음명령어를 입력한다.

|  |  |
| --- | --- |
| 1  2 | git reset --hard |

다음과 같이 실행하면 현재 레포지토리의 HEAD로부터 과거 3번째 커밋으로 돌아간다.

|  |  |
| --- | --- |
| 1  2 | git reset --hard HEAD~3 |

1. 파일별로 수정내용 돌려놓기

저장소 전체를 돌려놓는 것이 아닌 몇몇 파일만 원래대로 돌려놓으려면 unstage 된 상태에서 체크아웃 명령어를 이용하여 다음과같이 하면 된다.

|  |  |
| --- | --- |
| 1  2 | git checkout 파일명 |

수정된 파일이 이미 커밋을 위해 stage되어있을경우 다음 명령어를 통해 unstage 시킨 후 체크아웃을 해야한다.

|  |  |
| --- | --- |
| 1  2 | git reset 파일명 |

1. 만약 변경하려는 로컬저장소의 커밋이 이미 remote 저장소에 push가 된 상태인경우 reset명령어를 사용하면 안되고 revert를 사용해야 한다. 로컬에서 reset 해버리면 commit 기록이 삭제되어 해당 저장소의 커밋 히스토리를 공유하고있는 다른 사용자의 저장소에 문제가 생길 수 있기 때문이다. revert 의경우 돌려놓으려는 커밋의 반대되는 내용을 커밋하는 방식으로 상태를 돌려놓는기때문에 다른 사용자들에게도 정상적으로 반영될 수 있다.

## 현재 작업중인 내용을 저장소 상태를 저장해두고 이전 커밋 상태로 돌리기

**상황 설명:**

코드를 작성하다보면 새로 작성했거나 수정한 코드에 뭔가 문제가 생겨서 부분적으로 동작이 하지 않는 경우가 생긴다. 이때 어떤부분이 문제가되는지 확인하거나, 기존코드에서도 동일한 문제가 생기는지 확인하려면 저장소 상태를 이전 커밋상태로 checkout 하게된다. 이때 새로 작성한 부분들 또한 잃어버리지 않고 안전하게 보관해 두려면 어떻게 해야할까?

**해결하기:**

이전커밋 상태로 돌리기 전에 현재 저장소의 상태를 저장하려면 두가지 방법이있다.

1. commit을 해서 실제로 현재 상태를 저장하는 경우
2. stash 명령을 이용해서 임시로 현재 상태를 저장하는 경우

1번의 경우는 다들 익숙한 개념이니 넘어가고 stash 기능을 설명해보도록 하겠다. ‘stash’ 란 ‘안전한 곳에 넣어두다’ 라는 뜻을 가졌다. 이 말 뜻 그대로 stash 명령은 현재 변경 상태를 잠시 안전하게 보관해 두었다가 나중에 필요할때 꺼내서 사용할 수 있게 해주는 기능을 의미한다.

다음 명령어를 통해 현재 상태를 저장한다.

|  |  |
| --- | --- |
| 1  2 | git stash (혹은 git stash save) |

다음 명령어를 통해서 stash에 저장된 가장 최근의 상태를 불러와서 현재 저장소에 적용 후, 해당 stash는 삭제한다.

|  |  |
| --- | --- |
| 1  2 | git stash pop |

stash의 내용을 적용한 후에도 해당 stash를 계속 보관하고 싶을 경우에는 다음 명령어를 이용한다.

|  |  |
| --- | --- |
| 1  2 | git stash apply |

pop이나 apply 과정에서 충돌(conflict)가 발생할경우에는 일반적인 merge 상황에서 처럼 동일한 방법으로 해결해주면 된다.

마지막으로 stash된 내용들을 모두 삭제하려면 다음 명령어를 이용한다.

|  |  |
| --- | --- |
| 1  2 | git stash clear |

## 커밋(commit) 메시지 내용을 변경하거나 합치고 싶을경우

**상황 설명:**

메이저한 내용들을 수정하고 멋지게 커밋메시지를 작성해서 커밋했는데, 잠시 후 마이너한 버그를 발견해서 코드를 또 수정하게 되었다. 지저분한 커밋 히스토리를 남기고 싶지않아서, 이 마이너한 수정사항을 독립적인 커밋이 아닌 기존 메이저 커밋에 포함시키고 싶을경우 어떻게 해야 할까?

**해결하기:**

커밋된 내용을 수정 또는 통합하기 위해선 다음과 같은 명령어를 사용하면 된다. 현재 레포지토리의 HEAD로부터 과거 3번째 커밋까지를 인터랙티브한 방법으로 수정하겠다는 명령어이다. 수정하고 싶은 커밋 수많큼 뒤에 숫자로 적어주면 된다.

|  |  |
| --- | --- |
| 1  2 | git rebase -i HEAD~3 |

인터랙티브 모드에서, 통합할 커밋은 squash 옵션을 주면되고, 커밋 메시지만 수정하려면 reward 를 하면 된다[1](http://www.letmecompile.com/git-%EC%83%81%ED%99%A9%EB%B3%84-%EB%AA%85%EB%A0%B9%EC%96%B4-tips/#fn-258-ref_book).

* 주의: 이미 리모트 서버로 push된 내용이 있을경우에는 커밋 내용을 변경할 경우 리모트 서버와 싱크가 맞지않아 혼란이 발생하기 때문에 항상 push되지 않은 경우만 rebase를 하는것이 좋다.

인터랙티브 모드에대한 더 자세한 설명은 다음 문서를 참조하자.

[http://git-scm.com/book/ko/Git-도구-히스토리-단장하기](http://git-scm.com/book/ko/Git-%EB%8F%84%EA%B5%AC-%ED%9E%88%EC%8A%A4%ED%86%A0%EB%A6%AC-%EB%8B%A8%EC%9E%A5%ED%95%98%EA%B8%B0)

## 버그 수정을 위한 브랜칭(브랜치에서 특정 커밋만 반영하기)[2](http://www.letmecompile.com/git-%EC%83%81%ED%99%A9%EB%B3%84-%EB%AA%85%EB%A0%B9%EC%96%B4-tips/#fn-258-ref_site)

**상황 설명:**

개발중에 현재 상태를 보존해 둔 상태로 현재 계속 발생하고있는 버그를 수정해야 하는 상황에 처했다. 하지만 간단한 버그가 아니여서 먼저 bug\_fix 라는 branch를 생성했고, 며칠에 걸려 찾느라 버그를 찾기위한 logging 명령어들과 기타 테스트코드 또한 히스토리를 남기기 위해 bug\_fixbranch에 지속적으로 커밋을 했다.

결국 버그는 코드의 한 부분만 수정하면 되는것으로 밝혀졌고, 이 최종 수정을 bug\_fix branch에 마지막으로 커밋했다. 이때 디버깅을 위한 logging 코드를 제외하고 실제 버그를 수정하는 최종 수정된 커밋내용만 master 브랜치에 적용하고 싶을 경우 어떻게 하면 될까?

**해결하기:**

불필요한 디버그 코드는 bug\_fix 브랜치에 남겨둔채 마지막 커밋만 master 브랜치로 가져오려면cherry-pick 명령어를 이용하면 된다.

master 브랜치를 먼저 선택한 후

|  |  |
| --- | --- |
| 1  2 | git checkout master |

다음 명령어를 입력하면 bug\_fix 브랜치의 HEAD 포인터가 가리키고있는 커밋의 내용(가장 최근 커밋 내용)이 동일하게 master 브랜치에 커밋되게된다.

|  |  |
| --- | --- |
| 1  2 | git cherry-pick bug\_fix |

## 이미 리모트로 push된 마지막 커밋을 삭제하고싶을경우

마지막으로 수정해서 푸시한 커밋에 문제가 있을경우 이를 돌려놓고 싶을 경우가 있다. 단, 푸시된 커밋을 다른 유저가 이미 받아갔을경우에는 소스가 꼬이게되므로 리모트 저장소에있는 마지막 커밋을 아무도 받아가지 않은 경우에만 가능하다.

로컬에서 먼저 마지막 커밋을 리셋한 후에

|  |  |
| --- | --- |
| 1  2 | git reset --hard HEAD^ |

force push 옵션을 줘서 강제로 리모트저장소로 푸시해버리면 된다.

|  |  |
| --- | --- |
| 1  2 | git push -f |

## 부록: Git GUI Client 소스트리(SourceTree) 소개

비트버켓(bitbucket)이던 깃허브(github)던 개인 git서버를 쓰던 가장 추천하는 GUI 클라이언트는 소스트리이다. github에서 제공하는 클라이언트의 경우 매우 기능이 제한적인데 반해, 소스트리의 경우는 위에서 설명했던 커맨드라인에서 가능한 거의 대부분의 명령어들을 마우스 클릭 몇번으로 실행할 수 있다. 게다가 무료이며 Windows, Mac 모두 지원하니 강력 추천한다.

