

**Wind Lidar 개발 환경 구성**



2017.03

본 문서는 한진정보통신에서 문서의 저작권을 소유하고 있으며, 제공 목적 이외의 용도로 사용할 수 없습니다. 본 문서와 관련하여 전체 또는 부분적으로 복제하거나 복사 또는 한진정보통신의 승인 없이는 어떤 사람에게 전송 될 수 없습니다

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 개 정 현 황 | | | | | |
| 개정번호 | 일 자 | 개 정 사 유 | 작 성 | 검 토 | 승 인 |
| V 1.0 | 2017.03.24 |  | xxx |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

목 차

[1. Git 사용하기 ３](#_Toc478138335)

[1.1 SourceTree 사용하기 ３](#_Toc478138336)

[1.2 Git 설치하기 ３](#_Toc478138337)

[1.2.1 Git 설치 ３](#_Toc478138338)

[1.3 Git Bash 사용하기 ６](#_Toc478138339)

[1.3.1 git config ６](#_Toc478138340)

[1.3.2 git init ６](#_Toc478138341)

[1.3.3 git add ７](#_Toc478138342)

[1.3.4 git commit ７](#_Toc478138343)

[1.3.5 git remote ８](#_Toc478138344)

[1.3.6 git push ９](#_Toc478138345)

[1.4 Git 사용방법 １０](#_Toc478138346)

[1.4.1 새로운 저장소 만들기 １０](#_Toc478138347)

[1.4.2 저장소 받아오기 １１](#_Toc478138348)

[1.4.3 작업의 흐름 １１](#_Toc478138349)

[1.4.4 추가와 확정(commit) １１](#_Toc478138350)

[1.4.5 변경 내용 발행(push)하기 １２](#_Toc478138351)

[1.4.6 가지(branch)치기 １２](#_Toc478138352)

[1.4.7 갱신과 병합(merge) １３](#_Toc478138353)

[1.4.8 꼬리표 (tag) 달기 １４](#_Toc478138354)

[1.4.9 로컬 변경 내용 되돌리기 １５](#_Toc478138355)

[1.4.10 기타 명령어 １５](#_Toc478138356)

[1.4.11 저장소의 잘 못된 폴더 삭제 １６](#_Toc478138357)

[2. GIT Workflow １６](#_Toc478138358)

[2.1 Development Branch １６](#_Toc478138359)

[2.2 Function Branch １７](#_Toc478138360)

[2.3 Release Branch １９](#_Toc478138361)

[2.4 HotFix Branch ２０](#_Toc478138362)

[2.5 도구와 응용 ２１](#_Toc478138363)

[3. Visual Git Reference ２２](#_Toc478138364)

[3.1 기초 사용법 ２２](#_Toc478138365)

[3.2 관례 ２３](#_Toc478138366)

[3.3 명령어 살펴보기 ２４](#_Toc478138367)

[3.3.1 Diff ２４](#_Toc478138368)

[3.3.2 Commit ２５](#_Toc478138369)

[3.3.3 Checkout ２７](#_Toc478138370)

[3.3.4 Detached HEAD 에서 커밋하기 ２９](#_Toc478138371)

[3.3.5 Reset ３１](#_Toc478138372)

[3.3.6 Merge ３３](#_Toc478138373)

[3.3.7 Cherry Pick(열매 고르기) ３５](#_Toc478138374)

[3.3.8 Rebase ３５](#_Toc478138375)

[3.3.9 기술적인 내용 ３７](#_Toc478138376)

[3.3.10 명령어들의 결과 확인 ３８](#_Toc478138377)

# Git 사용하기

GitHub에 올라간 프로젝트 폴더를 GitHub에서 삭제를 할 수가 없다. 삭제를 위해서는 SoruceTree나 기타 Git Console에서 삭제를 수행해야 한다.

## SourceTree 사용하기

윈도우 환경에서 Git을 자동으로 업로드 할 수 있다. Source Tree는 버전관리 및 업로드에 최적화 되어 있다.

## Git 설치하기

### Git 설치

Git을 사용하려면 설치를 해야 하는데 다양한 방법으로 Git을 설치할 수 있지만 두가지 방법이 가장 일반적이다. 하나는 소스코드로부터 컴파일해서 설치하는 방법이고 다른 하나는 각 운영체게(혹은 플랫폼)의 패키지를 사용하여 설치하는 방법이다.

#### 소스코드로 설치하기

소스코드로 설치하면 Git의 가장 최시버전을 설치 할 수 있기 때문에 컴파일하여 설치할 시간이 있으면 소스코드로 Git을 설치하는 것이 좋다. Git은 계속 UI를 개선하고 있기 때문에 최신 버전을 사용하면 좋은 기능을 빨리 사용할 수 있다. 리눅스 패키지는 보통 최신 버전이 아니고 예전 버전이다. 그래서 Backport를 사용하거나 소스코드로 설치하는 것도 좋은 대안이다.

Git을 설치하려면 아래와 같은 라이브러리들이 필요하다. Git은 curl, zlib, opens니, expat, libiconv를 필요로 한다. 예를 들어 Fedora처럼 yum을 사용하는 시스템이나 apg-get이 있는 데비안류 시스템이면 아래 명령어를 실행하여 의존 패키지를 설치할 수 있다.

|  |
| --- |
| **$yum install curl-devel expat-devel gettext-devel \**  **Openssl-devel zlib-devel**  **$ apt-get install libcurl4-gnutls-dev libexpat1-dev gettext \**  **Libz-dev libssl-dev** |

필요한 라이브러리를 모두 설치하고 다음 단계를 진행한다. Git 웹사이트에서 최시 스냅샷을 가져온다.

|  |
| --- |
| http://git-scm.com/download |

그리고 컴파 일하고 설치한다

|  |
| --- |
| $tar –zxf git-1.7.2.2.tar.gz  $cd git-1.7.2.2  $make prefix=/usr/local all  $sudo make prefix=/usr/local install |

설치한 다음부터는 Git을 사용하여 Git 소스코드를 수정할 수 있다.

|  |
| --- |
| $git clone git://git.kernel.org/pub/scm/git/git.git |

#### 리눅스에 설치

리눅스에서 패키지로 Git을 설치할 때에는 보통 각 배포판에서 사용하는 패키지 관리도구를 사용하여 설치한다. Fedora에서는 아래와 같이 한다.

|  |
| --- |
| # yum install git-core |

Ubuntu같은 데비안류 배포판에선느 apt-get을 사용한다.

#### 윈도우에 설치

윈도우에서도 Git를 쉽게 설치할 수 있다. 구글 코드 페이지에서 msysGit 인스톨러를 내려받고 실행하면 된다.

|  |
| --- |
| **http://mysgit.github.com/** |

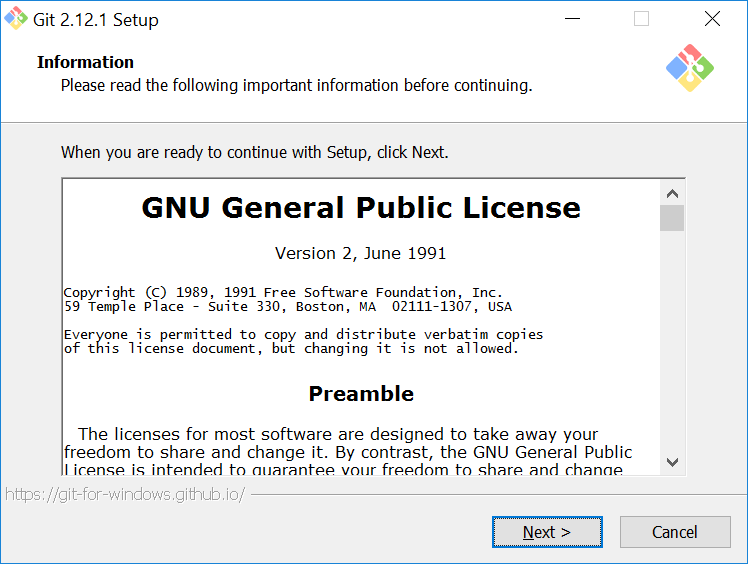
설치가 완료되면 CLI 프로그램과 GUI 프로그램을 둘 다 사용할 수 있다 CLI 프로그램에서는 SSH 클라이언트가 포함되어 있기 때문에 유용한다.

Windows 사용자가 Git의 다양한 명령어를 사용하려면 유닉스 스타일의 msysGit쉘을 사용하는 것이 좋다. 어쩔 수 없이 Windows에 포함된 기본 쉘(Command Prompt, 명령 프롬프트)을 꼭 써야 하면 공백이 포함된 파라미터를 Git 명령어에 넘길 때 작은 따옴표(‘’) 대신 큰 따옴표(“”)를 사용해야 한다. 파라미터 끝에 ^기호가 있을 때도 큰 따옴표로 파라미터를 감싸야 한다. Windows 쉘에서 ^기호는 다음줄로 명령어가 이어짐을 나타낸다.

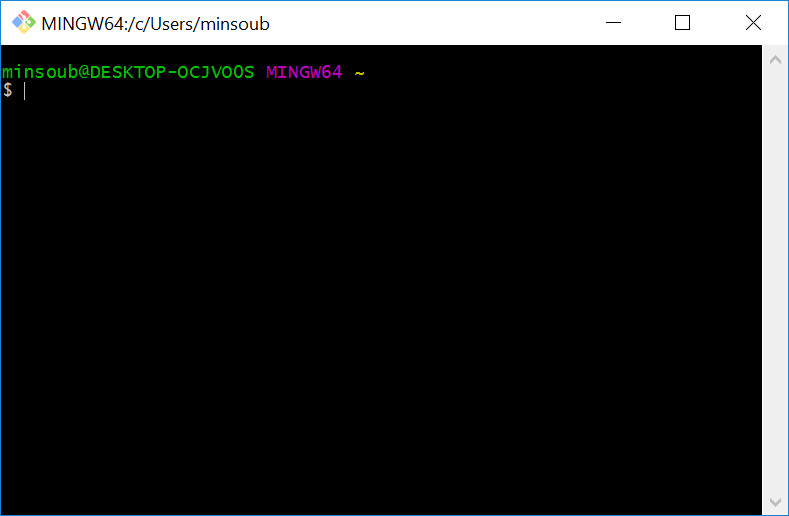


다운로드 버튼을 클릭하여 git를 다운로드 받는다.

설치버전은 Git-2.12.1-64-bit.exe 파일을 다운로드 받아서 설치한다.



Git command 창 실행모습



## Git Bash 사용하기

### git config

Git을 설치했을 때 가장 먼저해야 하는 것은 GitHub 사이트에 등록된 user name과 e-mail 주소이다. 이것은 git commit 시마다 사용하는 정보로서 초기에 설정해 놓아야 한다.

$ git config -–global user.name “Joung minsoub”

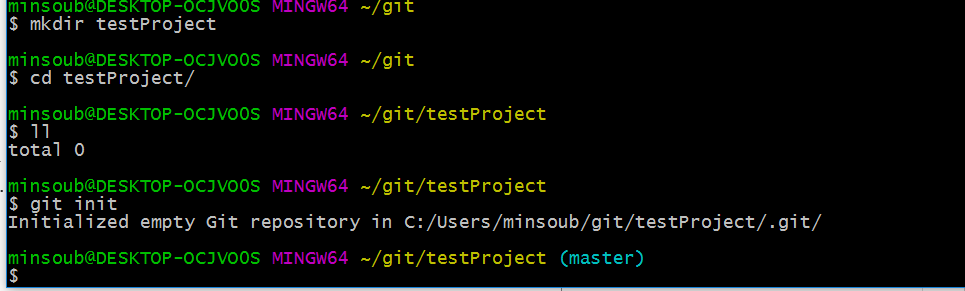
$git config –-global user.email [minsoub@gmail.com](mailto:minsoub@gmail.com)



### git init

우선 로컬저장소로 사용할 폴더를 하나 생성한다. 윈도우 탐색기로 D 드라이버에다 폴더를 생성했다. ( D:\WindLidarRepo )

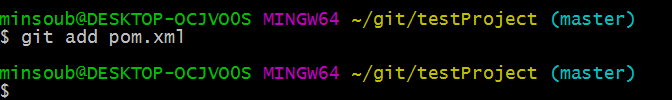
그리고 나서 Git Bash 실행창에서 로컬저장소로 이동한 후 **git init** 명령어를 실행한다.



### git add

GitHub에 올릴 테스트 아무 파일 하나를 로컬저장소에 붙여 넣는다. testProject 폴더에 아무 파일을 하나 복사해서 넣는다. 로컬저장소에 파일을 복사한다고 해서 git 저장소에 바로 연결(tracked)되는 것은 아니다. SVN에서도 새로운 파일을 인식하기 위해서 add 하는 절차가 있듯이 git도 add관련 명령어가 있다. 그리고 Git에서는 **준비영역(Staging area)**으로 add한다라고 설명하고 있다.

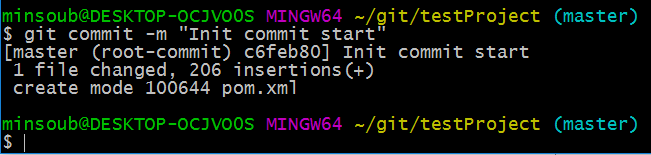
$git add pom.xml



### git commit

git add 명령어도 새로운 파일을 인식했다면 commit으로 버전관리를 시작하게 된다. 하지만 git commit은 원격저장소인 GitHub에 반영되는 것이 아닌, 로컬저장소에만 저장되는 것이다.

$git commit –m “Init commit”

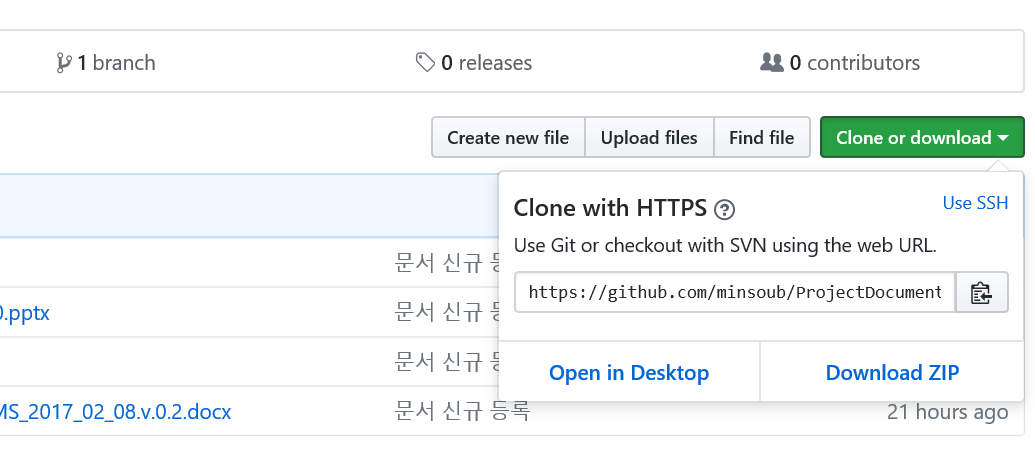


### git remote

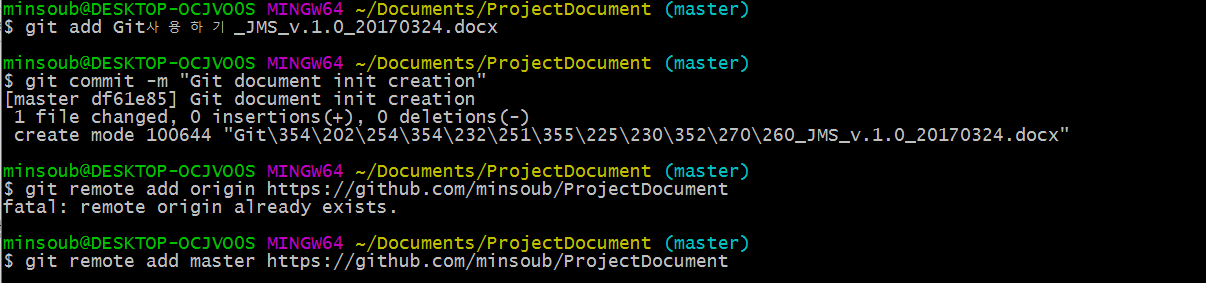
remote 레파지토리(repository)를 셋팅한다. 여기서는 https로 등록한다.

$git remote add origin <https://github.com/minsoub/testProject.git>

위의 주소는 GitHub 사이트에 가서 자신이 만든 repository 화면에 들어가면 아래와 같이 URL 정보가 나온다.



위의 명령을 수행한 후 서버에 파일이 올려져 있는지 확인한다.



기존에 Repository에 등록되어 있으면 origin으로 등록하면 에러가 발생할 것이다. 신규가 아니 기존에 사용하고 있는 Repository에 파일을 등록하려면 아래를 수행한다.

### git push

마지막 단계로 원격저장소에 파일을 적용하는 일이 남았다. git push 명령어를 실행한다.

$git push –u origin master

Username for “<https://github.com>”:

Password for “<https://minsoub@github.com>”:

Counting object: 3, done

Delta compression using up to 4 threads.

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

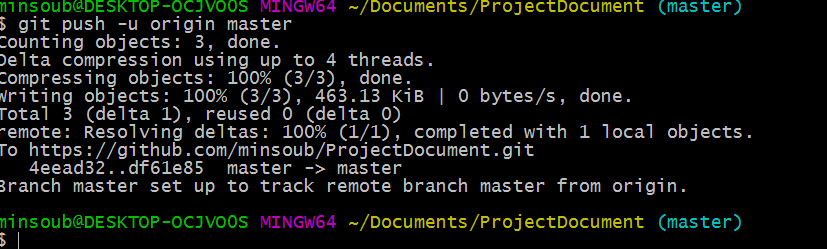
Writing objects : 100% (3/3), 407 bytes, done.

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

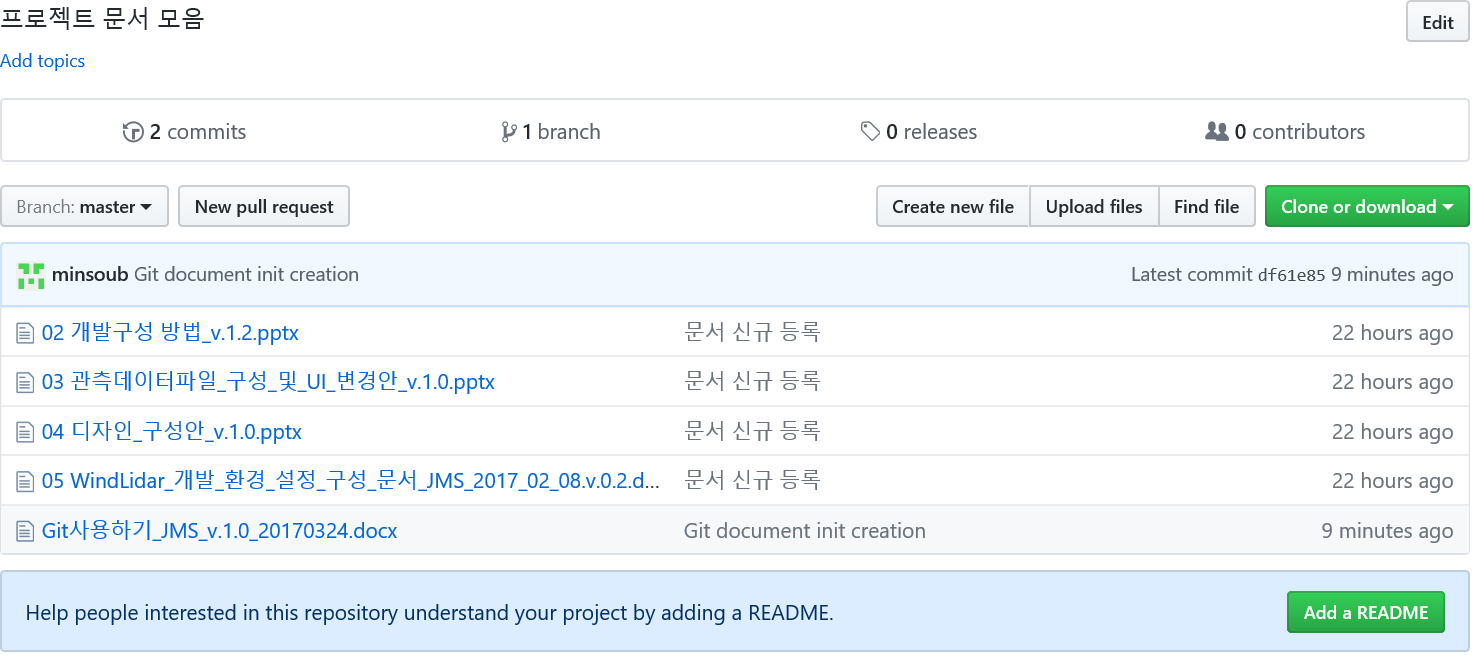
To <https://github.com/emflant/smaple.git>

\*[new branch] master -> master

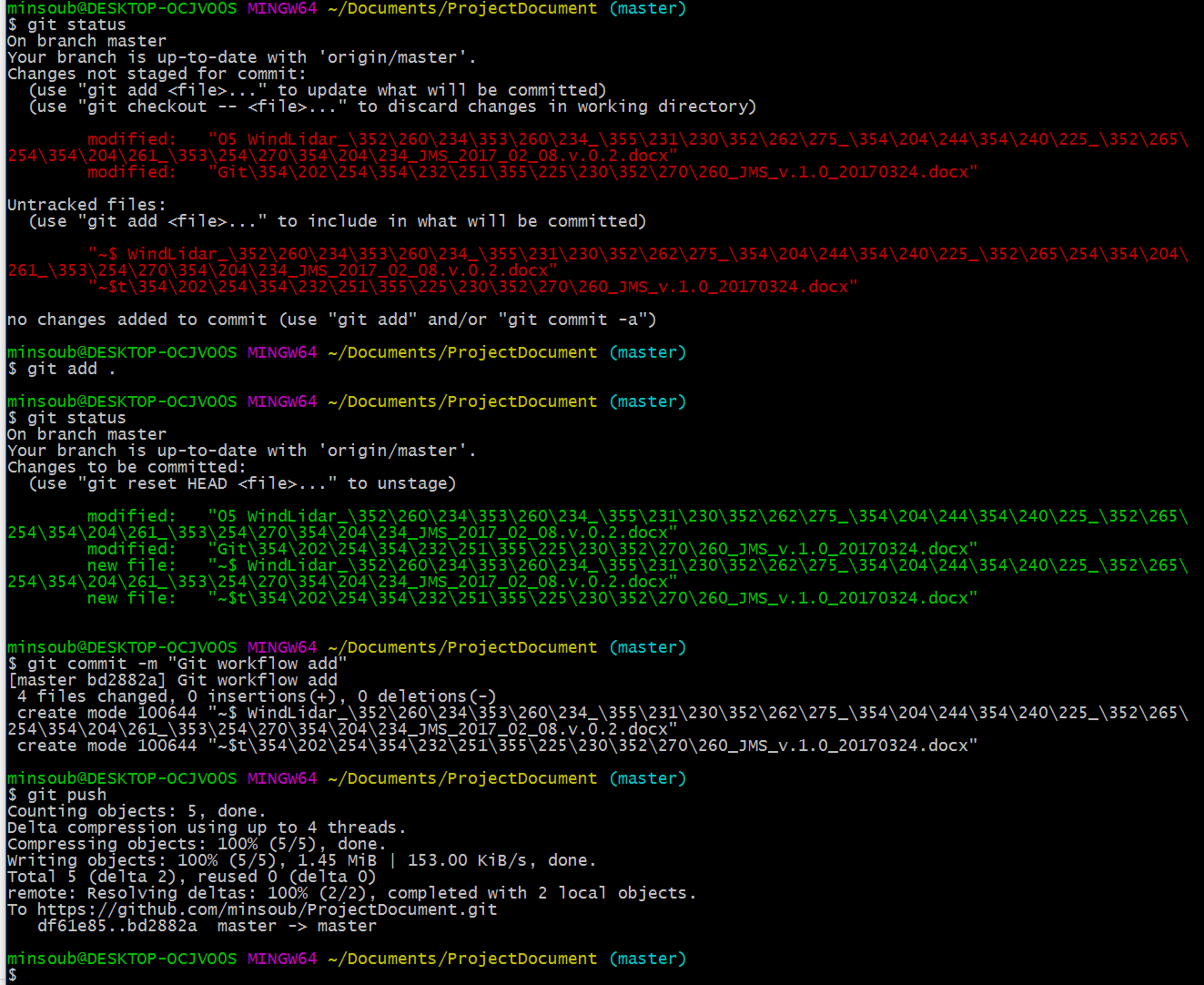
Branch master set up to track remote branch master from origin



파일 확인



수정 파일에 대해서 GitHub에 올리기



## Git 사용방법

참조 URL : http://rogerdudler.github.io/git-guide/index.ko.html

### 새로운 저장소 만들기

폴더를 생성한 후 git command 창으로 해당 폴더로 이동한 후 아래의 명령을 수행한다.

$ git init

위의 명령으로 새로운 git 저장소가 만들어진다.

### 저장소 받아오기

#### 로컬 저장소 복제하기

로컬 저장소를 복제(clone)하려면 아래 명령을 수행한다.

$ git clone /로컬/저장소/경로

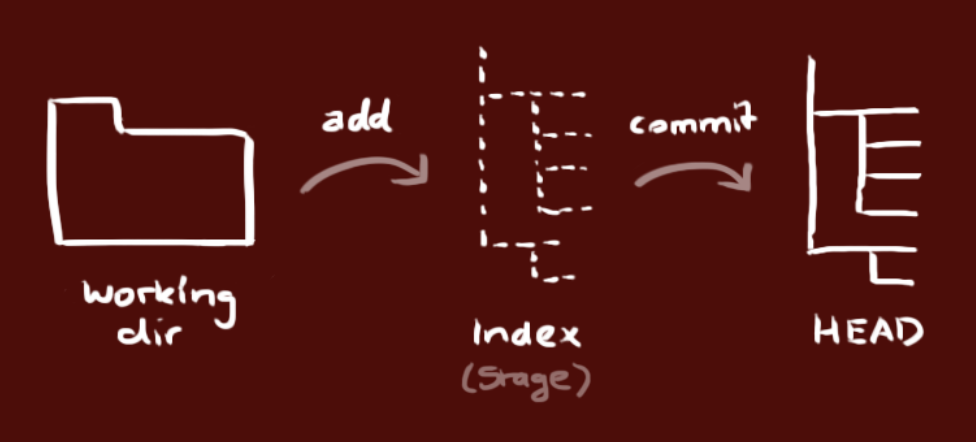
#### 원격 저장소 복제하기

원격 서버의 저장소를 복제하려면 아래 명령을 수행한다.

$ git clone 사용자명@호스트:/원격/저장소/경로

### 작업의 흐름

로컬 저장소는 git이 관리하는 세 그루의 나무로 구성되어 있다. 첫번째 나무인 작업 디렉토리(Working Directory)는 실제 파일들로 이루어져 있고, 두번째 나무인 인덱스(Index)는 준비 영역(staging area)의 역할을 하며, 마지막 나무인 HEAD는 최종 확정본(commit)을 나타낸다.



### 추가와 확정(commit)

변경된 파일은 아래 명령어로 (인덱스에) 추가할 수 있다.

$ git add <파일 이름>

$ git add \*

위의 명령은 git의 기본 작업 흐름에서 첫 단계 해당한다. 하지만 실제로 변경 내용을 확정하려면 아래의 명령을 수행해야 한다.

$git commit –m “comment”

이렇게 하면 변경된 파일이 HEAD에 반영된다. 하지만, 원격 저장소에는 아직 반영되지 않았다.

### 변경 내용 발행(push)하기

현재의 변경 내용은 아직 로컬 저장소의 HEAD안에 머물고 있다. 이 변경 내용을 원격 서버로 올리려면 아래의 명령을 수행해야 한다.

$ git push origin master

(다른 가지를 발행하려면 master 를 원하는 가지 이름을 바꾸어준다)

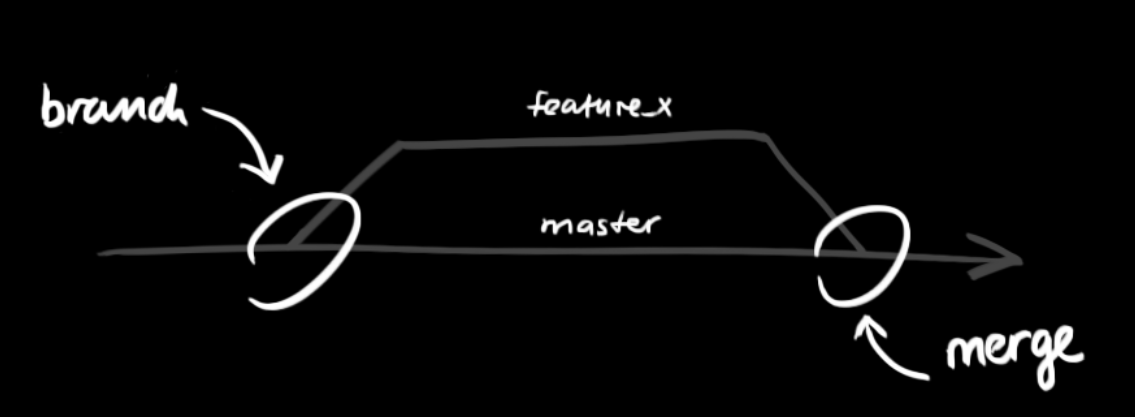
만약 기존에 있던 원격 저장소를 복제한 것이 아니라면, 원격 서버의 주소를 git에게 알려 줘야 된다.

$ git remote add origin <원격 서버 주소>

이제 변경 내용을 원격 서버로 발행할 수 있다.

### 가지(branch)치기

가지는 안전하게 격리된 상태에서 무언가를 만들 때 사용한다. 저장소를 새로 만들면 기본으로 master가지가 만들어진다. 이제 다른 가지를 이용해서 개발을 진행하고, 나중에 개발이 완료되면 master가지로 돌아와 병합하면 된다.



아래 명령으로 “feature\_x”라는 이름의 가지를 만들고 갈아탄다.

**# git checkout –b feature\_x**

아래 명령으로 master가지로 돌아 올 수 있다.

**# git checkout master**

아래 명령으로 가지를 삭제할 수 있다.

**# git branch –d feature\_x**

새로 만든 가지를 원격 저장소로 전송하기 전까지는 다른 사람이 접근할 수 없다.

**# git push origin <가지 이름>**

### 갱신과 병합(merge)

로컬 저장소를 원격 저장소에 맞춰 갱신하려면 아래 명령을 수행하낟.

**# git pull**

이렇게 하면 원격 저장소의 변경 내용이 로컬 작업 디렉토리에 받아지고(fetch), 병합(merge)된다. 다른 가지에 있는 변경 내용을 현재 가지(예를 들어, master가지)에 병합하려면 아래 명령을 실행한다.

**# git merge <가지이름>**

첫번째 명령이든 두번째 명령이든, git은 자동으로 변경 내용을 병합하려고 시도한다. 문제는, 항상 성공하는게 아니라 가끔 충돌(conflicts)이 일어나기도 한다. 이렇게 충돌이 발생하면, git이 알려주는 파일의 충돌 부분을 직접 수정해서 병합이 가능하도록 해야 한다. 충돌을 해결했다면, 아래 명령으로 git에게 아까의 파일을 병합하라고 알려준다.

# git add <파일 이름>

변경 내용을 병합하기 전에, 어떻게 바뀌었는지 비교 해 볼수도 있다.

# git diff <원래 가지> <비교 대상 가지>

### 꼬리표 (tag) 달기

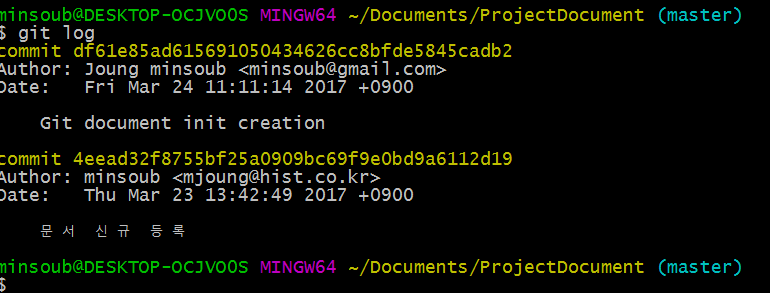
소프트웨어의 새 버전을 발표 할 때마다 꼬리표를 달아 놓을 수가 있다. 이 기능은 SVN 등에도 이미 존재하는 기능이다. 아래 명령을 실행하면 새로운 꼬리표인 1.0.0을 달 수 있다.

# git tag 1.0.0 1be21d63ff

위 명령에서 1be21d63ff 부분은 꼬리표가 가리킬 확정본 식별자이다. 아래 명령으로 확정본 식별자를 얻을 수 있다.

# git log

확정본 식별자의 앞부분 일부만 입력해도 꼬리표를 붙일 수 있지만, 그 일부분이 반드시 고유하다는 조건이 필요하다.



### 로컬 변경 내용 되돌리기

실수로 무언가 잘못한 경우, 아래 명령으로 로컬의 변경 내용을 되돌 릴 수 있다.

# git checkout -- <파일 이름>

위 명령은 로컬의 변경 내용을 변경 전 상태(HEAD)로 되돌려준다. 다만, 이미 인덱스에 추가된 변경 내용과 새로 생성한 파일은 그대로 남는다.

만약, 로컬에 있는 모든 변경 내용과 확정본을 포기하려면, 아래 명령으로 원격 저장소의 최신 이력을 가져오고, 로컬 master 가지가 저 이력을 가리키도록 할 수 있다.

# git fetch origin

# git reset –hard origin/master

### 기타 명령어

Git의 내장 GUI : gitck

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

### 저장소의 잘 못된 폴더 삭제

# git rm –f –cached folder\_name

# git commit –m “commit description : remove folder”

# git push origin master // 변경 내용 저장

그런 다음 git status 입력하면 현재의 상태를 확인하게 된다.

# git fetch origin // 동기화

# GIT Workflow

참조 URL : <http://dalinaum-kr.tumblr.com/post/15516936704/git-work-flow>

http://nvie.com/posts/a-successful-git-branching-model/

## Development Branch

GIT을 쓰는데 몇가지 작업 흐름을 생각해 볼 수 있다. 먼저 개발 브랜치로 development를 쓰는 것이다.



마스터 브랜치 master는 언제나 안정 버전으로 릴리즈 예정인 코드 중의 하나임을 의미한다. 필요에 따라 릴리즈 브랜치와 개발 브랜치가 구별이 없을 경우에는 (웹 서비스 등에서 릴리즈 시간이 의미가 없는 경우) development 브랜치 단계를 생략할 수도 있다. 또 다른 작업 흐름 중 하나는, 별도의 릴리즈 리포지토리가 있는 경우이다. 이 경우 master에서 머지된 것을 일정 릴리즈 주기로 다른 리포지토리에 푸쉬 하면 된다.

## Function Branch



그림에서 기능 브랜치는 development 에서 떼어와서 development에 머지되는 형태를 보이고 있다. 만약 master 브랜치가 없는 경우에는 master에서 떼어와서 master에 머지하는 형태가 되어도 된다. 중요한 것은 여러 사람의 작업이 별도로 떨어진 브랜치에서 이루어진다는 것이다.

기능 브랜치에 대해서 컨벤션을 정하는 것도 좋다. 예를 들어 feature-접두어가 들어간 브랜치를 만드는 것이다. 아래와 같이 브랜치를 만들 수 있다.

# git checkout –b feature-https development

Development 브랜치로부터 feature-https 브랜치를 만들어 https 대응 버전을 작업하는 것이다. 기능 브랜치에 대해 명확한 정책이 있다면 저장소로 푸쉬해도 좋다. 이 방법이 추천된다. 서로의 작업 상황을 외부에서 웹 인터페이스로 확인할 수 있고, 로컬로도 받아서 볼 수 있고 여러 번 서로 피드백을 나눌 수 있을 것이다.

그렇지 못해 로컬 브랜치로 만들더라도 의미가 있다. 개발 중에 저장소를 여러 번 fetch나 pull하게 될 때 그 과정이 훨씬 편안해 질것이다.

Merge를 할 때 no-off를 붙여 주시면 조금 더 명시적인 히스토리를 남길 수 있다.



## Release Branch

릴리즈 브랜치를 별도로 만들 수 있다. 특정 버전의 릴리즈를 준비하는 브랜치이다. 접두어로 release-를 쓰면 아래와 같이 된다.

# git checkout –b release-0.90 development

이렇게 development 브랜치로부터 분리시켜 릴리즈를 준비하는 것이다.

# git checkout master

# git merge –no-ff release-0.90

# git tag –a 0.90

이렇게 릴리즈 브랜치를 분리시키면 개발 브랜치를 작업할 팀과 릴리즈 브랜치를 작업할 팀을 나누어 안정 버전 릴리즈의 개발을 함께 준비할 수 있다.

## HotFix Branch

릴리즈 된 버전의 문제를 해결하기 위한 브랜치이다.



패치를 같이 development에 머지할 수도 있고 릴리즈 브랜치가 있다면 릴리즈 브랜치에 머지한 후에 해당 릴리즈 브랜치의 작업이 끝났을 때 master와 development에 같이 머지하는 방법도 있다. 보통은 릴리즈 브랜치가 진행되는 동안에 잡은 버그가 있어 development에 머지할 필요가 생기기 때문입니다.



## 도구와 응용

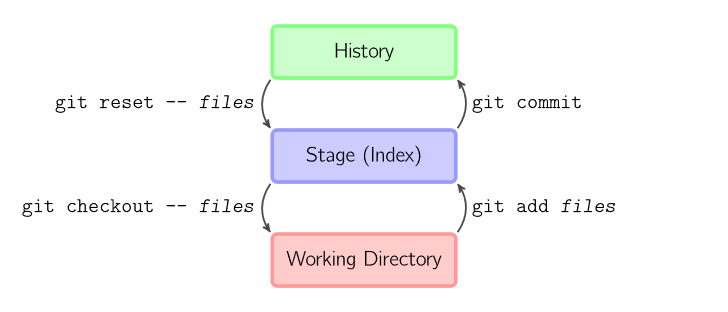
이 작업을 도와주기 위한 도구, git-flow가 있습니다. GitHub는 development나 release- 브랜치를 별도로 두지 않는다. 명시적인 릴리즈가 없고 최종 결과물은 다른 리포지터리(프로덕션)에서 서비스하기 때무이다.

GitHub 팀은 흥미롭게도 pull request기능을 이용하여 코드 리뷰 도구로 활용한다. Pull request는 자기 자신(같은 리포지터리)에 적용할 수 있다. GitHub팀은 master작업의 결과를 반드시 pull request를 통해서 하도록 하고 있다.

# Visual Git Reference

URL : <http://marklodato.github.io/visual-git-guide/index-ko.html>

## 기초 사용법

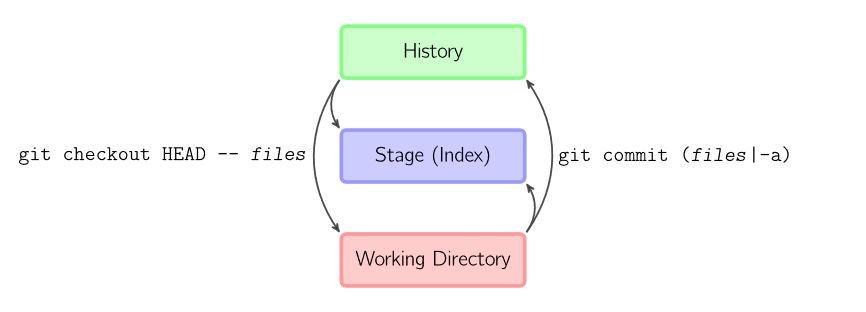


위의 네 가지 명령을 사용하여 작업 디렉토리, Stage 영역(Index), 히스토리(저장된 컷밋들) 사이에 파일을 복사한다.

* git add files 명령은(현재의) files 파일들을 Stage 영역으로 복사한다.
* git commit 명령은 Stage 영역의 현재 Snapshot 을 커밋으로 지정한다.
* git reset – files 명령은 마지막 커밋에서 Stage 영역으로 files 파일들을 복사한다. git add files 명령에 대한 되돌리기 명령이다. git reset 명령으로 모든 파일을 이전 커밋으로 복원할 수 있다.
* git checkout files 명령은 Stage 영역에서 작업 디렉토리로 파일을 복사한다. Stage 영역에 추가하지 않은 변경 내용에 대한 되돌리기 명령이다.

git reset –p나 git checkout -p또는 git add –p 명령과 같이 -p옵션을 사용하여 파일을 저장하기 않고 어떤 파일에 대해 명령을 적용할 지 대화형 명령을 사용할 수 있다.

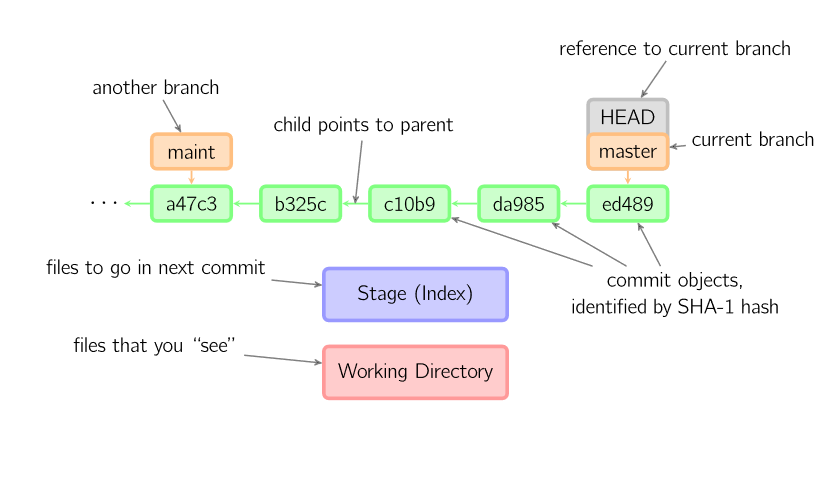
Stage 영역을 거치지 않고 직접 History로부터 파일을 Checkout 하거나, Stage 영역을 거치지 않고 직접 Commit을 할 수도 있다.



* git commit –a 명령은 마지막 커밋에 존재하는 모든 파일들에 대하여 git add 명령을 적용한 후 git commit 명령을 적용하는 것과 다르지 않다.
* git commit files 명령을 실행하면 마지막 커밋을 기반으로 files 의 변경된 내용을 포함하는 새로운 커밋을 하나 만든다. 이 때 files은 Stage 영역에 추가된다.
* git checkout HEAD – files 명령은 files을 마지막 커밋으로부터 Stage영역과 현재 작업 디렉토리에 동시에 복사한다.

## 관례

이 문서는 다음과 같은 그래프로 사용하여 Git 사용법에 대해서 설명한다.

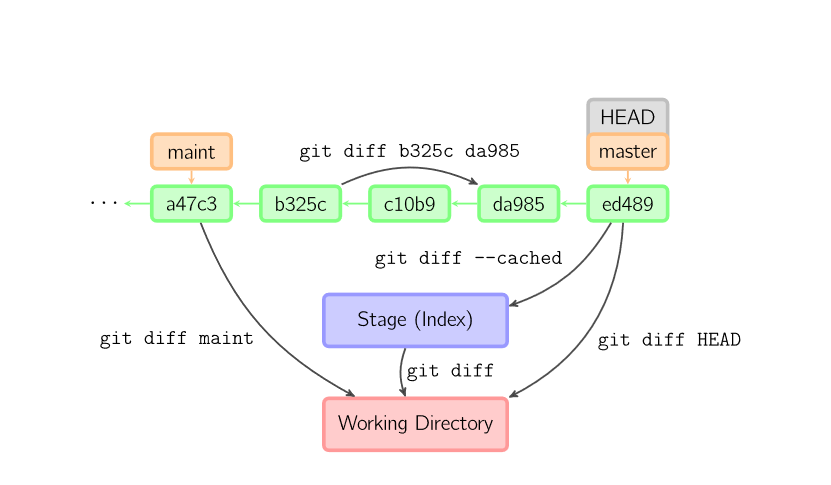


커밋은 5글자의 ID로 표현하며, 부모 커밋을 화살표로 가리킨다. 브랜치는 오렌지색이며 어떤 특정 커밋을 가리키고 있다. HEAD라는 이름으로 현재 브렌치를 가리킬 수 있다. 위의 그림에는 5개의 커밋이 있으며 ed489 커멋이 가장 최근의 커밋이다. master 브랜치(현재 선택한 브랜치)는 가장 최근의 커밋을 가리키고 있으며 maint 브랜치는 mastr 브랜치의 뿌리 부분(Ancestor)이다.

## 명령어 살펴보기

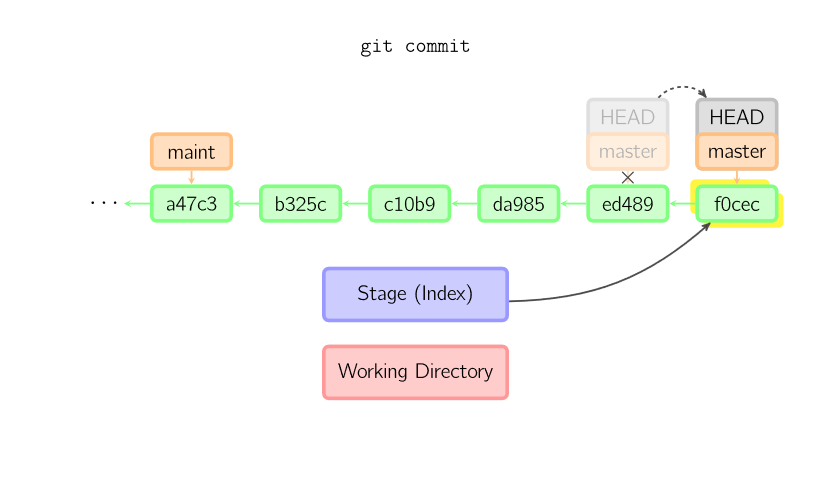
### Diff

커밋간의 변경된 사항을 살펴보는 방법은 여러가지가 있다. 아래 예제는 여러 방법 중 대표적인 것이다. 파일 이름을 옵션으로 지정하면 특정 파일에 대한 변경사항을 확인할 수 있다.

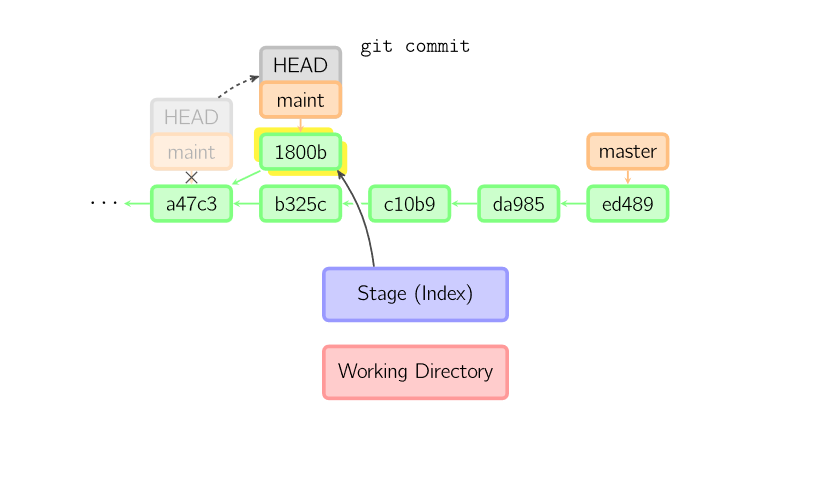


### Commit

커밋을 하면 Git은 Stage영역의 파일들과 부모 커밋 정보 그리고 현재 커밋 정보를 사용하여 새로운 커밋 개체(Commit Object)를 만든다. 그리고 현재 브랜치가 이 새로 만들어진 커밋을 가리키도록 만든다. 아래 그림에 보면 현재 브랜치는 master이고 명령을 실행하기 전에는 ed489 커밋을 가리키고 있다. 새로 커밋을 하게 되면 커밋의 부모가 ed489인 f0cec커밋이 만들어지고 master브랜치는 f0cec 커밋을 가리키게 된다.



이런 새로운 커밋이 추가되는 과정은 현재 브랜치가 다른 브랜치의 뿌리 부분(Ancestor)이라고 해도 가능한 일이다. 아래 그림을 보면 master 브랜치의 뿌리가 되는 maint 브랜치에서 커밋을 할 경우 I800b커밋이 만들어진다. 이렇게 되면 maint 브랜치는 master 브랜치의 직접적인 뿌리 부분이 되지는 않는다. 이 두개의 히스토리 내용을 합치기 위해서는 Merge(통합, 병합) 또는 Rebase 명령이 필요하다.

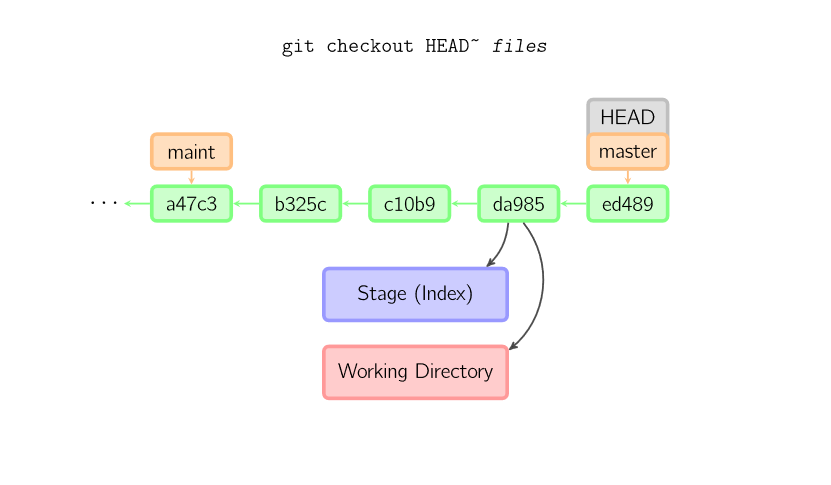


마지막 경우는 detached HEAD에서 커밋을 하는 것인데 아래에서 다시 다룬다.

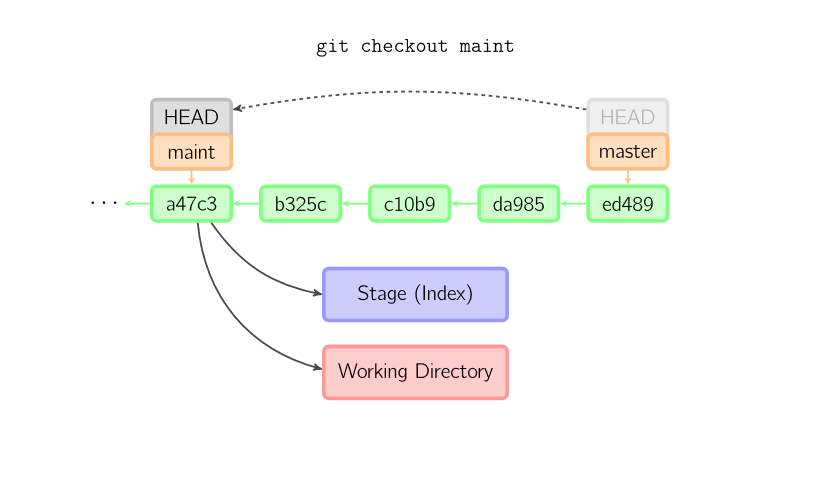
### Checkout

Checkout 명령은 히스토리나 Stage 영역으로부터 현재 작업 디렉토리로 파일을 복사하는 명령입니다. 또는 브랜치를 변경할 때 사용하기도 합니다.

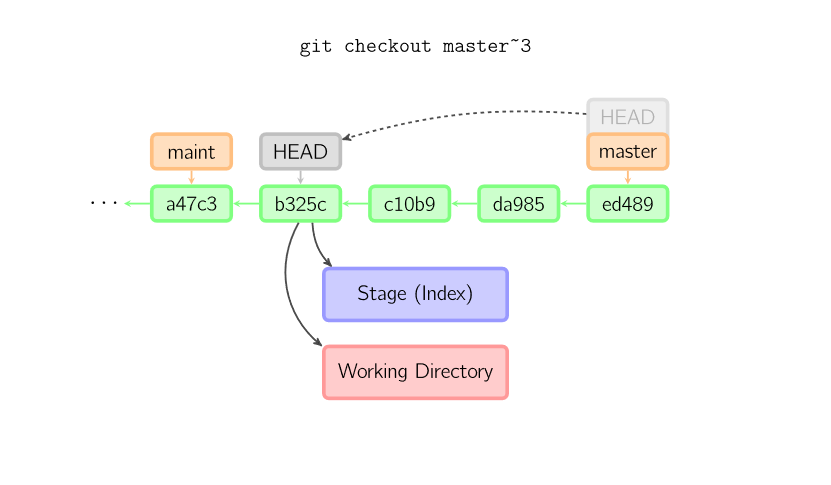
Checkout 명령에 파일 이름이 주어지면 (또는 –p 옵션) Git은 해당 파일을 주어진 커밋에서 Stage 영역과 작업 디렉토리로 복사한다. 예를 들어 git checkout HEAD~ foo.c 명령을 실행하면 HEAD~ 커밋(현재 커밋보다 한 단계 앞의 커밋)으로부터 foo.c 파일을 작업 디렉토리에 복사하고 stage 영역에도 추가한다. (커밋 이름을 지정하지 않으면 stage 영역에서 복사해온다.) 현재 브랜치가 가리키는 커밋은 바뀌지 않았다는 점을 주목한다.



파일 이름을 지정하지 않고 브랜치 이름만 지정하면 Git은 HEAD를 지정한 브랜치로 가리키도록 변경한다. 이것은 결과적으로 브랜치를 변경한 것과 같다. 따라서 자동으로 Stage 영역과 작업 디렉토리의 내용은 해당 브랜치의 내용으로 변경된다. a47c3 커밋에 포함된 파일들을 현재 디렉토리로 복사할 것이며 이전 ed489커밋에는 포함되었지만 a47c3 커밋에 포함되어있지 않은 파이들은 삭제될 것이다. 두 커밋에 모두에 포함되지 않은 파일은 무시될 것이다.

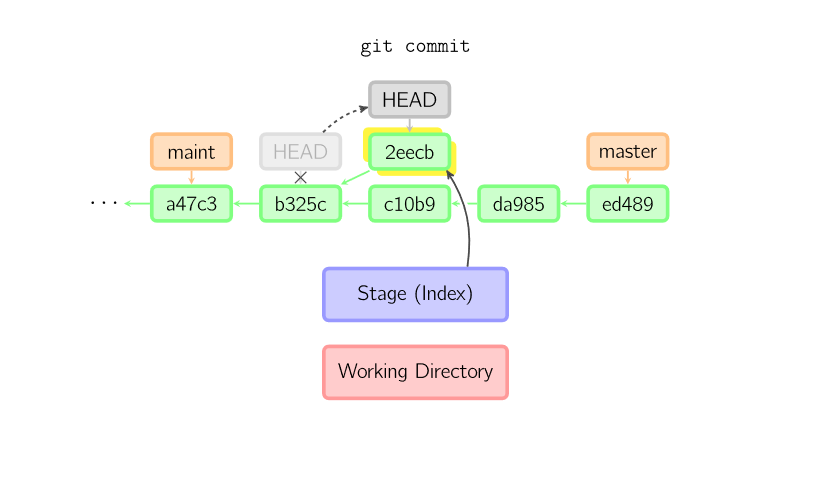


파일 이름을 지정하기 않고, (로컬) 브랜치 이름도 지정하지 않은 경우 – 즉, 태그, 리모트 브랜치, SHA-1 아이디, 혹은 master-3와 같은 유형을 인자로 지정했다면, detached HEAD라고 부르는 익명 브랜치(Anonymous Branch)를 사용하게 되는 프로젝트의 히스토리를 옮겨다닐 때 유용하게 사용할 수 있다. ‘Git 프로젝트’dml 1.6.6.1 버전을 컴파일 해보고 싶다면 git checkout v1.6.6.1 명령으로 소스를 Checkout하여 컴파일하고, 그 결과 바이너리들을 설치해 볼 수 있다. (v1.6.6. 은 브랜치는 아니고 태그이다). 그리고 나서 다시 git checkout master 명령으로 다른 detached 브랜치로 변경할 수도 있다. detached HEAD에서 커밋을 하면 경우가 좀 달라지는데 아래에서 다시 살펴볼 것이다.

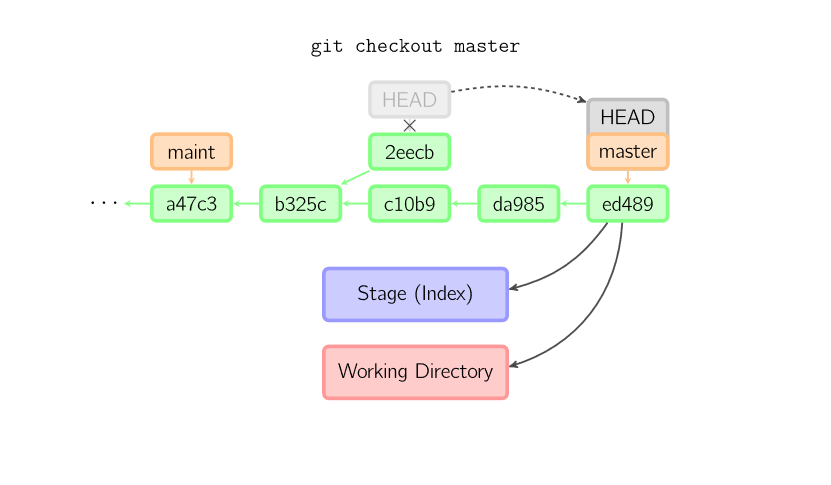


### Detached HEAD 에서 커밋하기

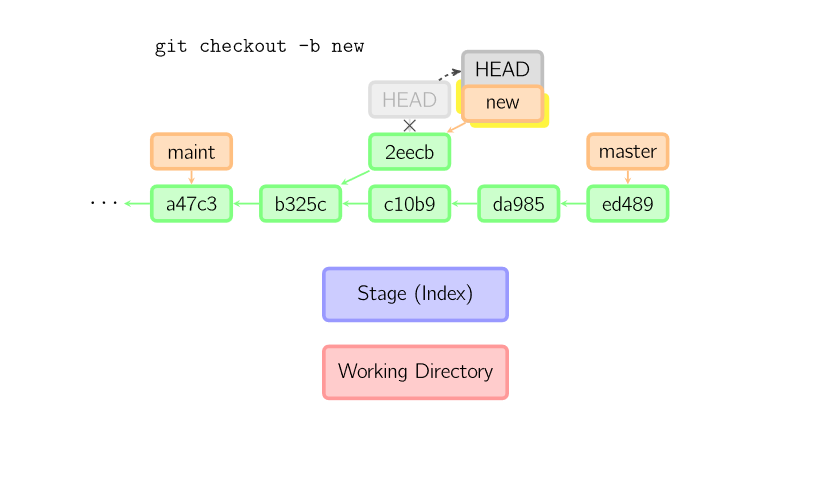
detached HEAD에서의 커밋도 별반 다르지 않다. 다만 아무 브랜치도 업데이트되지 않는다는 것만 다르다. (익명의 브랜치라고 생각해 볼 수 있다)



detached HEAD에서 다른 브랜치로 변경하게 되면(예를 들어 master 같은) detached HEAD의 커밋을 가리키는 어떤 이름도 갖지 못하게 되어 접근할 길을 잃고 말것이다. 아래 그림을 보면 브랜치를 변경 후 어떤 이름도 2eecb를 가리키고 있지 않다.



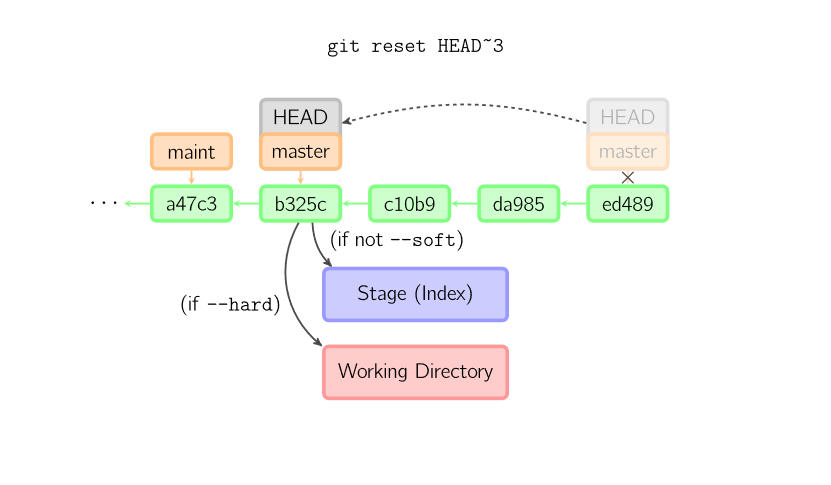
하지만 이 커밋을 가리키도록 새 브랜치를 만들 수 있는데 git checkout –b name 명령을 사용할 수 있다.



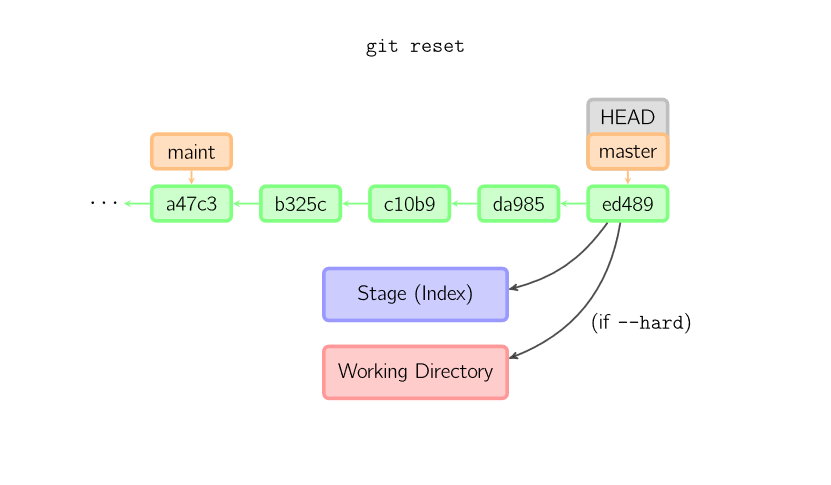
### Reset

Reset 명령은 현재 브랜치가 가리키고 있는 커밋을 이동시킬 때 사용하며 Stage 영역과 작업 디렉토리의 내용을 갱신한다. 또한 실제 작업 디렉토리 내용을 변경하지 않은 채로 이전 커밋에서 파일을 Stage 영역으로 복사할 때에도 사용한다.

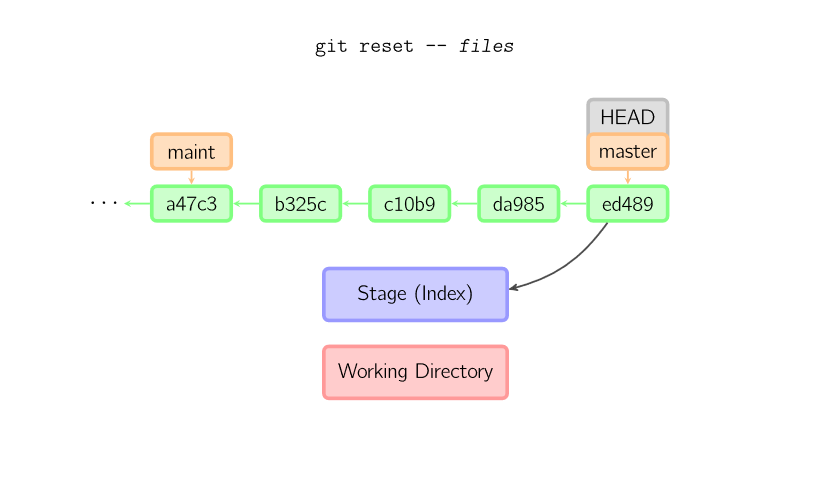
파일 이름 없이 커밋만 지정하는 경우 브랜치가 해당 커밋을 가리키도록 변경한다. Stage 영역 또한 해당 커밋에 맞게 갱신한다. –hard 옵션이 주어지면 작업 디렉토리 또한 갱신된다. --soft옵션이 주어지면 작업 디렉토리 및 Stage 영역 둘 다 갱신하지 않는다.



커밋 이름이 지정되지 않으면 HEAD를 대신 사용한다. 이 경우에는 브랜치가 가리키는 위치는 변경되지 않고 Stage 영역의 내용(또는 –hard 옵션이 주어지면 작업 디렉토리 까지)이 가장 마지막 커밋의 내용으로 갱신된다.

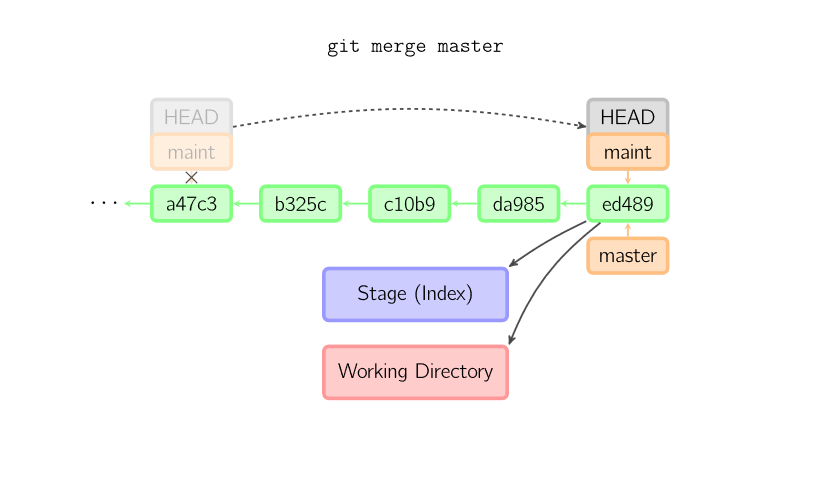


파일 이름을 지정하면 (또는 –p 옵션을 사용하면)Checkout 명령과 비슷한 역할을 합니다. 다만 Stage 영역만 갱신된다는 점이 다릅니다. (어떤 시점의 커밋으로부터 파일을 갱신할 지 HEAD 대신 커밋 이름을 지정하여 선택할 수 있다.)

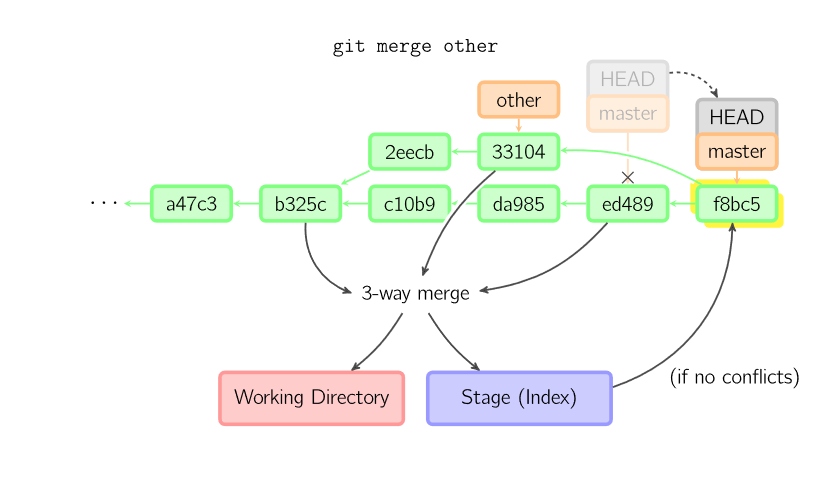


### Merge

Merge명령은 다른 커밋들을 하나로 합쳐서 새로운 커밋을 만듭니다. Merge 명령을 실행하기 전에 Stage 영역에 작업중인 파일이 없는지 꼭 확인해둔다. Merge 하는 경우에서 가장 간단한 경우는 Merge 할 대상이 현재 커밋의 직접적인 뿌리가 되는 경우 인데, 이 때는 합칠 내용이 없다. 다음은 현재 커밋이 Merge할 대상의 직접적인 뿌리가 되는 경우인데, 이 때는 fast-forward Merge가 실행되는데 간단히 가리키는 지점이 대상 커밋이 되고 대상 커밋의 내용을 Checkout 한다.

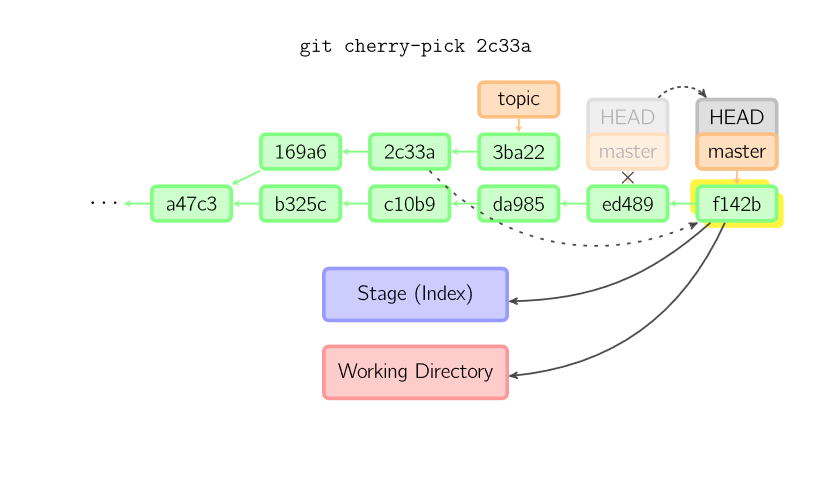


이젠 진짜 Merge를 살펴볼 것이다. 다른 Merge 전략을 선택할 수도 있지만 기본적으로 Git은 재귀적인 (Recursive) Merge전략을 사용한다. 이 전략은 현재 커밋(ed489), 대상이 되는 커밋(33104), 그리고 공통의 뿌리가 되는 커밋(b325c)을 가지고 3-way Merge를 수생한다. Merge한 결과는 작업 디렉토리와 Stage영역에 저장되며 부모가 여럿(33104, ed489)인 새 커밋을 만든다.



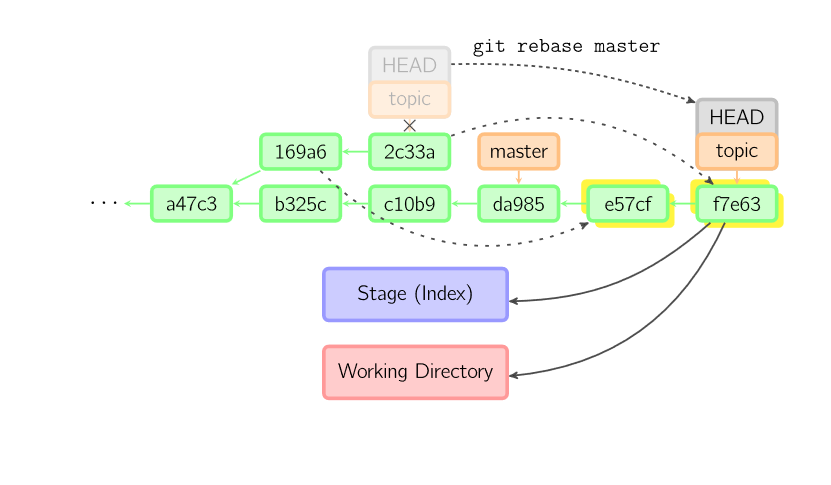
### Cherry Pick(열매 고르기)

Cherry-pick 명령은 커밋을 하나 꺼내서 현재 작업중인 브랜치 마지막 부분에 ‘복’'를 하면서 해당 커밋이 변경하는 부분을 적용하고 메시지나 저자 정보 등의 커밋 정보를 함께 저장한다.



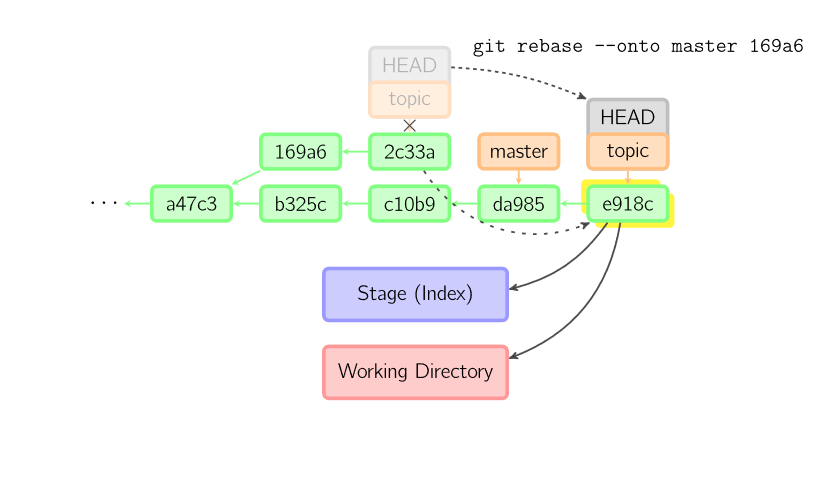
### Rebase

여러 브랜치를 하나로 모으고자 할 때 Rebase 명령을 Merge 명령 대신 사용할 수 있다. Merge 명령은 두 부모를 가지를 하나의 새 커밋을 만들기 때문에 히스토리가 직선적이지 않습니다. Rebase 명령을 사용하면 커밋들을 하나씩 순차적으로 적용해나가면서 히스토리를 직선으로 만들 수 있다. Cherry-pick 명령을 자동으로 한번에 수행하는 것이라고 보면 된다.



위의 Rebase 명령은 topic 브랜치에만 포함되어 있는 모든 커밋들(169a6와 2c33a)을 master브랜치에 추가한다. 커밋들을 추가하고 나서 topic 브랜치가 마지막 커밋을 가리키도록 이동한다. Rebase하고 나서 더 이상 가리킬(Reference)수 없는 커밋들은 쓰레기통으로 사라진다.

--onto옵션을 사용하면 Rebase에 사용할 커밋을 얼마나 오래 전 까지의 커밋을 사용할 지 제한할 수 있다. 아래 명령은 169a6 커밋 이후의 모든 커밋들(여기에서는 2c33a 커밋)을 master 브랜치에 적용시킨다.



추가로 git rebase –interactive 명령이 있는데 간단히 커밋을 적용하는 것 이외에도 더 복잡한 기능, 커밋에 대해서 Namely Dropping, Reordering, Modifying, Squashing을 할 수 있다. 이해하기 쉽게 그릴 수 있는 그림이 없어 부득이 메뉴얼 문서 git-rebase(1) (http://www.kernel.org/pub/software/scm/git/docs/git-rebase.html#\_interactive\_mode )링크를 참조한다.

### 기술적인 내용

파일의 실제 내용은 사실 Index(.git/index)나 커밋 개체(Commit Object)에 저장되는 것이 아니라, 개체 데이터베이스(Object Database, .git/objects)에 SHA-1 해시로 구분하여 blob 형태로 저장이 됩니다. Index는 파일 이름의 목록과 파일 blob을 가리키는 Hash를 저장하고 있습니다. 커밋에는 추가로 tree라는 형식의 데이터가 있는데 마찬가지로 Hash로 구분하고 디렉토리 구조를 담고 있습니다. 각 디렉토리는 포함된 파일 목록에 대한 tree 데이터를 담고 있다. 각 커밋은 가장 상위 디렉토리에 대한 tree 정보를 갖고 있어 커밋에 포함된 디렉토리 및 파일 정볼르 접근 할 수 있다.

detached HEAD에서 커밋을 만들게 되면 뭔가 만든 커밋을 가리킬 것이 필요한데 HeAD에 대한 reflog를 사용할 수 있다. 하지만 이 정보는 시간이 지나면 버려지기 때문에 결국 아무것도 가리키는 것이 없는 커밋은 git commit --amend명령이나 git rebase 명령으로 버려지는 커밋 처럼 버러지게 된다.

### 명령어들의 결과 확인

Visualizing Git Concepts with D3(<http://onlywei.github.io/explain-git-with-d3/#>)을 통해 git 명령어의 결과를 시각적으로 시뮬레이션하는 것처럼, 다음의 실습을 통해 저장소를 변경해 명령의 결과를 즉시 확인할 수 있다.