Table of Contents

책 소개

Git, Github 사용법을 설명합니다.

2018.11.29 송석원 softcontext@gmail.com

1. 깃 소개

Git

동일한 파일에 대한 여러 버전을 관리하는 `분산 버전관리 시스템(Distributed Version Control System)` 입니다. 원격 저장소와 참여하는 모든 개발자가 개별적으로 갖고 있는 로컬 저장소로 구성됩니다. 2005년 탄생했으며 리눅스 커널 개발자인 리누스 토르발즈가 기존 버전관리 프로그램들의 상용화에 반대하여 2주만에 만든 것으로 유명합니다.

- 공유물을 원격저장소와 협업자의 컴퓨터의 각각 저장하여 자료의 손실을 막아주는 도구
- 개발자들의 협업을 지원하는 도구
- 변화 이력을 추적할 수 있는 도구
- 창작물을 원하는 시점으로 되돌릴 수 있는 도구

깃에서의 논리적 단위는 커밋(Commit)으로 `커밋은 독립적인 변화의 묶음을 영속화한 것` 입니다. 또한 커밋은 단순히 커밋한 코드뿐만 아니라 관련된 메타 데이터도 포함합니다. 변화가 일어날 때마다 이전 커밋과의 변경점을 추적합니다. 커밋이 되면 해당 데이터는 정적인 상태로써 지울수는 있어도 절대로 변경할 수는 없습니다.

작동방식

깃의 데이터는 파일들의 Snapshot(그 순간의 상태의 기록물)입니다. 깃은 파일을 저장하거나 커밋하는 시점에서 변화를 감지합니다. 깃은 전체 소스를 대상으로 스냅샷을 남기지 않습니다. 파일이 달라지지 않았다면 깃은 성능을 위해서 파일을 저장하지 않고 링크만 저장합니다.

체크섬(커밋해쉬값)

깃은 커밋 명령으로 백업할 때 마다 유니크한 체크섬 값을 붙여서 데이터를 관리합니다. 체크섬은 원자적인 데이터 단위입니다. 깃은 SHA-1 해시를 사용하여 체크섬을 만듭니다. 체크섬은 40자 길이의 16진수 문자열입니다.

예: `24b9da6552252987aa493b52f8696cd6d3b00373`

세 개의 영역

Working Directory

프로젝트 파일들이 있는 디렉토리입니다. 물리적으로 프로젝트 폴더에서 .git 폴더를 제외한 파일 및 폴더를 의미합니다. 눈에 보이기 때문에 사용자가 편집하기 수월합니다. 워킹 디렉토리의 모든 파일은 `Tracked` 와 `Untracked` 상태로 구분됩니다. `Tracked` 파일은 깃의 관리대상으로써 `Unmodified` , `Modified` , `Staged` 상태 중 하나의 상태를 갖습니다. Tracked 파일이 아닌 모든 파일은 Untracked 상태입니다.

Staging Area(Index)

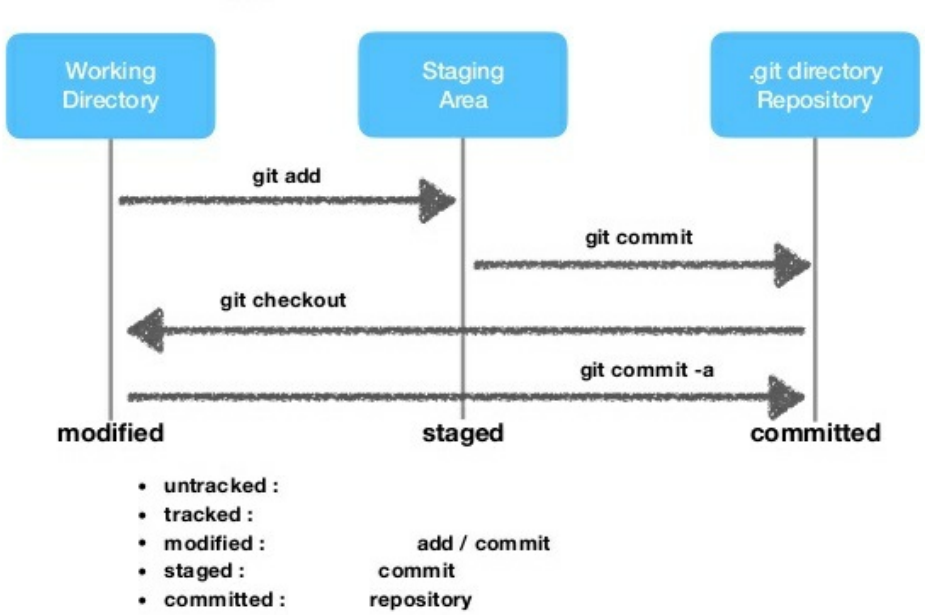
.git/index 파일입니다. 하나의 큰 바이너리 파일이고 곧 커밋할 파일에 대한 정보(파일 정보, SHA1 체크섬, 타임 스탬프)를 저장합니다. 커밋 대상들이 존재하는 공간입니다. Index 영역이라고 부르기도 합니다.

Index는 바로 다음에 커밋할 대상입니다. 이런 개념을 `Staging Area` 라고 부릅니다. `Staging Area` 는 사용자가 `git commit` 명령을 실행했을 때 깃이 처리할 것들이 있는 곳입니다.

Git Repository

.git/objects 폴더안에 존재합니다. 깃이 프로젝트의 메타데이터와 객체 데이터베이스를 저장하는 곳을 가리킵니다. 커밋된 내역들이 존재하는 공간입니다.

커밋을 보관하는 저장소는 압축된 BLOB 파일입니다. 저장소에 커밋은 압축된 한 덩어리로써 파일별로 구분할 수 없는 상태이므로 파일별 개별적인 복원은 지원하지 않습니다.



- Local Repository(로컬 저장소) 내 PC에 존재하는 개인 용 로컬 저장소입니다.
- Remote Repository(원격 저장소) 전용 서버에서 관리되며 여러 사람이 공유하는 자원을 위한 원격 저장소입니다. 관습적으로 `origin` 이라는 별칭으로 부릅니다.

특수 목적 포인터

HEAD

깃은 HEAD라는 특수한 포인터를 갖고 있습니다. 이 포인터는 현재 작업하고 있는 브랜치 또는 커밋을 가리키는 용도로 사용합니다.

HEAD는 현재 브랜치를 가리키는 포인터이며 브랜치는 브랜치에 담긴 커밋 중 가장 마지막 커밋을 가리키는 포인터입니다. 지금의 HEAD가 가리키는 커밋은 바로 다음 커밋의 부모가 됩니다.

```
HEAD --> master --> 커밋해쉬값(체크섬)
```

HEAD 포인터가 master 브랜치를 가리킵니다. master 브랜치 포인터가 유니크한 커밋해쉬값으로 특정 백업상태를 가리킵니다. 일반적으로 브랜치 포인터는 해당 브랜치의 최신 커밋을 가리키는 상태에서 사용합니다만 최근 몇 개의 커밋을 버리기 위해서 그렇지 않도록 조작할 수 있습니다.

```
HEAD --> 커밋해쉬값(체크섬)
```

HEAD 포인터가 특정 브랜치 포인터를 거쳐서 어떤 커밋을 지칭하는 것이 아니라 직접 특정 커밋을 가리키는 상태입니다. 이 상태를 `분리된 HEAD 모드` 상태라고 부릅니다. 주로 지난 커밋 중 하나로부터 새로운 브랜치를 만들고자 할 때 사용되는 방식입니다. 대부분의 경우 HEAD 포인터는 특정 브랜치를 지칭한 상태로 작업하게 됩니다.

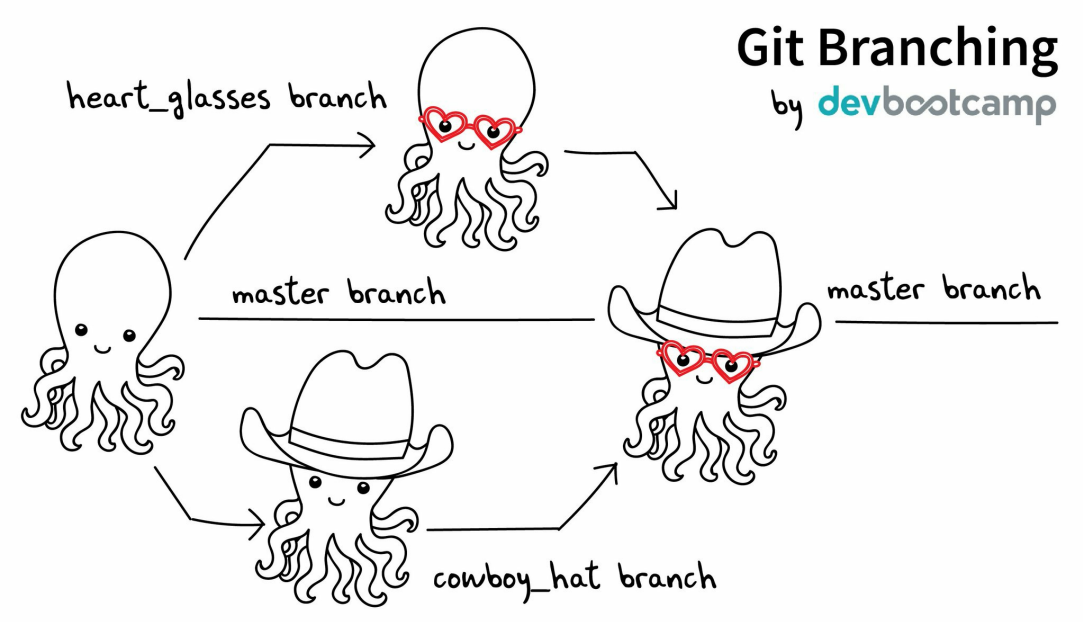
ORIG_HEAD

HEAD가 이전에 가리키던 커밋을 참조합니다. 커밋의 구성을 조작(rebase)할 때 이전 상태로 되돌리기 위한 참조용도로 사용합니다.

Branch(브랜치)

브랜치는 작업을 분리하여 무언가를 만들 때 사용합니다. 여러분이 저장소를 새로 만들면 기본적으로 master 브랜치가 만들어집니다. 다른 브랜치를 이용해서 개발을 진행하고 개발이 완료되면 master 브랜치에 병합해서 적용할 수 있습니다.

별도의 작업공간을 생성하여 작업하다가 마음이 바뀌면 이전 작업공간으로 휙~ 돌아갈 수 있다면 얼마나 멋질까요? 별도의 작업공간을 위해서 깃은 브랜치를 사용합니다. 별도의 작업공간을 위해서 매번 새로 폴더를 만드는 것이 아닙니다. 브랜치는 사실 특정 커밋(상태가 확정된 백업 본)을 가리키는 포인터입니다. 특정 브랜치로 체크아웃하게 되면 해당 커밍의 내역을 워킹 디렉토리로 복사해 옵니다. 이렇게 해서 하나밖에 존재하지 않는 워킹 디렉토리라는 작업공간의 상태가 변경될 수 있는 것 입니다. 이렇게 해서 필요할 때 마다 별도의 작업공간을 사용하는 듯한 효과를 얻을 수 있습니다.



여기 문어 그림이 있습니다. 문어에 안경을 그려주고 싶기도 하고 모자를 그려주고 싶기도 합니다. 안경을 그렸는데 마음에 들지 않을 때 원 상태로 되돌릴 수 있다면 참 좋겠습니다. 또한 모자를 그렸는데 마음에 들지 않으면 원 상태로 되돌릴 수 있기를 바랍니다. 각각 따로 작업을 하고 마음에 드는 경우에만 문어그림에 적용했으면 좋겠습니다. 어떻게 하면 가능할까요? 깃을 사용하면 가능합니다.

```
master --> 커밋(문어그림)
```

master 브랜치가 `문어그림을 보관한 상태(커밋)` 를 가리킵니다. master 브랜치로 체크아웃 하면 언제든지 문어그림의 복사본을 얻을 수 있습니다.

두개의 브랜치(포인터)를 추가합니다.

```
git branch heart_glasses
```

```
git branch cowboy_hat
```

이제 master, heart_glasses, cowboy_hat 브랜치 모두 같은 커밋을 가리킵니다. 따라서 master, heart_glasses, cowboy_hat 브랜치 중에 어느 브랜치로 체크아웃 하면 언제든지 문어그림의 복사본을 얻을 수 있습니다.

```
git checkout heart_glasses
```

heart_glasses 브랜치를 체크아웃 하면 저장소에 보관되어 있는 원본 문어그림의 복사본을 워킹 디렉토리에 추가합니다. 워킹 디렉토리에서 문어그림에 안경을 추가합니다. 여기서 중요한 것은 언제든지 master 브랜치로 체크아웃 하게되면 원본 문어그림의 복사본을 받을 수 있다는 부분입니다. 안경이 마음에 드는군요! 이 상태를 영속화 하기 위해서 다음 명령을 수행합니다.

```
git commit -a -m "하트 모양 안경 추가"
```

워킹 디렉토리는 하나 밖에 존재하지 않는 공간이기 때문에 commit 명령으로 변경된 상태를 저장하지 않고 다른 브랜치로 체크아웃 하게 되면 변경된 상태를 잃어 버리게 됩니다. 커밋을 수행하면 작업중인 브랜치는 새 커밋을 가리키도록 변경됩니다. 그래서 다른 브랜치에서 작업하다가 언제

든지 안경쓴 문어가 보고 싶다면 heart_glasses 브랜치로 체크아웃 하면 됩니다.

깃은 변화된 부분만을 처리하기 때문에 커밋을 자주한다고 데이터양이 급속도록 증가되지 않습니다. 다음으로 원본 문어그림에 모자를 그려 봅시다.

```
git checkout cowboy_hat
```

cowboy_hat 브랜치를 체크아웃 하면 저장소에 보관되어 있는 원본 문어그림의 복사본을 워킹 디렉토리에 추가합니다. 워킹 디렉토리에서 문어그림에 모자를 그려 줍니다. 작업결과가 마음에 들면 저장하기 위해서 커밋을 하시면 됩니다.

```
git commit -a -m "카우보이 모자 추가"
```

안경도 마음에 들고 모자도 마음에 드는군요. master 브랜치가 가리키는 문어그림에 heart_glasses 브랜치가 가리키는 커밋과 cowboy_hat 브랜치가 가리키는 커밋을 적용하고 싶습니다. master 브랜치로 이동해서 병합하면 됩니다.

```
git checkout master
git merge heart_glasses
git merge cowboy_hat
```

앞에서 살펴 본 내용을 정리해 봅니다. 브랜치는 특정 커밋을 가리키는 포인터입니다. 특정 브랜치로 체크아웃 하면 해당 브랜치가 가리키는 커밋의 복사본이 워킹디렉토리에 적용됩니다. 즉 브랜치를 이동한다는 것은 특정 커밋의 내용을 워킹디렉토리에 적용한다는 것을 의미합니다. 우리는 브랜치를 통해 특정 커밋의 내용을 워킹 디렉토리로 가져올 수도 있고 커밋해쉬값으로 지난 커밋들 중에 하나를 직접 선택해서 커밋의 내용을 워킹 디렉토리로 가져올 수도 있습니다. 지금 워킹 디렉토리에 있는 내용이 어떤 커밋으로부터 가져온 것인지 가리키는 도구로써 HEAD를 사용합니다.

```
git show HEAD
```

2. 설치 및 설정

Git 설치하기

입문자에게는 GUI 도구를 바로 사용하지 말고 커맨드 모드에서 먼저 학습하기를 권장합니다. GUI 도구는 도구마다 용어 및 구성이 상이하기 때문에 기초개념이 부족한 상태에서 사용하게 되면 자칫 혼란을 야기할 수 있습니다. 쉬워 보이는 길이 험지입니다. 커맨드 모드로 학습하는 것이 바른 길입니다. 바른길이 가장 빠른 길입니다.

모든 플랫폼을 위한 Git 다운로드 후 설치

깃을 설치할 때 자동적으로 `Git Bash` 도 같이 설치가 됩니다. 깃배쉬는 MINGW64 기술을 사용하므로 간단한 리눅스 명령들을 사용할 수 있습니다.

다운로드 주소 : `http://git-scm.com`

도움말 보기

`$ git help`

사용법을 살펴 봅니다.

`$ git help config`

config 사용법을 살펴 봅니다. 브라우저가 기동하면서 옵션들을 자세히 살펴볼 수 있습니다.

`$ git checkout --help`

특정 명령어 사용법을 살펴볼 수 있는 또 하나의 방법입니다.

사용환경 설정하기

커밋을 누가 했는지 기록하기 위해서 사용자 정보를 미리 설정하고 사용해야 합니다.

Global 설정

```
$ git config --global user.name [name]
```

유저의 커밋 기록에 부여되는 원하는 이름을 설정합니다.

```
$ git config --global user.email [email address]
```

유저의 커밋 기록에 부여되는 원하는 이메일을 설정합니다.

```
$ git config --global --list
```

설정정보를 확인합니다. `~/로그인아이디/.gitconfig` 파일에 저장됩니다.

Project별 설정

`.git` 폴더가 있는 프로젝트 마다 개별적으로 사용자 정보를 설정해 놓고 사용할 수 있습니다.

```
$ git config user.name [name]
```

```
$ git config user.email [email address]
```

3. 로컬 저장소

로컬 저장소 생성

새로운 저장소를 생성하거나 혹은 기존의 URL로부터 저장소를 획득할 수 있습니다. 프로젝트 개발 시 저장소 생성은 한번만 하면 됩니다.

`$ git init`

새로운 로컬 저장소를 생성합니다. `.git` 이라는 이름의 폴더가 생성됩니다. git init 명령으로 새 로컬 저장소를 만들면 기본적으로 `master` 라는 이름의 브랜치가 제공됩니다. 이 후 생기는 다른 브랜치들은 `master` 브랜치로부터 갈라지게 되어 같은 부모를 가지게 됩니다. 로컬 저장소를 새로 만든 상태에서는 아무런 커밋이 존재하지 않습니다.

`$ git clone [url]`

프로젝트와 프로젝트의 히스토리(커밋 내역)를 다운로드 합니다. `git clone` 명령을 통해 원격파일을 복사해오면, `origin` 이라는 이름으로 시작하는 클론해온 리모트 URL이 등록됩니다.

가장 먼저 알아차릴만한 변화는 우리의 로컬 저장소에 `origin/master` 라고하는 새 브랜치가 생긴겁니다. `origin` 는 원격저장소를 가리키는 별칭입니다. 이런 종류의 브랜치는 원격 브랜치라고 불립니다.

가장 최근 원격 원격저장소와 작업을 했을때를 기준으로 원격 브랜치는 원격 저장소의 상태를 반영합니다. `원격 브랜치는 로컬에서의 작업과 공개적으로 되고있는 작업의 차이를 이해하는데 도움을 줍니다.` 다른 사람들과 작업을 공유하기전에 반드시 해야 할 과정입니다.

원격 브랜치는 체크아웃을 하게 되면 `분리된 HEAD 모드` 로 가게되는 특별한 행동방식이 있습니다. Git은 여러분이 이 브랜치들에서 직접 작업할 수 없기 때문에 일부로 이렇게 처리합니다. 여러분은 작업을 하고 원격 저장소와 여러분의 작업을 공유해야합니다. 공유 작업을 한 이후에 원격 브랜치가 갱신됩니다.

```
git checkout origin/master; git commit -m "updated"
```

`origin/master` 를 체크아웃 하게되면 깃은 우리를 `분리된 HEAD 모드` 로 만들고 `새로운 커밋을 추가해도 origin/master를 갱신하지 않습니다.` 이것은 `origin/master` 가 원격 저장소가 갱신될 때만 갱신되어야 하기 때문입니다. 대부분의 경우 `origin/master` 에서 직접 커밋을 수행해야 할 이유는 없습니다. 로컬 저장소에 존재하는 원격브랜치는 원격 저장소와의 상태 비교를 위한 용도이지 새로운 커밋을 추가하는 용도가 아님을 기억하세요.

깃 추적 대상에서 제외하기

특정 파일 및 경로 밑에 존재하는 자원들을 깃 관리대상에서 제외시킵니다. 백업이 필요없는 대상을 깃에게 미리 알려주어 깃이 관리하지 않도록 조치합니다.

.gitignore

`.gitignore` 파일은 프로젝트에서 백업을 원하지 않는 파일들을 깃 관리에서 제외시킬수 있는 설정파일입니다. 이 파일도 커밋 대상에 포함시켜야 합니다. 특정 패턴으로 매칭되는 파일과 경로(폴더)를 관리대상에서 제외합니다.

```
# 이 기호 다음에 주석을 달 수 있습니다.

# 확장자가 .log로 끝나는 모든 파일들을 관리대상에서 제외합니다.
*.log

# 위 설정에도 불구하고 spring.log 파일은 관리대상에 포함합니다.
!spring.log

# 현재 디렉토리 바로 밑에 있는 schema.sql 파일을 관리대상에서 제외합니다.
# 서브 디렉토리 밑에 있는 schema.sql 파일은 관리대상에 포함합니다.
/schema.sql

# 파일명 temp-로 시작하는 모든 파일들을 관리대상에서 제외합니다.
temp-*

# build 폴더 밑에 존재하는 모든 파일들을 관리대상에서 제외합니다.
build/

# doc/note.txt 파일은 관리대상에서 제외되나
# doc/server/note.txt 파일은 관리대상에 포함합니다.
doc/*.txt

# doc 폴더부터 시작해서 모든 하부 폴더에 존재하는
# 확장자가 .pdf로 끝나는 모든 파일들을 관리대상에서 제외합니다.
doc/**/*.pdf
```

.gitignore 내용작성 도움받기

다음 사이트는 `.gitignore` 파일의 등록 내용을 자동으로 만들어 주는 서비스를 제공합니다.

`https://www.gitignore.io/`

관리대상에서 제외된 파일정보를 확인

```
$ git ls-files --others --ignored --exclude-standard
```

빈 폴더를 커밋에 포함시키기

기본적으로 비어있는 폴더는 깃의 관리대상이 아닙니다. 빈 폴더를 깃의 관리대상에 포함시키는 방법은 간단합니다. `.gitkeep` 파일을 빈 폴더에 배치하여 깃의 관리대상이 되도록 조치할 수 있습니다. 이럴 때 사용하는 `.gitkeep` 파일은 아무런 내용이 없어도 됩니다. `.gitkeep` 파일명은 관습이기에 무엇때문에 이 파일이 존재하는지 설명해야 하는 수고가 없습니다.

4. 원격 저장소

Github URL 패턴

1. HTTPS

`https://github.com/[사용자아이디]/[저장소명]`

2. SSH

`git@github.com:[사용자아이디]/[저장소명].git`

git 연결을 보다 안전하고 빠르게 하기 위해서 SSH 방식을 권장합니다.

1. SSH Key 생성

콘솔에서 `ssh-keygen` 명령을 사용합니다. SSH Key 값은 `~/[사용자 폴더]/.ssh/` 폴더 밑에 `id_rsa.pub` 파일로 존재합니다.

2. Github 설정

- Github 홈페이지에 로그인 합니다.
- Profile 중 Settings 메뉴를 선택합니다.
- Settings 화면 중 우측 사이드메뉴에서 `SSH and GPG keys` 메뉴를 선택합니다.
- SSH Keys 화면에서 `New SSH key` 버튼을 찾아서 클릭하고 `id_rsa.pub` 파일의 내용을 복사해 넣습니다.

3. 로컬 사용자 환경설정

Remote URL 설정이 HTTPS 방식이라면 SSH 방식으로 변경해야 합니다.

- origin의 Remote URL 변경방법 `$ git remote set-url origin git@github.com:[사용자아이디]/[저장소명].git`
- origin의 Remote URL 변경여부 확인 `$ git remote show origin`

5. 명령어 학습

브랜치 변경 명령이 거부될 때

현재 작업중인 브랜치에 커밋되지 않은 변경사항이 존재하는 상태에서 다른 브랜치로 체크아웃 하려할 때 거부당할 수가 있습니다. 다른 브랜치로 체크아웃 하는 명령은 충돌이 발생하지 않는 상황에서만 허락됩니다. 현재 브랜치가 C0이고 커밋하지 않은 변경내용이 있을 때 이동하려는 새 브랜치가 C0가 아닌 상황에서 내용이 다른 경우, 현재 브랜치의 변경 사항이 커밋되거나 숨겨질 때까지 브랜치를 전환 할 수 없습니다.

해결방법

이를 해결하는 방법은 여러가지가 있습니다. 다음 중 하나를 수행하면 됩니다.

- 변경사항을 커밋한 후 다른 브랜치로 이동합니다.
- 커밋되지 않은 변경사항을 `stash` 명령으로 임시로 보관한 후 다른 브랜치로 이동합니다.
- `git checkout -f 다른브랜치` 옵션을 추가하여 변경사항을 버리면서 다른 브랜치로 이동합니다.
- `git reset --hard HEAD` 또는 `git checkout HEAD -- [file]` 명령으로 변경사항을 버린 후 다른 브랜치로 이동합니다.

파일 관리

버전이 부여된 파일들을 재배치하거나 제거합니다.

`$ git rm [file]`

워킹 디렉토리로부터 파일을 제거하고 삭제기록을 스테이지 영역에 등록 합니다.

`$ git rm --cached [file]`

스테이지에 등록된 파일을 취소합니다. 워킹 디렉토리의 파일은 삭제하지 않습니다. `.gitignore` 파일에 무시대상으로 등록하는 작업을 잊어버린 경우에 사용합니다.

$ git mv [file-original] [file-renamed]

파일명을 변경하고 스테이지 영역에 등록 합니다.

임시 저장

때때로 작업을 하다보면 작업이 완료되지 않아서 커밋하기는 싫지만 버릴 수도 없는 상황에 직면하게 됩니다. 커밋하기에는 불완전한 변경 사항을 임시로 보관하고 복원 시키는 방법입니다. 주로 현재 작업중인 브랜치에서 변경 내용을 커밋할 수 없는 상태인데 급하게 다른 브랜치로 이동해야 하거나 커밋 변경을 해야할 때 사용합니다.

$ git stash

추적중인 수정된 파일들을 임시로 저장합니다. 수정된 내용은 제거됩니다. 기본적으로 WIP 명칭으로 저장됩니다. -u 옵션을 추가하여 새롭게 추가한 파일도 함께 stash 영역에 저장하도록 조치할 수 있습니다.

$ git stash save "your message here"

임시로 저장할 때 메시지를 추가해 놓으면 나중에 작업내역을 복원할 때 작성한 메시지가 구분하는 작업에 도움을 줍니다.

$ git stash list

모든 보관한 사항들의 리스트를 보여줍니다.

```
stash@{0}: WIP on master: 049d078 added the index file
stash@{1}: WIP on master: c264051 Revert "added file_size"
stash@{2}: WIP on master: 21d80a5 added number to log
```

$ git stash apply stash@{2}

보관한 이름을 입력하면 골라서 적용할 수 있습니다. 이름이 없으면 깃은 가장 최근의 보관한 내역을 적용합니다. apply 옵션은 단순히 Stash를 적용하는 것뿐입니다. Stash는 여전히 스택에 남아 있습니다. `git stash drop` 명령을 사용하여 해당 Stash를 제거할 수 있습니다.

깃은 Stash를 적용할 때 Staged 상태였던 파일을 자동으로 다시 Staged 상태로 만들어 주지 않습니다. git stash apply 명령을 실행할 때 --index 옵션을 주어 Staged 상태까지 적용할 수 있습니다.

$ git stash pop

가장 최근에 숨겨진 파일을 복원합니다.

$ git stash drop

가장 최근에 숨겨진 변경 사항을 버립니다. stash 아이디를 추가하여 특정 stash를 삭제할 수 있습니다.

$ git stash clear

모든 stash 내역을 삭제합니다.

상태 관리

커밋은 깃 저장소에 디렉토리에 있는 모든 파일에 대한 스냅샷을 기록하여 영속화 하는 것입니다. 깃은 가능한 한 커밋을 작게 만들고자 하기때문에, 커밋할 때마다 디렉토리 전체를 복사하지는 않습니다. 각 커밋은 저장소의 이전 버전과 다음 버전의 `변경내역의 차이(Delta)` 만을 저장합니다. 그래서 커밋은 앞서서 수행한 부모 커밋을 가리키게 됩니다.

$ git status

커밋 대상의 모든 새로운 파일 혹은 수정된 파일 리스트를 보여줍니다. 추가로 현재 상태에서 작업을 취소할 수 있는 명령어를 보여줍니다.

$ git checkout [branch]

지정한 브랜치로 이동합니다. 작업 대상인 커밋이 변화지 않는 경우, 수정되거나 추가된 파일은 유지됩니다.

$ git checkout (HEAD) -- [file]

워킹 디렉토리에서 파일의 변경 부분을 버립니다. 최근 커밋된 버전으로 되돌리는 방법입니다. `checkout` 명령은 언제나 `--hard` 방식입니다. 따라서 최근 커밋이 인덱스에 반영되고 그 다음 작업으로 워킹 디렉토리에 반영됩니다. `HEAD` 는 생략할 수 있습니다.

$ git checkout [커밋해쉬값]

커밋해쉬값이 C1 이라고 할 때, `HEAD -> master -> C1` 상태에서 위 명령을 수행하면 `HEAD -> C1` 상태가 됩니다. 결국 HEAD가 브랜치에서 분리됩니다. HEAD의 분리는 지난 커밋을 기준으로 새 브랜치를 만들고자 할 때 주로 사용됩니다.

$ git checkout master^

master 위에 있는 부모를 체크아웃 합니다. HEAD는 브랜치와 분리되고 직접 커밋을 가리키게 됩니다. ^(캐럿) 한개는 한 단계 위에 부모를 지칭합니다. ^^ 두개는 두 단계 위에 부모를 지칭합니다.

git checkout HEAD^

HEAD가 가리키는 커밋의 한 단계 위에 있는 부모 커밋을 체크아웃 합니다.

$ git add [file]

파일을 스테이지에 등록합니다. 스테이지 상태의 파일은 아직 기록된 상태가 아닙니다. 파일의 기록을 위해서는 커밋 수행이 필요합니다.

$ git add .

변경된 모든 파일을 스테이지에 등록합니다. 점(.)으로 시작하는 파일도 포함됩니다. 점(.) 대신 `*` 기호를 사용하면 점(.)으로 시작하는 파일은 포함하지 않습니다.

- `git add --dry-run` 실제로 인덱스에 등록되지 않습니다. 변화되는 정보를 표시만 합니다.
- `git add --all` 신규, 수정, 삭제 파일들이 대상
- `git add .` 신규, 수정 파일들이 대상
- `git add --update` 수정, 삭제 파일들이 대상

$ git add . -f

ignore 파일이나 삭제한 파일 이력까지 커밋을 원하는 경우 -f 옵션을 이용합니다.

$ git commit -m "커밋 메시지"

버전 히스토리에 파일 스냅샷을 영원히 기록합니다. 이것을 커밋한다고 표현합니다. -m 옵션을 붙이지 않으면 VI 에디터가 작동합니다. -a 옵션을 사용하면 변경된 모든 파일을 대상으로 바로 커밋을 수행하여 커밋전에 스테이지 등록작업을 생략할 수 있습니다.

$ git commit -m "커밋 메시지" --amend

--amend 옵션을 주면 최종 커밋을 대체하는 새로운 커밋을 만들어 덮어씁니다. 아무런 변경 내역이 없다면 커밋 메시지만 변경하는 효과가 적용됩니다. 원격 저장소의 푸시한 커밋을 --amend 옵션으로 로컬 저장소에서 변경하지 마십시오. 원격 저장소에 있고 로컬 저장소에는 없는 커밋이 존재하게 되면 다음 push 작업 시 원격 저장소가 거부하기 때문입니다. 이 경우, 먼저 pull 하고 push 해야하는 번거로움이 발생합니다.

$ git revert [커밋해쉬값]

각자의 컴퓨터에서 작업하는 로컬 브랜치의 경우 리셋(reset)을 잘 쓸 수 있습니다만, "히스토리를 고쳐쓴다"는 점 때문에 다른 사람과 같이 작업하는 리모트 브랜치에는 쓸 수 없습니다. 변경분을 되돌리고, 이 되돌린 내용을 다른 사람들과 공유하기 위해서는, git revert를 써야합니다.

커밋을 되돌릴 때, 지운내역을 커밋으로 남깁니다. 이미 다른 개발자들과 공유된 커밋(원격 저장소에 푸쉬된 커밋) 내역을 수정하는 것은 위험합니다. 대신, 커밋으로 발생한 변경 내역의 반대 커밋을 수행하여 변경 내역을 되돌리는 것이 안전합니다. 즉, 추가한 코드를 빼거나 지운 코드를 다시 추가하는 새로운 커밋을 수행하는 것을 의미합니다.

$ git revert HEAD

HEAD가 master 브랜치를 가리키고 있는 상태에서 위 명령을 수행하면 master가 가리키는 커밋을 이전 커밋으로 되돌리는 커밋을 수행합니다. master는 새 커밋을 가리키도록 변경됩니다.

브랜치 관리

브랜치는 특정 커밋에 대한 참조(Reference)에 지나지 않습니다. 따라서 브랜치를 많이 만들어도 메모리나 디스크 공간에 부담이 되지 않습니다. 작업을 브랜치 하나에서 모두 작업하여 복잡하게 만들기 보다는 여러 브랜치로 나누어서 작업하는 것이 관리관점에서 더 좋습니다.

$ git branch

현재 저장소 안의 모든 로컬 브랜치 리스트를 보여줍니다. * 기호는 현재 작업 중인 브랜치를 가리킵니다. -a 옵션을 사용하면 로컬 저장소에 있는 리모트 브랜치까지 조회할 수 있습니다.

$ git branch [branch-name]

새로운 브랜치를 생성합니다. master 브랜치에서 위 명령을 수행했다면 새 브랜치 포인터가 생기면서 master 브랜치 포인터가 가리키는 커밋을 같이 가리키도록 설정됩니다. 새 브랜치가 만들어졌지만 여전히 HEAD는 작업 중인 master 브랜치를 가리키는 상태임을 기억하세요.

$ git checkout [branch-name]

특정 브랜치로 전환하고 워킹 디렉토리를 업데이트합니다. 보다 자세히 설명하자면 HEAD 포인터가 가리키는 대상을 지정한 브랜치의 포인터로 변경합니다. checkout 명령은 언제나 --hard 방식입니다. 따라서, 커밋의 내용이 인덱스로 복사되고 인덱스의 내용이 워킹 디렉토리로 복사됩니다.

$ git checkout --force [branch-name]

브랜치를 전환 할 때 인덱스 또는 워킹 디렉토리가 HEAD와 다른 경우에도 진행하도록 강제합니다. 이 명령은 변경 사항을 버린다는 것을 기억하셔야 합니다.

`$ git merge [branch-name]`

HEAD가 가리키는 현재 작업중인 브랜치에 [branch-name]이 가리키는 커밋을 결합시킵니다. 깃의 머지는 두 개의 부모를 가리키는 특별한 커밋을 만들어 냅니다. 두개의 부모가 있는 커밋이라는 것은 "한 부모의 모든 작업내역과 나머지 부모의 모든 작업, 그리고 그 두 부모의 모든 부모들의 작업내역을 포함한다"라는 의미가 있습니다.

지정한 브랜치의 루트 커밋이 현재 브랜치의 커밋과 같은 경우 작업 중인 브랜치의 포인터가 Fast-Forward 되는 방식으로 결합됩니다. 이때 머지 커밋은 생기지 않습니다. 그렇지 않으면 Three-Way 방식의 머지 작업이 이루어 지는데 이 때는 자동으로 머지 커밋이 추가됩니다.

`$ git branch -d [branch-name]`

특정 브랜치를 삭제합니다

`$ git branch -m [old-branch] [new-branch]`

브랜치명을 변경합니다.

히스토리(커밋의 기록) 리뷰

프로젝트의 진행상황을 살펴보고 점검합니다

`$ git log`

현재 브랜치의 버전 히스토리를 보여줍니다. 옵션은 다음 사이트를 참고하세요.

`log options`

Git의-기초-커밋-히스토리-조회하기

`$ git log --follow [file]`

파일의 버전 히스토리를 보여줍니다.

`git log --branches --graph --decorate --oneline`

모든 브랜치의 히스토리를 그래픽컬하고 간소하게 한줄로 보여줍니다.

`$ gitk`

히스토리를 GUI 화면으로 보여줍니다.

`$ git diff`

워킹 디렉토리와 스테이지 영역인 Index의 차이를 비교합니다. 주로 충돌(Conflict) 발생 시 충돌지점을 찾기위해서 사용합니다. 충돌을 해결한 후 작업 대상을 add 하여 충돌의 해결을 마킹해야 합니다.

`$ git diff --cached`

스테이지 영역과 커밋의 차이를 비교합니다. `--cached` 옵션 대신 `--staged` 옵션을 사용할 수 있습니다.

`$ git diff HEAD`

워킹 디렉토리와 HEAD가 가리키는 커밋의 차이를 비교합니다.

`$ git diff [first-branch] [second-branch]`

두 브랜치 간의 컨텐츠 차이점을 보여줍니다.

`$ git diff [branch] [origin/branch]`

로컬 브랜치와 원격 브랜치간의 차이점을 보여줍니다.

`$ git diff [커밋해쉬값] [커밋해쉬값]`

두 커밋끼리 비교하여 차이점을 보여줍니다.

`$ git show [커밋해쉬값]`

특정 커밋의 컨텐츠 변경 사항을 출력합니다.

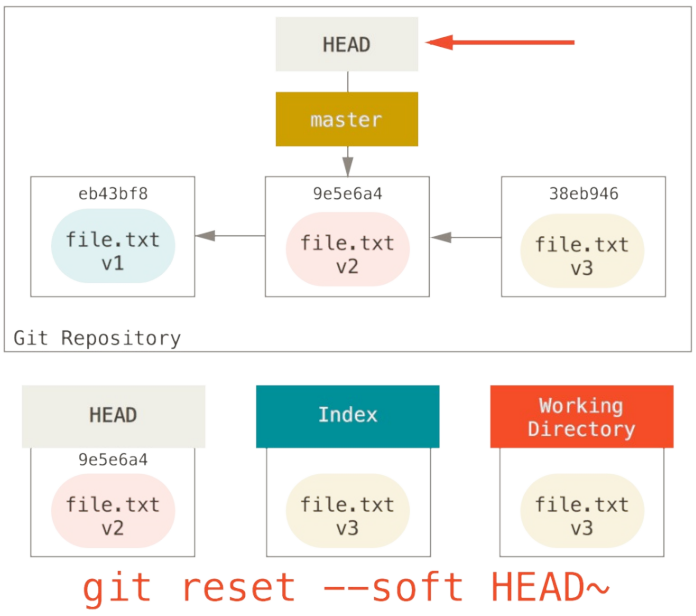
---

reset

reset은 과거 상태로 되돌아 가는 것입니다. push 작업 후에는 다른 사람의 코드에 문제를 일으킬 소지가 있으므로 사용하지 말고 대신 revert 명령을 사용하는 것이 좋습니다.

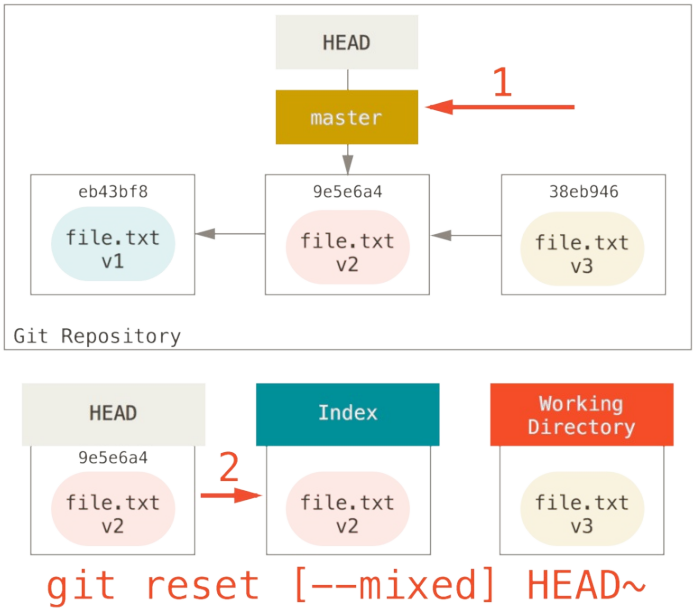
$ git reset --soft HEAD~

HEAD가 가리키는 커밋의 부모의 커밋으로 되돌립니다. 인덱스와 워킹 디렉토리의 내용은 유지됩니다.



$ git reset (--mixed) [commit]

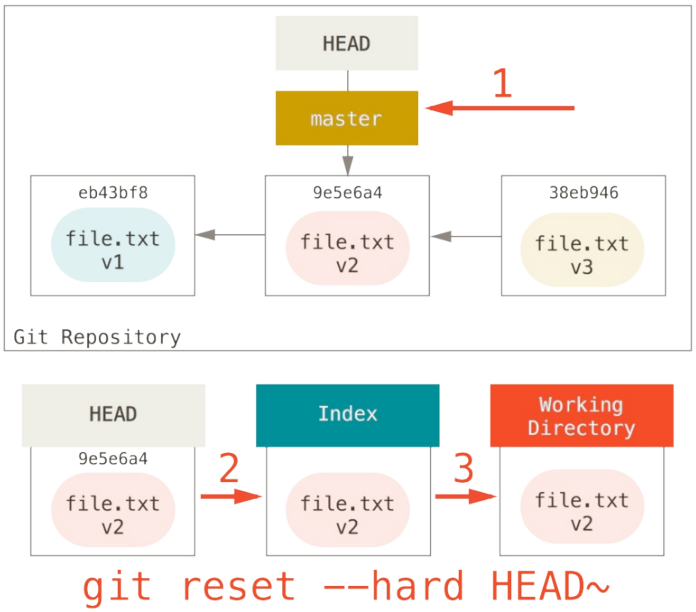
지정한 커밋 이 후의 모든 커밋을 되돌립니다. 로컬에 변경사항은 보존합니다. 옵션을 생략하면 `--mixed` 가 적용됩니다.



$ git reset --hard [commit]

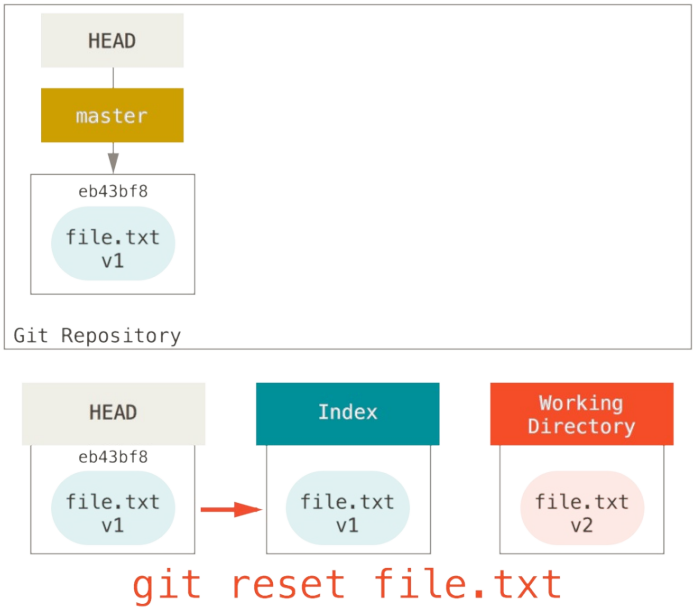
지정한 커밋 이 후의 모든 커밋을 되돌립니다. `--hard` 옵션은 매우 중요합니다. reset 명령을 위험하게 만드는 유일한 옵션입니다. 깃에서 데이터를 실제로 삭제하는 방법이 별로 없는데 이 방법은 그 중 하나입니다. `--hard` 옵션은 되돌리는 것이 불가능합니다. 이 옵션을 사용하면 워킹 디렉토리의 파일까지 강제로 덮어쓰게 됩니다.

그림의 예제는 파일의 v3버전을 아직 깃이 커밋으로 보관하고 있기 때문에 `git reflog` 를 이용해서 다시 복원할 수 있지만 만약 커밋한 적 없다면 깃이 덮어쓴 데이터는 복원할 수 없다는 점을 명심해야 합니다.



$ git reset [file]

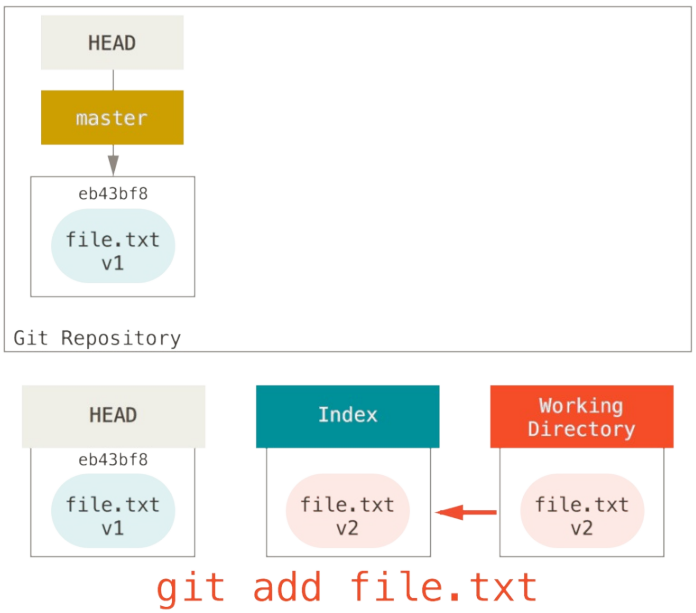
git reset은 브랜치로 하여금 예전의 커밋을 가리키도록 이동시키는 방법으로 변경 내용을 되돌립니다. 이런 관점에서 "히스토리를 고쳐쓴다"라고 말할 수 있습니다. 즉, git reset은 마치 애초에 커밋하지 않은 것처럼 예전 커밋으로 옮기는 것입니다.



파일을 언스테이지(Unstage) 상태로 변경시키나 파일 컨텐츠는 그대로 보존합니다. `git reset --mixed HEAD -- [file]` 명령의 줄임입니다.

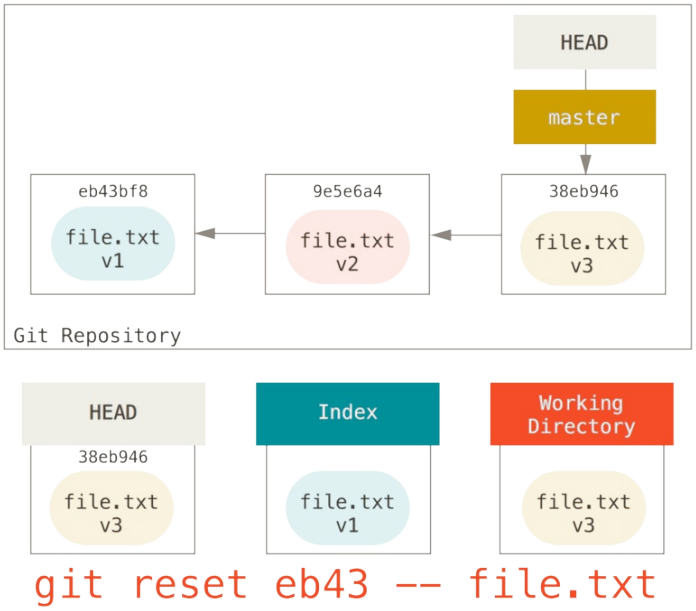
`HEAD는 포인터이고 포인터가 가리키는 것은 압축된 BLOB 파일이므로 커밋에서 일부 파일들을 부분적으로 되돌리는 건 불가능합니다.` 하지만, Index나 워킹 디렉토리는 일부분만 갱신할 수 있습니다. 그래서 파일을 지정한 경우 변경 대상은 커밋은 건너뛰고 옵션에 따라서 Index 영역 및 워킹 디렉토리 영역만이 작업대상이 됩니다.

`gid add [file]` 명령으로 인덱스 영역을 워킹 디렉토리와 같게 만들었다면 `git reset [file]` 명령은 정 반대의 작업을 수행하는 것이 됩니다. 이것이 `git status` 명령의 수행결과에서 `git reset [file]` 명령을 보여주는 이유다. 이 명령으로 파일을 Unstaged 상태로 만들 수 있다.




```
$ git reset eb43bf -- file.txt
```

특정 커밋을 명시하면 깃은 `HEAD에서 파일을 가져오는 것` 이 아니라 지정한 커밋에서 파일을 가져온다.



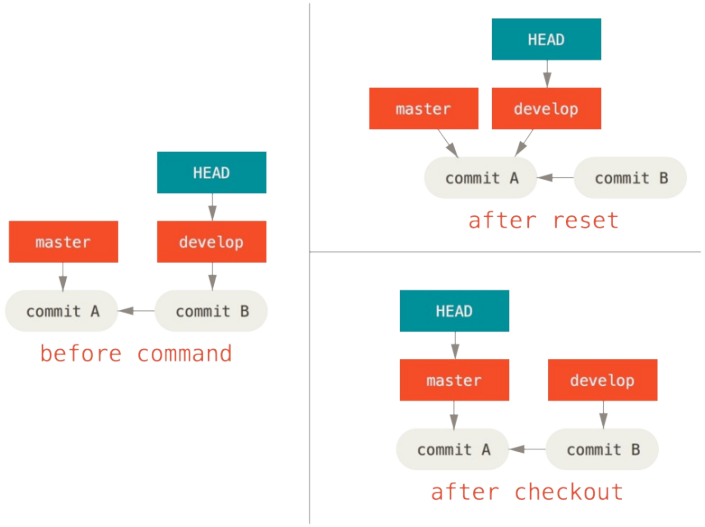
---

reset vs checkout

아마도 checkout 명령과 reset 명령에 어떤 차이가 있는지 궁금할 것이다. reset 명령과 마찬가지로 checkout 명령도 위의 세 트리를 조작한다.

브랜치를 checkout 하면, HEAD가 새로운 브랜치를 가리키도록 바뀌고, 새로운 커밋의 스냅샷을 Index에 놓는다. 그리고 Index의 내용을 워킹 디렉토리로 복사한다.

reset 명령은 checkout 명령처럼 HEAD가 가리키는 브랜치 포인터를 바꾸지는 않는다. HEAD는 계속 현재 브랜치 포인터를 가리키고 있고, 현재 브랜치 포인터가 가리키는 커밋을 바꾼다.



checkout과 reset 명령은 되돌린다는 점에서 비슷해 보이지만 결정적인 차이점이 있다.

- `git checkout 체크섬값` 해당 브랜치 이후에 변경 내역이 사라진다.
- `git reset --soft 체크섬값 / git reset --mixed 체크섬값` 해당 브랜치 이후에 변경 내역이 유지된다.

checkout 파헤치기

checkout 명령이 어떻게 수행되는지 파악하면 reset 명령과의 차이점을 알 수 있을 것이다.
checkout 명령도 reset 처럼 파일 경로를 쓰느냐 안 쓰느냐에 따라 동작이 다르다.

파일 경로 없음

`git checkout [branch]` 명령은 `git reset --hard [branch]` 명령과 비슷하게 [branch] 가 가리키는 커밋의 스냅샷을 기준으로 세 트리를 조작한다. 하지만, 두 가지 사항이 다르다.

1. 첫 번째로 reset --hard 명령과는 달리 checkout 명령은 워킹 디렉토리를 안전하게 다룬다. 저장하지 않은 것이 있는지 확인해서 날려버리지 않는다는 것을 보장한다. 사실 보기보다 좀 더 똑똑하게 동작한다. 워킹 디렉토리에서 합병 작업을 한번 시도해보고 변경하지 않은 파일만 업데이트한다. 반면 reset --hard 명령은 확인도 하지 않고 단순히 모든 것을 바꿔버린다.
2. 두 번째 중요한 차이점은 어떻게 checkout 명령이 HEAD를 업데이트 하는가이다. reset 명령은 HEAD가 가리키는 브랜치 포인터를 움직이지만(브랜치 Refs를 업데이트하지만), checkout 명령은 HEAD 포인터 자체를 다른 브랜치로 옮긴다.

예를 들어 각각 다른 커밋을 가리키는 master 와 develop 브랜치가 있고 현재 워킹 디렉토리는 develop 브랜치라고 가정해보자(즉 HEAD는 develop 브랜치를 가리킨다). `git reset master` 명령을 실행하면 develop 브랜치는 master 브랜치가 가리키는 커밋과 같은 커밋을 가리키게 된다. 반면 `git checkout master` 명령을 실행하면 develop 브랜치가 가리키는 커밋은 바뀌지 않고 HEAD가 master 브랜치를 가리키도록 업데이트된다. 이제 HEAD는 master 브랜치를 가리키게 된다.

그래서 위 두 경우 모두 HEAD는 결과적으로 A 커밋을 가리키게 되지만 방식은 완전히 다르다. reset 명령은 HEAD가 가리키는 브랜치의 포인터를 옮겼고 checkout 명령은 HEAD 포인터 자체를 옮겼다.

파일 경로 있음

checkout 명령을 실행할 때 파일 경로를 줄 수도 있다. reset 명령과 비슷하게 HEAD는 움직이지 않는다. 동작은 `git reset [branch] file` 명령과 비슷하다. Index의 내용이 해당 커밋 버전으로 변경될 뿐만 아니라 워킹 디렉토리의 파일도 해당 커밋 버전으로 변경된다. 완전히 `git reset --hard [branch] file` 명령의 동작이랑 같다. 워킹 디렉토리가 안전하지도 않고 HEAD도 움직이지 않는다.

변경사항 연동하기

원격(URL)을 등록하고 저장소 기록을 주고 받습니다.

`$ git fetch [remote]`

원격 저장소로부터 프로젝트와 프로젝트 히스토리를 다운로드합니다. master 브랜치는 변화가 없습니다. `origin/master` 브랜치는 이제 원격 저장소와 동일한 상태로 변화되었습니다. 개발자가 직접 로컬 저장소의 커밋과 가져온 원격 저장소의 커밋을 병합해야 합니다. 충돌이 발생한다면 개발자가 직접 충돌을 해결하고 커밋해야 합니다. 주로 충돌이 예상될 때 사용합니다.

- 원격 저장소에는 있지만 로컬에는 없는 커밋들을 다운로드 받습니다.
- 로컬의 `origin/master` 가 가리키는곳을 원격 저장소에 `master` 브랜치가 가리키는 곳과 일치하도록 업데이트합니다.

`git fetch` 는 본질적으로 로컬에서 나타내는 원격 저장소의 상태를 실제 원격 저장소의 상태와 동기화합니다. `git fetch` 는 그러나, 여러분의 로컬 브랜치의 상태는 전혀 바꾸지 않는습니다. 여러분의 `master` 브랜치도 업데이트하지 않고 파일 시스템의 모습이던 그 어떤것도 바꾸지 않습니다.

이것을 이해하는게 아주 중요한데, 왜냐하면 수 많은 개발자들이 `git fetch` 를 하면 자신의 로컬 작업이 변경되어 원격 저장소의 모습을 반영해 업데이트 될것이라고 생각하기 때문입니다. 앞의 과정에 필요한 데이터를 다운로드는 하지만, 실제로 로컬 파일들이나 브랜치를 변경하지는 않습니다. 간단하게 `git fetch` 를 다운로드 단계로 생각할 수 있습니다.

이제 우리의 작업을 업데이트해서 변경들을 반영하는 방법은 여러가지 있습니다. 새 커밋들을 로컬에 내려받은 이후에는 그냥 다른 브랜치에있는 일반 커밋처럼 활용할 수 있습니다. 이런 명령들을 실행할 수 있다는 뜻 입니다.

- `git cherry-pick origin/master`
- `git rebase origin/master`
- `git merge origin/master`
- 기타 등등

사실 원격 저장소의 변경을 fetch하고 그이후에 merge하는 작업의 과정이 워낙 자주있는 일이라서 git은 이 두가지를 한번에 하는 명령을 제공합니다. 이 명령어는 `git pull` 입니다.

`$ git merge [remote] [branch]`

원격 브랜치를 현재의 로컬 브랜치와 결합합니다.

`$ git push [remote] [branch]`

원격 저장소에 로컬 브랜치의 커밋을 업로드합니다.

공유된 작업을 내려받는것의 반대는 작업을 업로드해 공유하는것입니다. 그렇다면 `git pull` 당기기의 반대는 `git push` 미는 겁니다.

`git push` 는 여러분의변경을 정한 원격저장소에 업로드하고 그 원격 저장소가 여러분의 새 커밋들을 합치고 갱신하게 합니다. `git push` 가 끝나고 나면, 여러분의 친구들은 원격저장소에서 여러분의 작업을 내려받을 수 있게됩니다.

여러분은 `git push` 를 작업을 "공개"하는 과정이라고 생각해도 될것입니다.

`git push` 를 매개변수 없이 사용하는 디폴트 행동은 `push.default` 라 불리는 git의 설정에 따라 결정 됩니다. 이 설정의 기본값은 여러분이 사용하는 git 버전에 따라 다릅니다만, 보통 origin 값으로 사용합니다.

로컬 `origin/master` 는 `git pull(또는 git fetch), git push` 명령을 사용하는 순간 원격 저장소와 같은 상태가 됩니다. 그 후 여러분의 친구들이 원격저장소를 상대로 작업을 수행하면 로컬 `origin/master` 는 원격 저장소와 다른 상태가 됩니다.

$ git push [remote] --all

원격 저장소에 모든 로컬 브랜치 커밋을 업로드합니다.

$ git push [remote] --tags

하나의 로컬 브랜치 또는 `--all` 옵션을 사용하는 경우 태그는 자동으로 업로드 되지 않습니다. `--tags` 옵션을 추가하여 태그 정보도 업로드 되도록 조치합니다.

$ git remote add origin [url]

url이 가리키는 remote 저장소를 origin이라는 이름으로 추가합니다. 다음부터 간단히 origin이라는 별칭으로 remote 저장소를 지칭할 수 있습니다.

$ git remote -v

등록된 remote 저장소 리스트를 조회합니다.

$ git push -u origin master

origin이 가리키는 원격 저장소에 로컬 master 브랜치에 내용을 업로드합니다. -u 옵션은 upstream(상류 저장소)을 의미합니다. 이 옵션을 사용하면 다음 부터는 간단히 git push 명령으로 업로드하고 git pull 명령으로 다운로드해서 로컬 저장소에 병합할 수 있습니다. 상류 저장소는 최초 저장소를 의미하며 다른 개발자가 fork 할 때 fork 한 저장소와 기원이 되는 저장소를 구분하기 위한 개념입니다.

$ git pull

원격 저장소의 프로젝트와 프로젝트 히스토리를 다운로드 합니다. 다운로드 후 자동으로 로컬 브랜치에 병합하는 작업이 이루어 집니다. git pull 작업은 `git fetch + git merge` 작업과 같습니다. 주로 충돌이 예상되지 않을 때 사용합니다.

```
로컬 저장소
C0      <--       C1      <--     C2
                  origin/master   master

원격 저장소
C0      <--       C1      <--     C3
                                  master

원격 저장소에 로컬 저장소에는 존재하지 않는 C3 커밋이 존재하므로
로컬 저장소보다 빠른 상태입니다.

git fetch && git merge origin/master

로컬 저장소
C0      <--       C1      <--     C2      <--     C4
                          <--     C3              master
                                  origin/master

원격 저장소
C0      <--       C1      <--     C3
                                  master

git fetch 명령으로 원격 저장소의 C3 커밋이 다운로드 됩니다.
로컬 저장소의 origin/master가 C3를 가리키도록 변경됩니다.
HEAD가 master를 가리키고 있다면 git merge origin/master
명령으로 master가 가리키는 C2에 origin/master가 가리키는
C3를 결합시킵니다. 결합결과로 C4 커밋이 만들어 집니다.
master가 C4를 가리키도록 변경됩니다.
```

원격 저장소의 C3를 fetch로 내려 받고 git merge origin/master로 병합했습니다. 로컬 master 브랜치는 원격 저장소의 새 작업들을 반영하게 됩니다.

`git fetch origin; git diff --name-only master origin/master`

origin이 가리키는 원격 저장소로부터 다운로드 합니다. 로컬 master 브랜치와 origin/master 브랜치를 비교하여 어떤 파일들이 갱신되는지 조회합니다.

커밋에 태그 붙이기

프로젝트의 히스토리(작업 이력)에서 중요한 지점들에 영구적으로 표시를 할 방법이 존재합니다. 태그는 주요 릴리즈나 큰 브랜치 병합(merge)이 있을때 이를 기록하기 위해 존재합니다. 태그는 특정 커밋들을 브랜치로 참조하듯이 영구적인 "milestone(이정표)"으로 표시합니다.

중요한 점은, 태그는 커밋들이 추가적으로 생성되어도 절대 움직이지 않는다는 것입니다. 그래서 여러분은 태그를 "체크아웃"한 후에 그 태그에서 어떤 작업을 완료할 수 없습니다. 태그에 직접 커밋을 할 수 없다는 얘기입니다. 태그는 커밋 트리에서 특정 지점을 표시하기 위한 닻같은 역할을 수행합니다.

`$ git tag 1.0`

HEAD가 가리키는 지점에 light weight 태그를 추가합니다.

`$ git log -2`

가장 최근 두개의 커밋만을 조회합니다.

`$ git tag 2.0` 커밋해쉬값

특정 커밋에 태그를 추가합니다. 만약 커밋해쉬값을 지정해주지 않으면 깃은 HEAD가 가리키는 지점에 태그를 붙일 것입니다.

`$ git tag -a 3.0` 커밋해쉬값

-a 옵션을 사용하여 annotated 태그를 추가하면 태그 정보를 편집할 수 있는 vi 에디터가 기동합니다. 보다 상세한 설명을 추가할 수 있습니다. -m 옵션 다음 메시지를 바로 작성하면 vi 에디터는 기동하지 않습니다.

`$ git show-ref --tags`

관련 커밋 ID와 함께 로컬 저장소에서 사용 가능한 참조를 표시합니다.

`$ git show 3.0`

3.0 태그가 붙은 정보를 조회합니다.

`$ git tag -d 3.0`

3.0 태그를 삭제합니다.

`$ git checkout` 태그명

태그명으로 체크아웃 할 수 있습니다. 단, 태그에 직접 커밋을 할 수 없다는 것을 기억하세요.

show

git show 명령은 Git 개체를 사람이 읽을 수 있도록 요약해서 보여줍니다. 태그나 커밋 정보를 보고 싶을 때 이 명령을 사용합니다.

`$ git show`

현재 브랜치의 가장 최근 커밋 정보를 확인합니다.

`$ git show` 커밋해쉬값

특정 커밋 정보를 확인합니다.

`$ git show HEAD`

HEAD 포인터가 가리키는 커밋정보를 확인합니다.

`$ git show` 커밋해쉬값 또는 `HEAD^`

^표시 한 개면 한 개 전 커밋을 대상으로 조회합니다. 예를 들어 `master^` 는 "master의 부모"와 같은 의미가 되고 `master^^` 는 "master의 조부모(부모의 부모)"를 의미합니다

`$ git show` 커밋해쉬값 또는 `HEAD~`숫자

~숫자는 명시적으로 몇 개 전인지 표시하여 사용합니다. ~ 표시와 ^ 표시는 혼합 사용이 가능합니다.

기타

`$ git reflog`

Git은 자동으로 브랜치와 HEAD가 지난 몇 달 동안에 가리켰었던 커밋을 모두 기록하는데 이 로그를 Reflog라고 부릅니다. HEAD의 변화를 확인하시고 싶을 때 git reflog 명령을 사용합니다.

`$ git rebase master`

merge 명령말고 브랜치끼리의 작업을 접목하는 방법은 리베이스(Rebase)입니다. 리베이스는 기본적으로 커밋들을 모아서 복사한 뒤, 다른 곳에 떨궈 놓는 것입니다.

다른 브랜치에서 커밋을 한 상태에서 master 브랜치에서 추 후 커밋을 한 경우, 다른 브랜치에서 위 명령으로 다른 브랜치의 루트 커밋을 master 브랜치가 가리키는 커밋으로 이동시킵니다. 그런 다음 master를 Fast-Foward하여 다른 브랜치를 가리키게 하면 한 줄기의 히스토리를 얻을 수 있습니다. 리베이스를 쓰면 저장소의 커밋 로그와 이력이 한결 깨끗해집니다.

`$ git rebase bugFix`

master 포인터가 bugFix 포인터가 가리키는 커밋보다 지난 커밋을 가리키고 있다면 위 명령으로 master 브랜치 포인터를 앞으로 Fast-Foward 할 수 있습니다.

- bugFix 포인터가 가리키는 커밋을 master 앞에 떨구려고 보니 이미 bugFix 포인터가 가리키는 커밋이 master 포인터가 가리키는 커밋보다 앞에 있으므로 이 작업은 건너 뜁니다.
- master 포인터가 bugFix 포인터가 가리키는 커밋을 가리키도록 변경합니다.

`$ git branch -f master HEAD~3`

위 명령을 사용하면 강제로 master 브랜치 포인터를 HEAD에서 세번 위에 있는 커밋으로 옮길 수 있습니다. -f 옵션으로 특정 브랜치 포인터를 특정 커밋으로 옮길 수 있습니다.

`git blame [file]`

해당 파일의 수정 이력을 볼 수 있습니다.

`git fetch origin; git reset --hard origin/master`

로컬에 있는 모든 변경 내용과 확정본을 포기하기 위해서 원격 저장소의 최신 이력을 가져오고 로컬 master 브랜치가 그 이력을 가리키도록 조치합니다.

---

cherry-pick

히스토리를 하나의 가지로 관리하려고 master브랜치에 병합하기 전 master 브랜치를 기반으로 Rebase 할 수 있습니다. rebase는 master 브랜치를 Fast-Forward시켜 평평한 히스토리를 유지할 수 있는 방법입니다.

한 브랜치에서 다른 브랜치로 작업한 내용을 옮기는 또 다른 방법으로 Cherry-pick이 있습니다. Cherry-pick은 커밋 하나만 Rebase하는 것입니다. 커밋 하나로 Patch 내용을 만들어 master 브랜치에 적용을 하는 것입니다. 단, 그 커밋이 현재 HEAD가 가리키고 있는 커밋이 아니어야합니다. 다른 브랜치에 있는 커밋 중에서 하나만 고르거나 다른 브랜치에 커밋이 하나밖에 없을 때 Rebase보다 유용합니다.

`$ git cherry-pick` 커밋해쉬값`(C2)` 커밋해쉬값`(C4)`

번거로울 수 있는 브랜치 정리 작업(rebase -i)을 피하거나 여러 브랜치의 이력을 그대로 남기고자 한다면 master 브랜치는 새로 파일을 만들거나 수정하지말고 오직 병합작업만 수행하는 것이 좋습니다.

HEAD가 master브랜치를 가리키는 상태라고 가정합니다. 다른 브랜치에 C2, C3, C4 커밋이 존재할 때 C2, C4 커밋만을 복사해서 master 브랜치에 붙여 넣고 싶습니다. 이 때 위 명령을 사용하면 됩니다. 다른 브랜치에 커밋은 그대로 보존됩니다. 커밋해쉬값은 유니크하기 때문에 여러 브랜치에 존재하는 다수의 커밋을 한 번에 작업으로 체리픽 작업을 수행할 수 있습니다.

Squash(대화방식 리베이스)

체리픽은 여러분이 원하는 커밋이 무엇인지 알때(각각의 커밋해시값) 아주 유용합니다. 하지만 원하는 커밋을 모르는 상황에서는 대화형 리베이스를 사용하면됩니다. 리베이스할 일련의 커밋들을 검토할 수 있는 가장 좋은 방법입니다. 스쿼시는 여러 커밋을 합치는 작업을 의미합니다.

인터렉티브 리베이스가 의미하는 뜻은 rebase 명령어를 사용할 때 -i 옵션을 같이 사용한다는 것입니다. 이 옵션을 추가하면, git은 리베이스의 목적지가 되는 곳 아래에 복사될 커밋들을 보여주는 vi를 띄울것 입니다. 각 커밋을 구분할 수 있는 각각의 해시들과 메시지도 보여줍니다.

- pick : 해당 커밋을 남긴다.
- reword : 해당 커밋을 남긴다. 커밋 메시지 변경을 위해서 메시지 수정창이 뜬다.
- edit : rebase 작업이 잠시 중단된다. 커밋을 amend 할 수 있다.
- squash : 해당 커밋을 이용하고 이전 커밋과 합친다. 커밋 메시지도 합쳐지고 메시지 수정창이 뜬다.
- fixup : squash와 같지만 이 커밋 메시지는 버린다.
- exec : 쉘을 이용한 명령을 수행한다.
- drop : 커밋을 삭제한다.

인터렉티브 리베이스 대화창이 열리면, 3가지를 할 수 있습니다:

- 적용할 커밋들의 순서를 UI를 통해 바꿀수 있습니다.
- 원하지 않는 커밋들을 뺄 수 있습니다.
- 커밋을 스쿼시(squash)할 수 있습니다.

```
$ git rebase -i HEAD~4
```

4개의 커밋을 대상으로 스쿼시 작업을 수행합니다.

rebase 시 발생하는 충돌 해결방법

- git rebase --continue 충돌을 해결한 후 rebase 작업을 계속 진행하여 완료합니다.

- git rebase --skip 병합 대상(master) 브랜치의 내용이 그대로 복사됩니다. 다시 git rebase 명령을 실행할 수 없습니다. 권장하지 않는 방법입니다.
- git rebase --abort 명령을 취소합니다.

병합한 흔적을 남기고자 할 때 --no-ff 라는 옵션을 사용합니다.

Remote repository 에 squash 한 commit push 하기

가능하면 이미 push하지 않은 작업만 squash 하는것을 추천합니다. push와 pull 작업이 살짝 번거로워 집니다.

아무도 squash 한 commit 들을 pull 하지 않았다는 가정 하에 이 작업이 수행되어야 한다. 누군가 이미 해당 commit 들을 pull 했는데 그것을 rebase -i 를 통해 수정하고 commit 하면 큰일 날 수 있다.

대표적으로 이러한 squash 작업이 필요한 경우가 Open Source repository 를 fork 해서 수정한 후에 pull request 를 날렸는데 수정요청이 들어올 때이다.

Pull request 가 여러개의 commit 으로 이루어지게 되면 중간에 완성되지 않은 commit 이 존재하는 것이기 때문에 관리상에 어려움이 생길 수 있다. 이를 방지하기 위해 squash 를 통해 제대로 동작하는 하나의 commit 으로 만들어 주는 것이 좋다.

```
git push origin <branch-name> --force
```

또는

```
git push origin +<branch-name>
```

`--force` 는 푸시 된 모든 참조에 적용되므로 `push.default` 가 일치하도록 설정된 경우 또는 `remote.*.push` 로 구성된 여러 푸시 대상과 함께 사용하면 그 참조 이외의 `refs` 를 덮어 쓸 수 있습니다. 현재 지점 (원격 상대방보다 엄격하게 뒤에있는 로컬 참조 포함).

```
하나의 브랜치 만 강제로 푸시하려면 'refspec'앞에 '+'를 사용하여 푸시합니다.
```

```
git fetch && git merge --squash
```

병합 작업 시 가져오는 브랜치에 모든 커밋을 하나의 커밋으로 뭉개서 작업 대상 브랜치에 적용한다.

부모를 선택하는 방법

~ 수식처럼 ^ 수식 또한 뒤에 숫자를 추가 할 수 있습니다.

몇개의 세대를 돌아갈지 정하는 것 대신(~의 기능) ^수식은 병합이된 커밋에서 어떤 부모를 참조할지 선택할 수 있습니다. 병합된 커밋들은 다수의 부모를 가지고 있다는것을 기억하시나요? 어떤 부모를 선택할지 예측할 수가 없습니다.

Git은 보통 병합된 커밋에서 "첫"부모를 따라갑니다. 하지만 ^수식을 를 숫자와 함께 사용하면 앞의 디폴트 동작대로가 아닌 다른 결과가 나타납니다.

`$ git checkout master^`

현재 master가 가리키는 커밋은 병합 커밋이라고 가정합니다. master를 수식없이 체크아웃한다면 병합된 커밋의 첫 부모를 따라 HEAD가 올라갈 것입니다.

`$ git checkout master^2`

두 번째 부모를 따라 HEAD가 올라갈 것입니다.

`$ git checkout HEAD~^2~2`

- ~^2 : 두 번째 부모를 따라 2번 올라갑니다. 부모가 하나일 때 ^ 설정은 아무런 영향을 미치지 않습니다.
- ~2 : 첫 번째 부모를 따라 2번 올라갑니다.

`$ git checkout [커밋해쉬값] && git branch bugWork`

현재 HEAD는 master 브랜치를 가리키고 있다고 가정합니다. 이 상황에서 한참 이전 커밋을 기준으로 새로운 브랜치 bugWork를 생성합니다.

`push` 의 인자들

push를 하면 git이 push를 할 대상인 원격저장소 브랜치를 현재 작업중인 브랜치에 설정된 속성("추적" 대상)을 통해 알아냅니다. 이것은 인자를 넣지않고 실행할 때 일어나는 것 입니다. 그런데 git push에 다음과 같은 형식으로 선택적으로 인자를 사용할 수도 있습니다.

```
git push [remote] [place]
```

[place] 인자가 무엇을 의미할것 같나요? 세부사항은 알아보기 전에 예시부터 봅시다. 다음 명령어를 보세요.

```
git push origin master
```

해석해 보면, 내 저장소에 있는 "master"라는 이름의 브랜치로 가서 모든 커밋들을 수집합니다. 그 다음 "origin"의 "master"브랜치로 가서 이 브랜치에 부족한 커밋들을 채워 넣고 완료 되면 알려줍니다.

master를 "place" 인자로 지정해서 우리가 git에게 어디서부터 커밋이 오는지, 그리고 어디로 커밋이 가야하는지 알려줍니다. 두 저장소간에 동기화 작업을 할 "장소"를 지정해 주는것이라고 볼 수 있습니다.

깃이 알아야 할 것은 다 알려줬기 때문에, 깃은 현재 우리가 체크아웃 한 브랜치는 무시하고 명령을 수행합니다.

```
git checkout C0; git push
```

위 명령은 실패합니다. HEAD가 원격저장소를 추적하는 브랜치에 체크아웃 되어 있지 않은 상태에서 push 명령을 내렸기 때문이죠.

```
git checkout C0; git push origin master
```

위 명령처럼 사용하면 성공합니다.

`git push` 인자에 대한 세부사항

앞서서 우리는 master를 커밋의 근원이되는 source와 목적지가 되는 destination으로 명령어의 인자로 넣어줌으로써 지정해줬습니다. 여러분은 이런 생각이 들 수 있어요. 내가 source와 destination이 다르길 원하면 어떻게 해야되지? 로컬의 foo 브랜치에서 원격의 bar 브랜치로 커밋을 push 하고 싶으면 어떻게 해야 되지?

source와 destination을 모두 지정하기 위해서는 간단히 두개를 콜론 기호를 사이에 두고 표현하면 됩니다.

```
git push origin [source]:[destination]
```

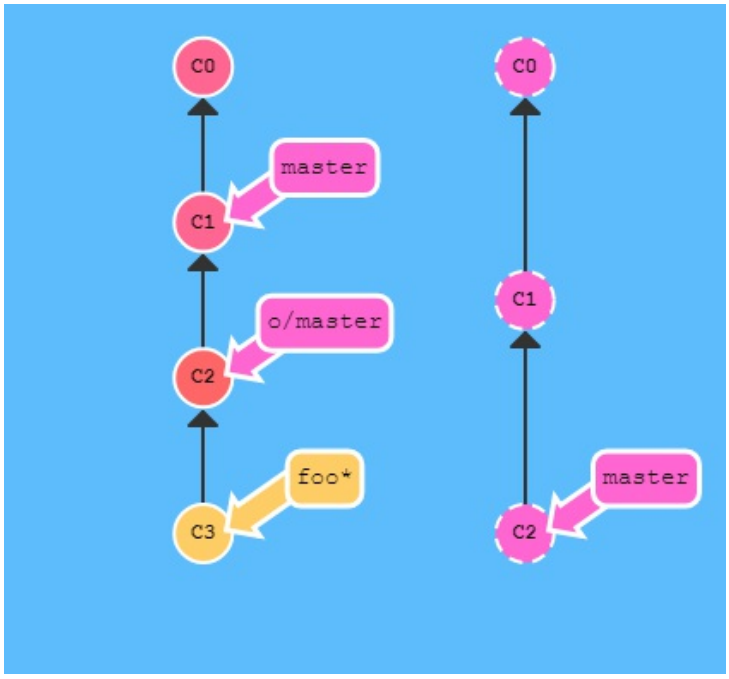
이것을 일반적으로 `colon refspec(콜론 참조스펙)` 이라고 부릅니다. 참조스펙은 그냥 "git이 알아낼 수 있는 위치"를 이름 붙여서 말하는거에요.

예: 브랜치 'foo' 또는 HEAD~1

source와 destination을 따로 지정할 수 있게 되면서, 이제 원격관련 명령어를 좀 멋지고 정확히 사용할수 있게 되었어요.

```
git push origin foo^:master
```

깃은 foo^의 위치를 알아내서 원격 저장소에 아직 반영되지 않은 커밋들을 업로드하고 destination 브랜치인 master를 갱신합니다. 만약 destination 브랜치가 없다면 자동으로 생성됩니다.



`git fetch` 인자에 대한 세부사항

여태까지 우리는 git push 인자들에 대해 배워봤습니다. 이 멋진 [place] 인자 그리고 콜론 참조스펙도 말이죠([source]:[destination]). 우리가 알아낸 이 지식을 git fetch에도 적용 할 수 있으려나요?

당연하죠! git fetch에 넘기는 인자들은 사실 git push의 그것들과 아주 아주 비슷합니다. 같은 컨셉으로 적용되지만 방향이 반대일 뿐이죠. 커밋을 업로드하는게 아니라 다운받는 것이니까요.

git fetch에 다음 명령어와 같이 place를 지정해주면:

```
git fetch origin foo
```

깃은 원격 저장소의 foo 브랜치로 가서 현재 로컬에 없는 커밋들을 가져와 로컬의 'origin/foo' 브랜치 아래에 추가 할 것입니다.

"source와 destination를 모두 직접 지정해주면 어떻게될까요?

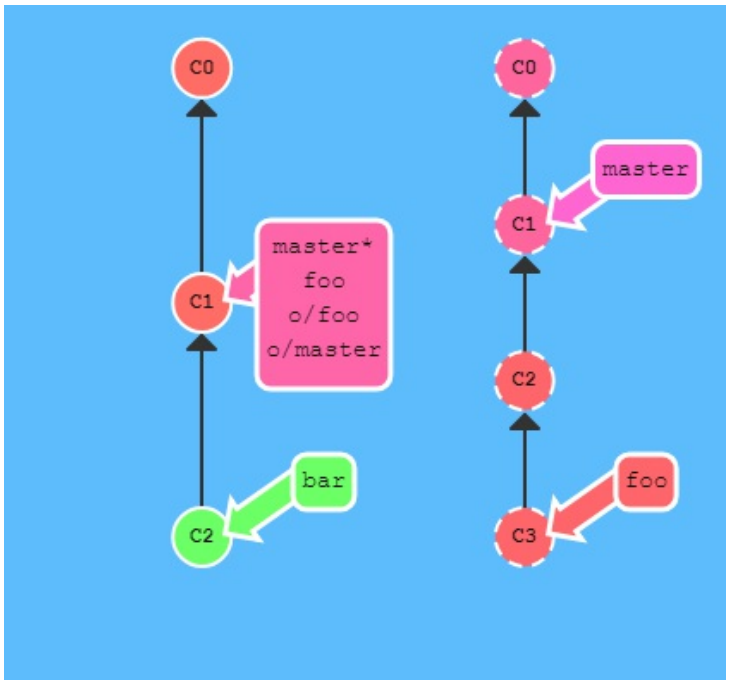
여러분이 커밋을 직접 로컬 브랜치로 fetch할 열의가 있다면 콜론 참조스펙으로 지정해서 할 수 있습니다. `하지만 체크아웃된 브랜치에 fetch할 수는 없고 체크아웃되지 않은 브랜치만 가능합니다.`

주의 할 점이 하나 있는데 [source]는 이제 받아올 커밋이 있는 원격에 있는 place를 넣어줘야하고 [destination]은 그 커밋들을 받아올 local의 place를 인자로 넣어줘야 합니다. git push와 정반대로 하는거죠. 데이터를 반대의 방향으로 옮기는 작업이니 이게 더 납득이 갑니다.

언급한 것 처럼 실제로 이것을 사용하는 개발자들은 많지 않습니다. 이것을 소개하는것은 fetch와 push가 방향이 반대일뿐 컨셉이 비슷하다는 것을 설명하기 위해서 입니다.

```
git fetch origin foo~1:bar
```

깃이 foo~1을 origin의 place로 지정하고 커밋들을 내려받아 bar(로컬 브랜치)에 추가했습니다. foo와 origin/foo는 갱신되지 않습니다. destination을 지정해줬기 때문입니다.



`git fetch` 를 인자없이 수행하면 원격저장소에서 모든 원격 브랜치들의 커밋들을 내려받습니다.

`git fetch` 는 로컬의 원격 브랜치가 아닌 브랜치는 갱신하지 않습니다. 커밋들을 내려받기만 합니다. 여러분이 확인해보고 나중에 병합할 수 있도록 말이죠.

없음을 의미하는 인자

깃은 [source] 인자를 두가지 방법으로 이상하게 사용합니다. 이 두가지 오용은 여러분이 `git push` 와 `git fetch` 에 source에 "없음"을 지정할 수 있기 때문에 나타납니다. "없음"을 지정하는 방법은 인자로 아무것도 안쓰면 됩니다.

```
git push origin :side
```

`"없음"을 원격 브랜치로 push하면 무엇을 할까요? 원격저장소의 그 브랜치를 삭제합니다!` Null을 푸쉬해서 원격 저장소의 브랜치를 삭제한다고 이해하세요. 이 때, 로컬의 origin/side 브랜치도 같이 삭제가 됩니다.

```
git fetch origin :bar
```

`"없음"을 fetch하면 로컬에 새 브랜치를 만듭니다.` 기괴합니다. Null을 가져오는 것이므로 실제로 다운로드 할 것은 없는 상태에서 로컬의 해당 브랜치가 없으니 만들기는 한다고 이해하세요.

`git pull` 인자에 대한 세부사항

`git fetch` 와 `git push` 의 인자들을 다 알아 보았기 때문에 `git pull` 에서 더 설명할게 사실 없습니다.

`git pull` 은 결국 merge가 따라오는 fetch 그 자체이기 때문이죠. `git fetch` 와 같은 인자를 사용하며 커밋들을 어디로 merge되는지만 알면 됩니다.

```
git pull origin foo
```

위 명령은 다음 명령어들과 같습니다.

```
git fetch origin foo; git merge origin/foo
```

그리고,

```
git pull origin bar~1:bugFix
```

위 명령은 다음 명령어들과 같습니다.

```
git fetch origin bar~1:bugFix; git merge bugFix
```

`git pull` 은 그저 `fetch + merge` 의 축양형일 뿐이에요. `git pull` 은 커밋들이 도착하는곳을 대상으로 삼아서 merge가 수행됩니다.

```
git pull origin master
```

위 명령은 다음 명령어들과 같습니다.

```
git fetch origin master; git merge origin/master
```

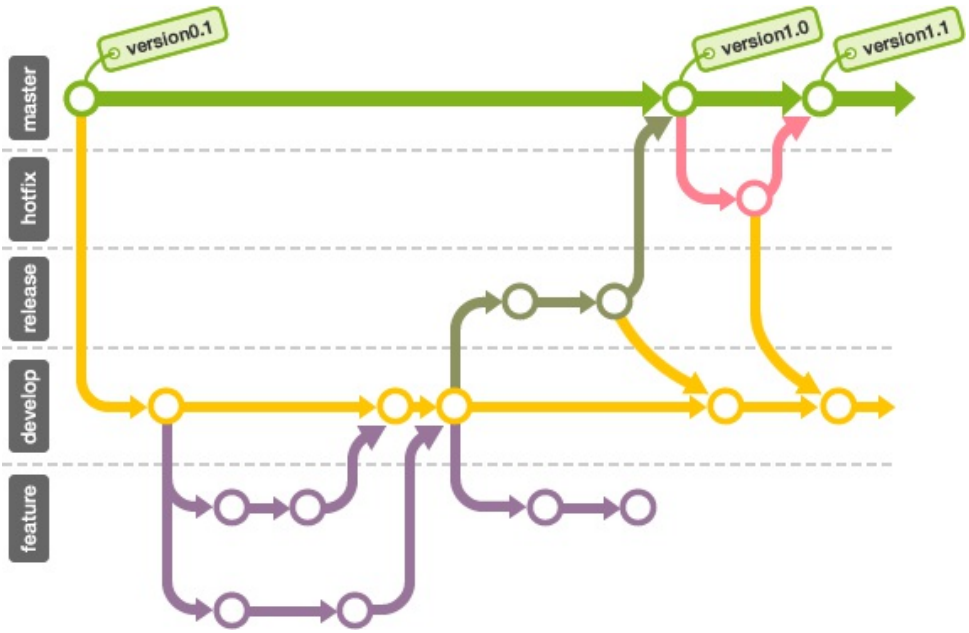
`주의할 점은 HEAD가 master에 있는 상태에서 위 명령을 수행했다면 origin/master를 master에 병합하고 HEAD가 bar에 있는 상태에서 위 명령을 수행했다면 origin/master를 bar에 병합한다는 점 입니다.`

그러므로 git pull 명령 시 HEAD가 어디를 가리키고 있는 상태인지 꼭 확인하셔야 합니다.

6. 브랜칭 전략

Git Flow

깃플로우는 널리 사용되고 있는 브랜칭 모델입니다. 기본 브랜치는 5가지를 사용합니다. `feature` → `develop` → `release` → `hotfix` → `master` 브랜치가 존재합니다. 병합순서는 앞에서 뒤로 진행됩니다. release, hotfix 브랜치모두 develop 브랜치에 병합 하도록 구성이 되어 있습니다. GUI 툴들에서 기본 내장 git-flow 명령이나 플러그인을 설치하여 작업을 진행할 수 있도록 보편화되어 있는 브랜칭 모델입니다.



feature 브랜치

새로운 기능을 추가하는 작업을 위한 브랜치입니다. feature 브랜치는 원격 저장소에 반영하지 않고 개발자의 로컬 저장소에만 존재합니다. `--no-ff` 옵션을 이용하여 develop 브랜치에 병합 기록을 남깁니다.

develop 브랜치

여러 개발자가 추가한 기능들을 모으고 검토하는 브랜치입니다. `--no-ff` 옵션을 이용하여 release 브랜치에 병합 기록을 남깁니다.

release 브랜치

새로운 릴리즈 준비를 위한 브랜치입니다. develop 브랜치로부터 다음번 릴리즈에서 사용할 기능들을 모으고 검토합니다. `--no-ff` 옵션을 이용하여 master 브랜치에 병합 기록을 남깁니다. master 브랜치로 합병한 후 tag 명령을 이용하여 릴리즈 커밋에 대해 버전 정보를 추가합니다.

hotfix 브랜치

master 브랜치에서 발생한 버그들을 처리하기 위한 브랜치입니다. 수정이 끝나면 master, develop 브랜치에 반영하고 master 브랜치 커밋에 tag를 추가합니다.

급하게 고치지 않아도 되는 상황이라면 release, develop 브랜치에 hotfix 브랜치를 병합하여 릴리즈될 때 반영이 될 수 있도록 조치합니다.

master 브랜치

master 브랜치는 항상 최신의 상태이며 stable 상태로 배포되는 브랜치입니다. 팀 단위로 협업한다면 이 브랜치는 엄격하게 관리되어 몇몇의 책임자만 수행하는 것이 좋습니다.

만약 이 브랜치의 커밋을 대상으로 추가적인 논의가 필요하다면 master 브랜치 위에 production 브랜치를 두고 production 브랜치를 배포되는 브랜치로 삼는 것도 하나의 방법입니다.

7. 시나리오 학습

좋은 커밋 메시지 작성하기

좋은 git 커밋 메시지를 작성하기 위한 7가지 약속

커밋을 골라서 적용하기

개발중에 종종 이런 상황이 생깁니다. 잘 띄지 않는 버그를 찾아서 해결하려고, 어떤 부분의 문제인지를 찾기 위해 디버그용 코드와 화면에 정보를 프린트하는 코드를 몇 줄 넣습니다. 디버깅용 코드나 프린트 명령은 bugFix 브랜치에 들어있습니다. 마침내 버그를 찾아서 고쳤고, 원래 작업하는 브랜치에 합치면 됩니다.

이제 bugFix 브랜치의 내용을 master에 합쳐 넣으려 하지만 한 가지 문제가 있습니다. 그냥 간단히 master 브랜치를 `최신 커밋으로 이동(Fast-Forward)` 시킨다면 그 불필요한 디버그용 코드들도 함께 들어가 버린다는 점이 문제입니다.

여기에서 Git의 마법이 드러납니다. 이 문제를 해결하는 여러가지 방법이 있습니다만, 가장 간단한 두가지 방법은 아래와 같습니다:

```
$ git rebase -i
```

대화형(-i 옵션) 리베이스(rebase)로는 어떤 커밋을 취하거나 버릴지를 선택할 수 있습니다. 또 커밋의 순서를 바꿀 수도 있습니다. 이 커맨드는 어떤 작업의 일부만을 골라내는 작업에 유용합니다.

```
$ git cherry-pick
```

체리픽은 개별 커밋을 골라서 HEAD위에 떨어뜨릴 수 있습니다. 예를 들어 debug 커밋과 print 커밋은 선택하지 말고 bugFix 커밋만을 체리픽해서 master 브랜치에 떨구면 됩니다.

한참 전 커밋의 내용 바꾸기

이번에도 꽤 자주 발생하는 상황입니다. newImage와 caption 브랜치에 각각의 변경내역이 있고 서로 약간 관련이 있어서, 저장소에 차례로 쌓여있는 상황입니다.

때로는 한참 전 커밋의 내용을 살짝 바꿔야하는 골치아픈 상황에 빠지게 됩니다. 이 문제를 다음과 같이 풀어봅시다.

```
$ git rebase -i HEAD~4
```

명령으로 우리가 바꿀 커밋을 가장 최근 순서로 바꾸어 놓습니다.

```
$ git commit -m "changed" --amend
```

명령으로 커밋의 내용을 정정합니다. 정정할 커밋이 바로 직전에 있으면 --amend 옵션으로 간단히 수정할 수 있습니다.

```
$ git rebase -i HEAD~4
```

명령으로 이 전의 커밋 순서대로 되돌려 놓습니다.

```
$ git branch -f master HEAD
```

명령으로 master를 지금 트리가 변경된 부분으로 이동합니다.

충돌이 예상될 때는 체리픽으로 작업하는 것이 훨씬 좋습니다. 위 작업을 다시 수행하되 rebase -i 를 쓰지 말고 체리픽으로 할 수 있는 방법을 살펴봅니다.

```
$ git checkout master
```

```
$ git cherry-pick C2
```

```
$ git commit -m "changed" --amend
```

```
$ git cherry-pick C3
```

이정표(태그) 확인

커밋 트리에서 태그가 훌륭한 "닻"역할을 하기 때문에, git에는 여러분이 가장 가까운 태그에 비해 상대적으로 어디에 위치해있는지 설명해주는 명령어가 있습니다. 이 명령어는 `git describe` 입니다.

`git describe` 명령은 커밋 히스토리에서 앞 뒤로 여러 커밋을 이동하고 나서 커밋 트리에서 방향 감각을 다시 찾는데 도움을 줍니다. 이런 상황은 `git bisect` (문제가 되는 커밋을 찾는 명령어라고 간단히 생각하자)를 하고 나서라던가 휴가를 다녀온 동료의 컴퓨터에 앉는 경우가 있습니다.

`git describe` 는 다음의 형태를 가지고 있습니다:

`git describe <ref>`

`<ref>` 에는 commit을 의미하는 그 어떤것(HEAD, 브랜치, 커밋해쉬값)이던 쓸 수 있습니다. 만약 ref를 특정 지어주지 않으면 git은 그냥 지금 `체크아웃된 곳(HEAD)` 을 사용합니다.

명령어의 출력은 다음과 같은 형태로 나타납니다.

`<tag>_<numCommits>_g<hash>`

tag는 가장 가까운 부모 태그를 나타냅니다. numCommits은 그 태그가 몇 커밋 멀리있는지를 나타냅니다. `<hash>` 는 묘사하고있는 커밋의 해시를 나타냅니다.

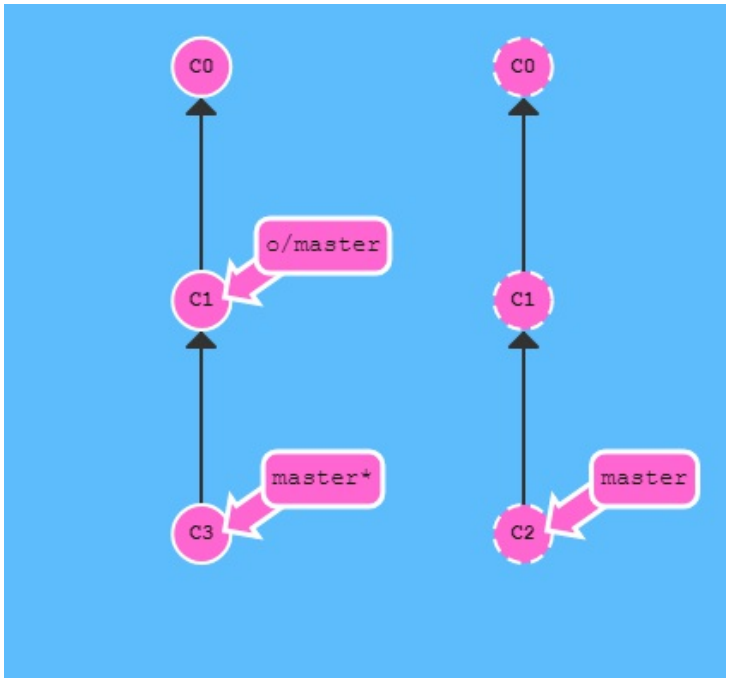
---

origin/master가 원격 저장소보다 뒤처져 있을 때에 **push** 작업

여러분은 월요일에 저장소를 clone해서 부가기능을 만들기 시작했습니다. 금요일쯤 기능을 공개할 준비가 되었습니다. 그런데 동료들이 주중에 코딩을 잔뜩해서 여러분이 만든 기능은 프로젝트에 뒤떨어져서 무용지물이 되었습니다. 이 사람들이 그 커밋들을 공유하고있는 원격 저장소에도 공개했습니다, 이제 여러분의 작업은 이제 의미가 없는 구버전의 프로젝트를 기반으로한 작업이 되어버렸습니다.

이런 경우, 명령어 git push가 할 일이 애매해집니다. git push를 수행했을때, git은 원격 저장소를 여러분이 작업했던 월요일의 상태로 되돌려야 할까요? 아니면 새 코드를 건들지 않고 여러분의 코드만 추가해야 되나요? 아니면 여러분의 작업은 뒤 떨어졌기 때문에 완전히 무시해야되나요?

히스토리가 엇갈려서 상황이 애매모호하기 때문에 git은 여러분이 push하지 못하게 합니다. 사실 여러분이 작업을 공유하기전에 원격 저장소의 최신 상태를 합치도록 강제합니다.



그림과 같은 상태에서 다음 명령은 실행되지 않아서 아무것도 잃어나지 않습니다.

```
git push
```

여러분의 최근 커밋 C3가 원격저장소의 C1을 기반으로 하기 때문에 git push가 실패합니다. 원격저장소는 C2까지 갱신된 상태기때문에 git은 여러분의 push를 거부하게됩니다.

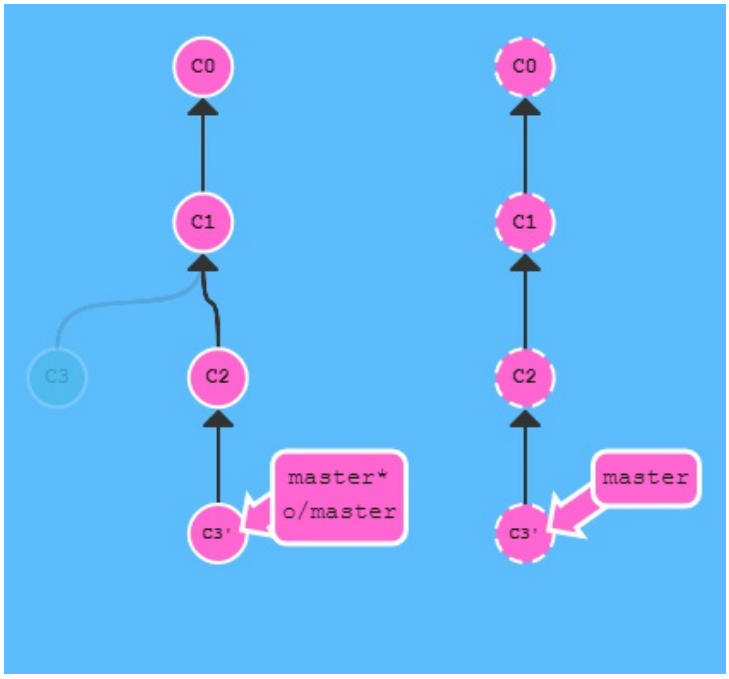
그러면 이 상황을 어떻게 해결할까요? 쉽습니다, 여러분의 작업을 원격 브랜치의 최신상태를 기반으로 하게 만들면 됩니다.

이렇게 하기위한 방법이 여러가지가 있는데, 가장 간결한 방법은 리베이스를 통해 작업을 옮기는 방법입니다.

```
git fetch && git rebase origin/master
```

```
git push
```

git fetch로 원격 저장소의 변경정보를 가져오고, 새 변경들로 우리 작업을 리베이스 했습니다. 그런 다음 이제 git push하면 업로드가 됩니다! 다음 화면을 참고하여 작업결과를 확인하세요.



rebase 명령을 사용하지 않고 해결할 수 있는 다른 방법으로는 merge가 있습니다.

git merge가 여러분의 작업을 옮기지는 않지만 merge 커밋을 생성합니다. git에게 원격 저장소의 변경을 합쳤다고 알려주는 방법중에 하나입니다. 이제 원격 브랜치가 여러분 브랜치의 부모가 되었기때문입니다. 여러분의 커밋이 원격 브랜치의 모든 커밋을 반영했다는 뜻이죠.

```
git fetch && git merge origin/master
```

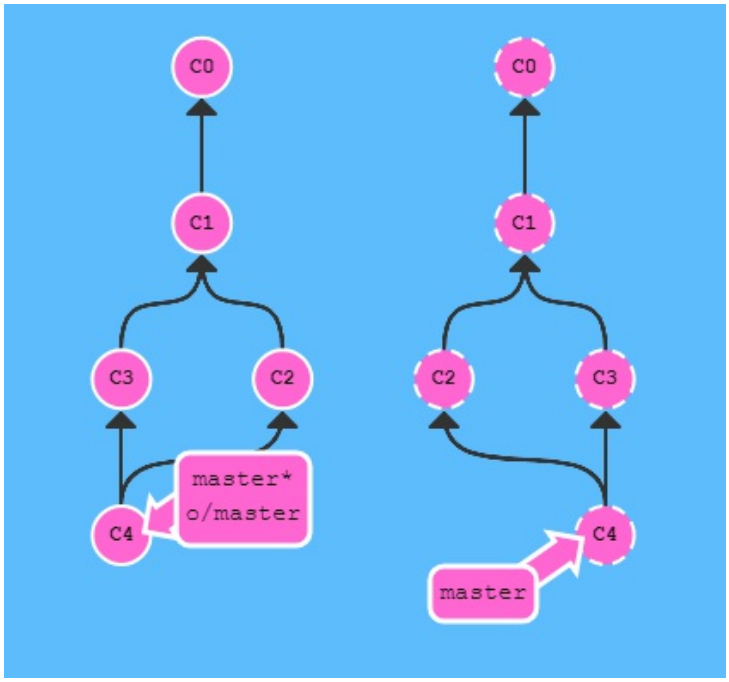
```
git push
```

위 명령을 줄여서 다음과 같이 사용할 수 있습니다. 결과는 같습니다.

```
git pull
```

```
git push
```

git fetch로 원격 저장소의 변경정보를 가져옵니다. 원격 저장소의 변경을 반영하기 위해서 새 작업을 우리 작업으로 병합했습니다. 이제 git push하면 업로드가 됩니다.



rebase 대신 merge 했을 때에 차이점을 그림으로 살펴보세요.

명령어를 좀더 적게써서 하는 방법은 없나요? 물론 있습니다. 여러분은 git pull이 fetch와 merge의 줄임 명령어라는 것은 이미 알고 있을 것입니다. 아주 간단하게 `git pull --rebase` 명령을 사용하면 fetch 후 merge 대신 rebase를 수행합니다.

```
git fetch && git rebase origin/master
```

```
git push
```

위 명령을 줄여서 다음과 같이 사용할 수 있습니다. 결과는 같습니다.

```
git pull --rebase
```

```
git push
```

팀 프로젝트 수행 시나리오

팀장 **manager**

1. 프로젝트 생성

- Client : Angular Project 콘솔에서 `ng new project`
- Server : Spring Project STS에서 `spring starter project` 선택

2. 로컬 저장소 생성

```
git init
```

3. `.gitignore` 파일 생성

- 빈 폴더 공유 시 `.gitkeep` 파일 생성

4. 커밋 대상 등록 ⇒ 초기 커밋

```
git add .
```

```
git commit -m "커밋 메시지"
```

5. 원격 저장소에 업로드

```
git remote add origin [GitHub URL]
```

```
git push -u origin master
```

6. Collaborators 등록

팀원 A, 팀원 B ...

7. 팀원들에게 공지

- 팀 프로젝트 URL 통보
- 작업 파일을 구분하면 충돌이 발생하지 않는다.
- 작업 폴더 자체를 구분하면 더욱 더 충돌이 발생하지 않는다.

8. 팀 매니저도 개발에 참여

이후 내용은 팀원의 작업과 동일 함

팀원 A

1. 팀 프로젝트 다운로드

```
git clone [GitHub URL]
```

2. 코드 작성

3. 개발 된 코드 업로드

```
git add .
```

```
git commit -m "커밋 메시지"
```

`git push`

`git push` 거부 시 `git pull` 수행 후 다시 `git push`

팀원 **B**

팀원 A 작업과 동일 함

원격 추적

개발자들은 주로 큰 프로젝트를 개발할때 작업을 feature 브랜치(topic 브랜치)들에서 수행하고 작업이 끝나면 그 작업을 주 브랜치에 통합하여 반영합니다.

어떤 개발자들은 master 브랜치에 있을때만 push와 pull을 수행합니다. 이렇게하면 master는 원격 브랜치(origin/master)의 상태와 동일하게 최신의 상태로 유지될 수 있기 때문입니다.

이런 작업흐름은 다음 두가지 작업을 병행하게 됩니다.

- feature 브랜치의 작업을 master로 통합하는 작업
- 원격저장소에 push하고 pull하는 작업

pull 작업을 하는 도중, 커밋들은 origin/master에 내려받아 지고 그다음 master 브랜치로 merge됩니다. push 작업을 하는 도중, master 브랜치의 작업은 원격의 master브랜치로 push 됩니다. 로컬의 origin/master는 원격의 master브랜치와 동기화 됩니다. push의 목적지는 master와 origin/master의 연결에서 결정됩니다.

간단히 말해서, 이 master와 origin/master사이의 연결은 브랜치의 "원격 추적" 속성을 통해 간단하게 설명됩니다. master브랜치는 origin/master브랜치를 추적하도록 설정되어 있습니다. 이것은 master가 merge와 push할 내재된 목적지가 생겼다는 뜻 입니다.

여러분은 어떻게 이 속성을 지정해주는 그 어떤 명령어 없이 master 브랜치에 설정되있는지 궁금할것 입니다. 사실, 여러분이 git으로 저장소를 clone할 때 이 속성이 여러분을 위해 자동으로 설정됩니다.

clone을 진행하면서 git은 원격 저장소에있는 모든 브랜치에 대해 로컬에 원격 브랜치를 생성합니다. origin/master 같은것들 말이죠. 그 후 원격 저장소에서 현재 active한 브랜치를 추적하는 로컬 브랜치를 생성합니다, 대부분의 경우 master가 됩니다.

git clone이 완료되면, 여러분은 오로지 하나의 로컬 브랜치를 가지게 됩니다. 물론 원격 저장소에 있는 여러 다른 브랜치도 여전히 확인할 수 있습니다.

여러분이 clone을 수행할 때 아래의 명령어를 볼 수도 있는 이유입니다:

```
local branch "master" set to track remote branch "origin/master"
```

개발자가 직접 지정할 수도 있나요?

당연하죠! 여러분은 임의의 브랜치가 origin/master를 추적하게 만들 수 있습니다. 이렇게 하면 이 브랜치 또한 내재된 push, merge 목적지를 master로 할 것입니다. 여러분은 이제 totallyNotMaster라는 브랜치에서 git push를 수행해서 원격 저장소의 브랜치 master로 작업을 push할 수 있습니다.

이 속성을 설정하는데에는 두가지 방법이 있습니다.

방법 #1

첫 번째는 지정한 원격 브랜치를 참조해서 새로운 브랜치를 생성하여 checkout 하는 방법 입니다.

```
git checkout -b totallyNotMaster origin/master
```

위 명령을 실행하면 totallyNotMaster라는 이름의 새 브랜치를 생성하고 origin/master를 추적하게 설정합니다. 브랜치의 이름을 전혀 다른것으로 지었는데도 불구하고 우리 작업이 원격 저장소의 master와 pull/push 될 수 있습니다.

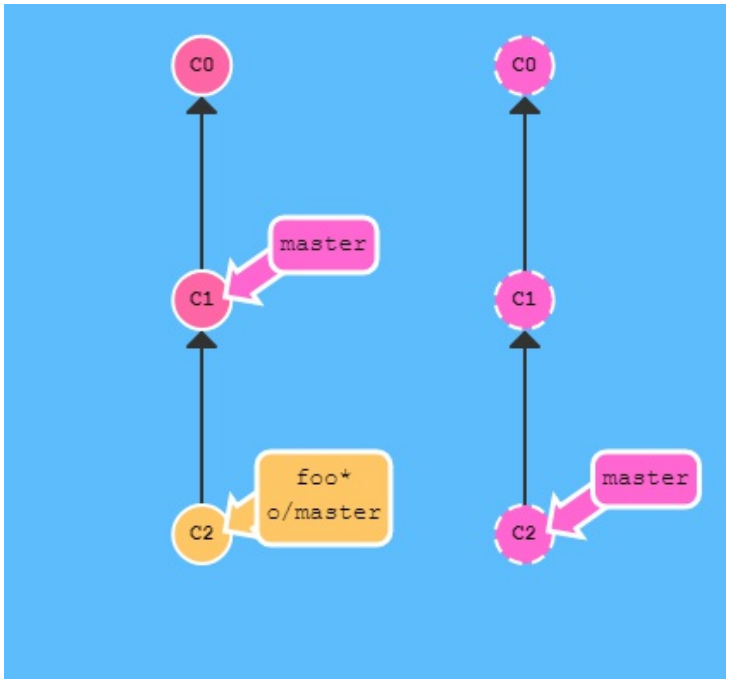
방법 #2

브랜치에 원격 추적 설정을 하는 또 다른 방법으로는 간단하게 git branch -u 옵션을 사용하는 방법이 있습니다.

```
git branch -u origin/master foo
```

foo 브랜치가 이미 만들어진 상태에서 수행합니다. 위 명령은 foo 브랜치가 origin/master를 추적하도록 설정합니다. 만약 foo가 현재 작업하고 있는 브랜치라면 foo를 생략해도 됩니다.

```
git branch -u origin/master foo
```
```
git commit
```
```
git push
```



```
git checkout master
```

```
git branch side
```

```
git branch -u origin/master side
```

```
git checkout side
```

```
git commit -m "updated"
```

```
git pull --rebase
```

```
git push
```

위 명령 대신 아래처럼 사용할 수도 있습니다.

```
git checkout master
```

```
git checkout -b side origin/master
```

```
git commit -m "updated"
```

```
git pull --rebase
```

```
git push
```

따라해 보기

명령을 하나씩 따라하면서 그 의미를 추론해 보세요.

새 프로젝트 생성

```
mkdir git_lesson
cd git_lesson
```

로컬 깃 저장소 생성

```
git init
git status
```

새 파일 생성, 깃 추적 확인

```
echo 'Hello World' > hello.txt
git status
```

스테이지에 등록

```
git add hello.txt
git status
```

커밋하기

```
git commit -m 'first commit'
git status
```

커밋 히스토리 조회

```
git log
```

브랜치 목록 조회

```
git branch
```

새 브랜치 생성

```
git branch hotfix1
git branch
```

브랜치 이동

```
git checkout hotfix1
git branch
```

파일목록 조회

```
ls -al
```

파일에 코드 추가, 상태 확인

```
echo 'Good Night' > hello.txt
type hello.txt
git status
```

워킹 디렉토리 파일 변경내용 되돌리기

```
git checkout HEAD -- hello.txt
type hello.txt
git status
```

-- 옵션은 앞 부분에 옵션과 패스 문자열을 구분하는 용도로 사용합니다. `git checkout hello` 에서 hello는 브랜치임을 나타냅니다. `git checkout -- hello` 에서 hello는 파일 패스를 가리킵니다.

파일에 코드 추가, 상태 확인

```
echo 'Good Night' > hello.txt
type hello.txt
git status
```

스테이징 및 커밋하기

```
git commit -am 'changed hello.txt'
git status
git log
```

브랜치 전환

```
git branch
git checkout master
git branch
```

브랜치 합병하기

```
git merge hotfix1
type hello.txt
git status
```

코드 변경하기

```
echo 'added on master' >> hello.txt
git status
```

커밋하기

```
git commit -am 'updated hello.txt'
git status
git log
```

브랜치 전환하기

```
git branch
git checkout hotfix1
git branch
type hello.txt
```

합병하기(합병 커밋 남기기)

```
git merge master --no-ff
git status
type hello.txt
```

합병 시 merge commit 기록을 남기기 위해서 `--no-ff` 옵션을 사용한다.

합병 커밋 취소하기

```
git reset --merge ORIG_HEAD
git log
git status
```

```
type hello.txt
```

`ORIG_HEAD` 는 합병 커밋이 아닌 이전에 수행된 오리지널 커밋을 가리킨다. 실수로 지운 커밋을 되돌릴 때 사용한다.

A1. VI 사용법

VI 에디터의 세 가지 모드

1. 일반 모드

파일 진입 시, 첫 모드는 일반모드입니다. 입력모드, 명령모드에서 `esc` 키를 누르면 일반모드로 전환됩니다.

2. 입력 모드

일반모드에서 `i` , `o` , `a` 키 중에 하나를 누르면 입력모드로 전환됩니다.

- i 현재 커서의 위치부터 텍스트 삽입을 시작합니다.
- o 현재 커서의 바로 다음에 새 줄을 추가하고 다음 줄에서 삽입을 시작합니다.
- a 현재 커서의 위치 다음 글자부터 텍스트 삽입을 시작합니다.

3. 명령 모드

일반모드에서 `:` 키를 누르면 명령모드로 전환됩니다. 커서 이동도 명령이므로 명령 모드에서 수행해야 합니다.

- w 저장합니다.
- q 종료합니다.
- wq 저장 후 종료합니다.
- d d 현재 줄을 삭제합니다.
- y y 현재 줄을 복사합니다.
- x 커서가 위치한 곳의 글자를 잘라냅니다.
- p 복사한 내용을 붙여 넣습니다.