Git and Build Server

temporary used

Copyright Note

- Author: Youngdeok

- Issue Date: 2016-08-23
- Revision #: rev02 (temporary for engineering students)
- Homepage: acaroom.net

- email: medusakiller@gmail.com

- Cooperated with MDS Academy

5. 서버 구축 실무 II

5.1 FTP와 NFS

5.2 웹서버 구축과 운영

5.3 버전, 형상관리의 개념

5.4 Git 서버의 구축과 운영

5.5 지속적 통합(CI)

5.6 배포와 자동화

버전 관리 시스템

- 개념
  - 언제, 누가, 어떻게 변경했는지를 기록으로 남기는 것은 매우 중요함.
  - 어떤 문제가 발생했을 때 그 기록을 추적하면 원인을 판명하는데 도움이 된다.

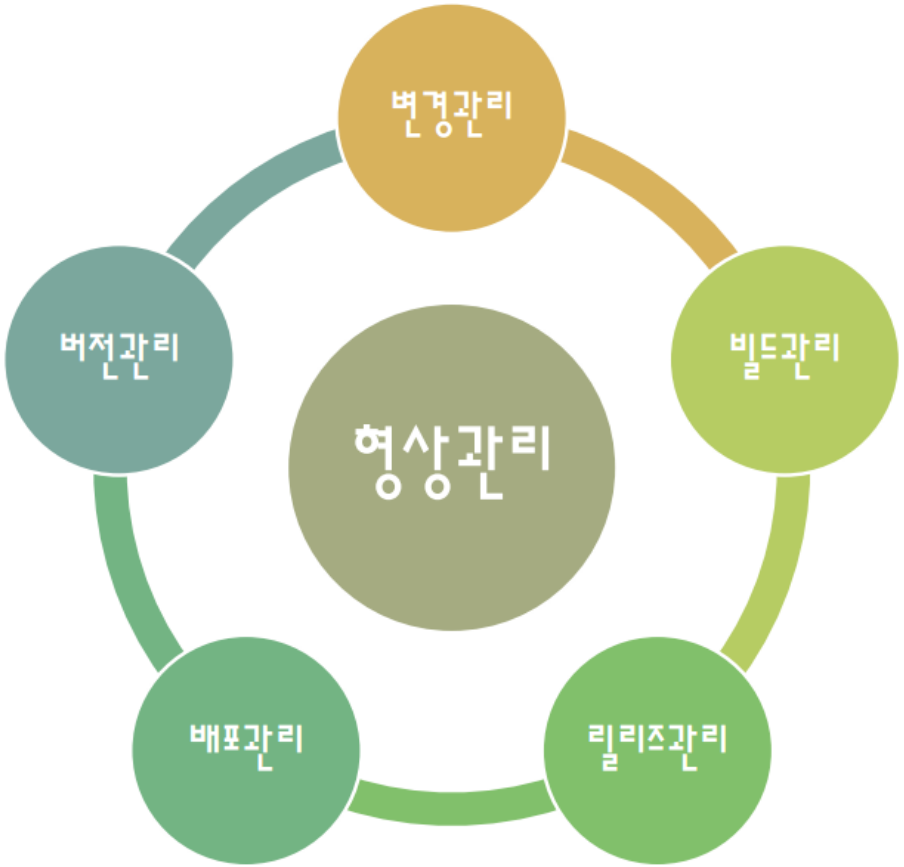
- 버전 관리의 이점
  - 변경 내용이라는 가장 기본적인 기록이 남는다.
  - 버전 간 차이를 간단히 확인할 수 있다.
  - 다른 사람의 변경을 실수로 덮어쓰는 사태를 방지할 수 있다.
  - 임의의 시점으로 복원이 가능하다.
  - 복수의 파생 데이터를 만들 수 있어서 특정 시점의 상태 데이터를 저장할 수 있다.

형상 관리(CM: Configuration Management)

- 형상 관리의 시작
  - SW개발 유지 보수 과정에서 발생하는 소스코드, 문서 인터페이스 등 각종 결과물에 대해 형상을 만들고, 이들 형상에 대한 변경을 체계적으로 관리, 제어하기 위한 활동
  - CVS나 SVN, 또는 GIT와 같은 버전 관리 시스템을 이용할 수 있다.

용어	설명
중앙 저장소(Repository)	원본 소스를 저장하고 있는 저장소
작업 디렉터리(Working Copy)	원본 저장소로부터 체크아웃을 통해 내려 받은 내 로컬 PC에 있는 작업 사본 디렉터리
커밋(Commit) or 체크인(Check-In)	작업 디렉터리에서 변경, 추가 및 삭제된 파일을 원본 저장소인 서버에 적용하는 것
갱신(Update)	체크아웃을 받은 작업 디렉터리를 원본 저장소의 가장 최신 커밋된 버전까지 업데이트하는 명령어
리비전(Revision)	소스 파일을 수정하여 커밋하게 되면 일정한 규칙에 의해 숫자가 증가한다. 저장소에 저장된 각각의 파일 버전이라 할 수 있다.
되돌리기(Roll Back)	작업 디렉터리에 저장되어 있는 사본을 특정 리비전 또는 특정 시간으로 복원할 수 있도록 하는 명령

형상 관리의 범위



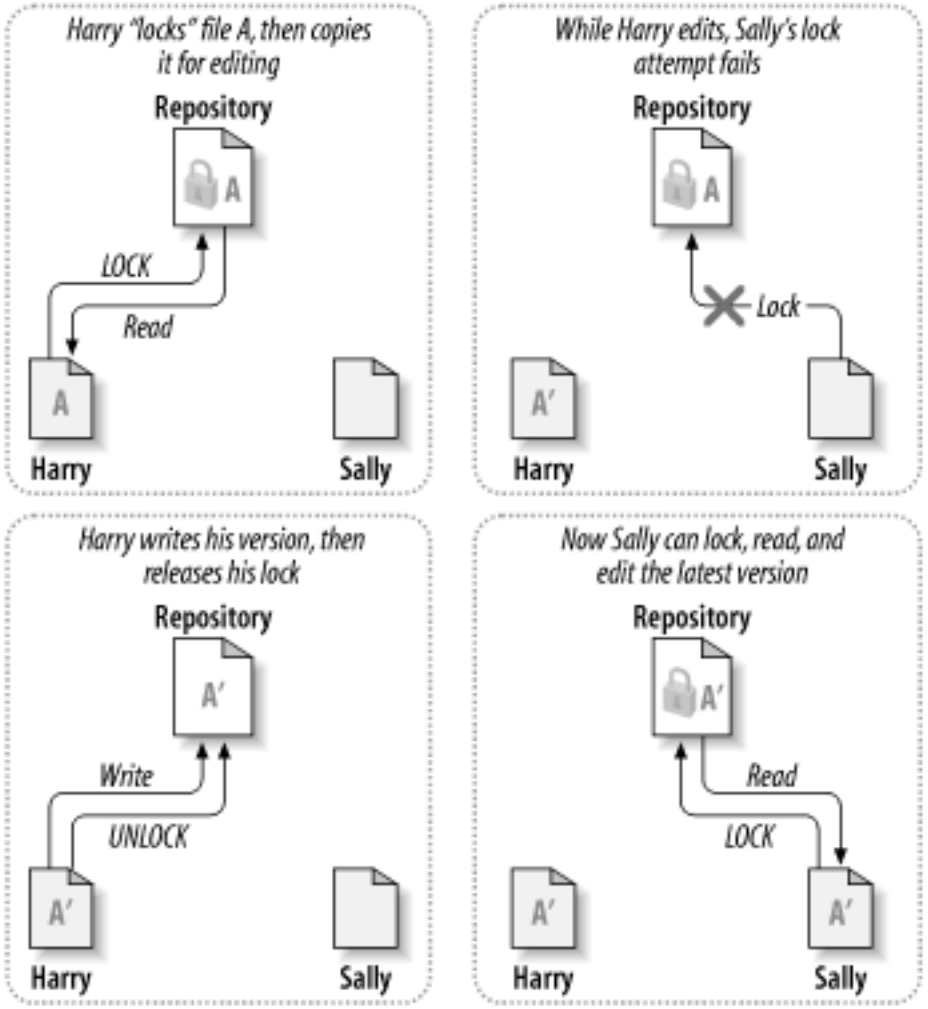
관리 시스템의 종류

- 일반적인 변경 확인
  - vim -d source target, svn diff, git diff 등의 명령
- 상용 버전 관리 시스템의 종류
  - IBM Rational ClearCase
  - VisualSourceSafe (락(Lock)모델)
  - Perforce (락모델)
  - PTC Integrity
- 오픈 소스 (대부분 머지;Merge 모델)
  - Subversion(SVN) TortoiseSVN (GUI) 클라이언트-서버 모델 (중앙집중형)
  - CVS 클라이언트-서버 모델
  - Git 분산 모델 TortoiseGit (GUI) GitHub, Trac (웹기반)

형상 관리의 락(Lock)모델과 머지(Merge) 모델

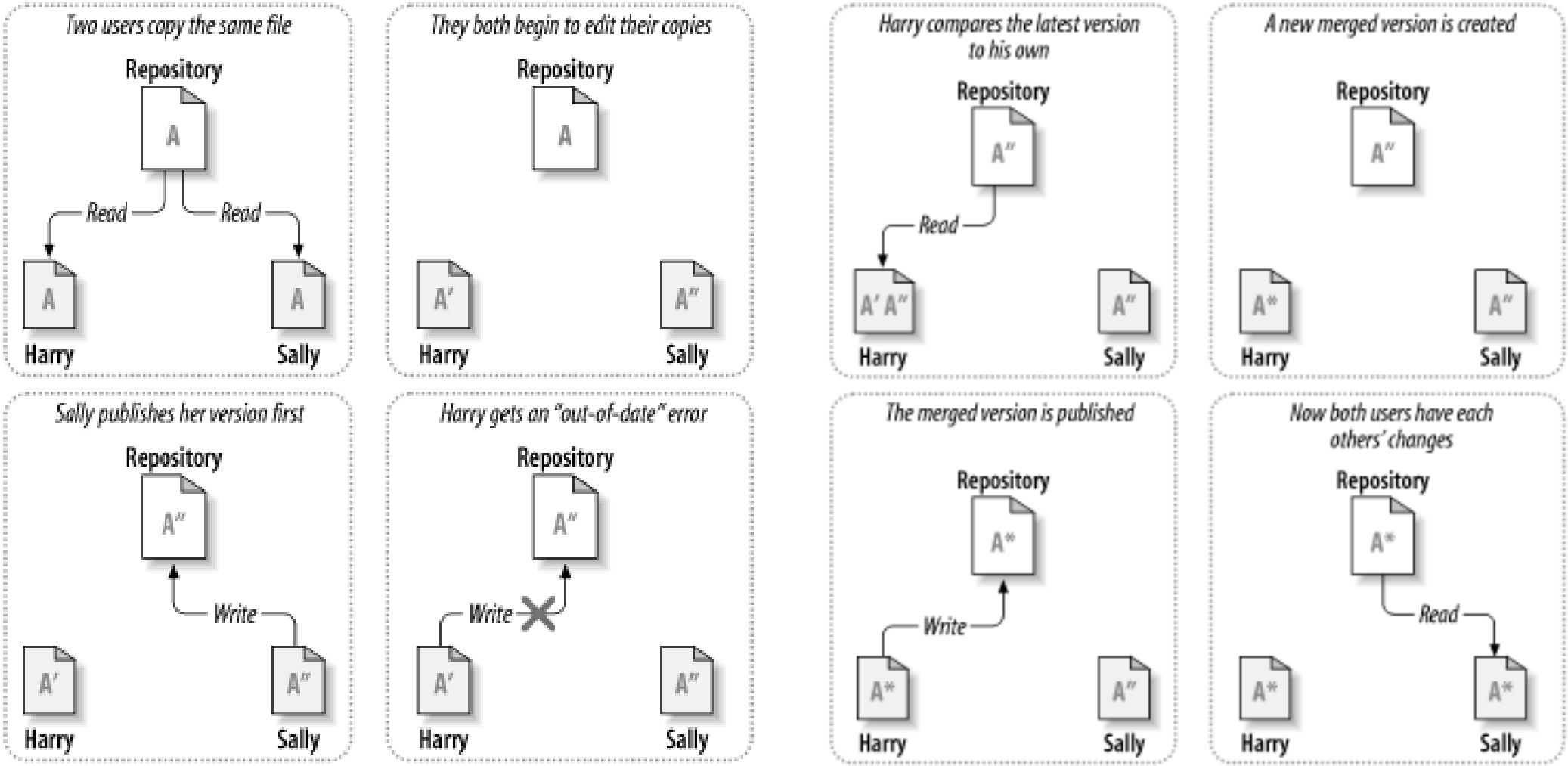
- 락모델(Lock-modify-unlock)
  - 누군가 편집하고 있는 사이에 파일을 잠가 둬서 다른 멤버가 편집하지 못하도록 하여 데이터 일관 된 편집이 가능하다.
  - 방식이 단순하여 쉽게 이해할 수 있지만 여러명이 동시에 병행해서 개발을 진행하기가 어려워 개발 속도가 잘 나지 않는 문제가 있다.
- 머지 모델(copy-modify-merge)
  - 파일을 잠그지 않고 개발자가 복사본을 체크아웃해서 편집한 후 리포지토리에 커밋한다.
  - 여러명이 동시에 최신 소스 코드를 취득해서 다른 사람의 작업을 기다리지 않고 병행해서 개발할 수 있다는 특징이 있다.
  - 다른 맴버와 수정이 충돌 하는 경우는 머지를 통해 해결할 수 있다.

락모델과 머지모델 (1/2)



Lock-modify-unlock Model

락모델과 머지모델 (2/2)



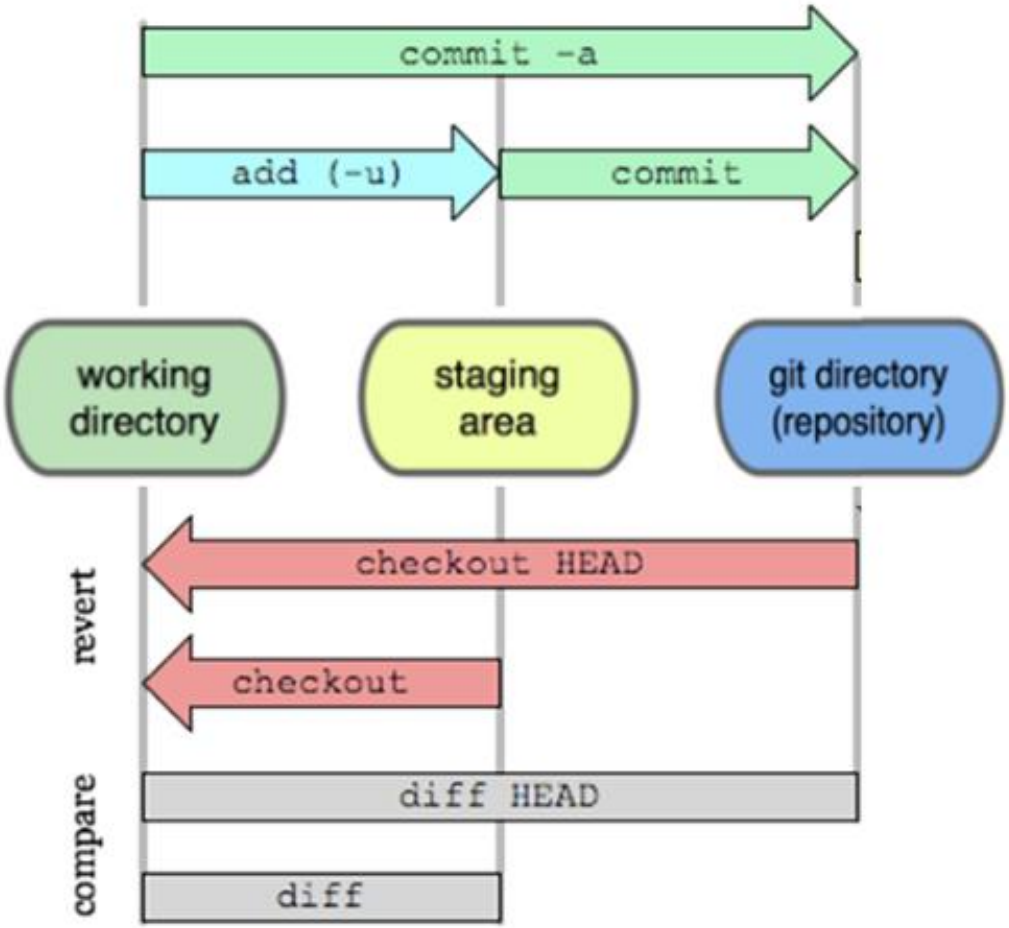
Copy-modify-merge Model

분산 버전 관리 시스템

- 장점
  - 리포지토리의 완전한 복사본을 로컬 장비에 둘 수 있다.
  - 처리 속도가 빠르다.
    - 모든 파일이 로컬 장비에 있으므로 통신에 따른 부하가 없음
  - 일시적인 작업에 대한 이력 관리가 쉽다.
  - 브랜치, 머지가 쉽다.
  - 장소에 구애 받지 않고 협업이 가능하다.
- 단점
  - 진정한 의미의 최신 버전은 시스템상에 존재하지 않는다.
    - (실제 운영 시 중앙 리포지토리 편리성으로 GitHub를 사용해서 클론하는 방식을 많이 사용)
  - 진정한 의미의 리비전 번호는 없다.
    - 이 때문에 분산 버전 시스템의 체인지셋 하나하나에 GUID가 부여
    - 즉, 리비전 이력을 관리하는 것이 아닌 체인지셋이 중복되지 않도록 한다.
  - 작업의 유연성 때문에 오히려 혼란을 야기한다.

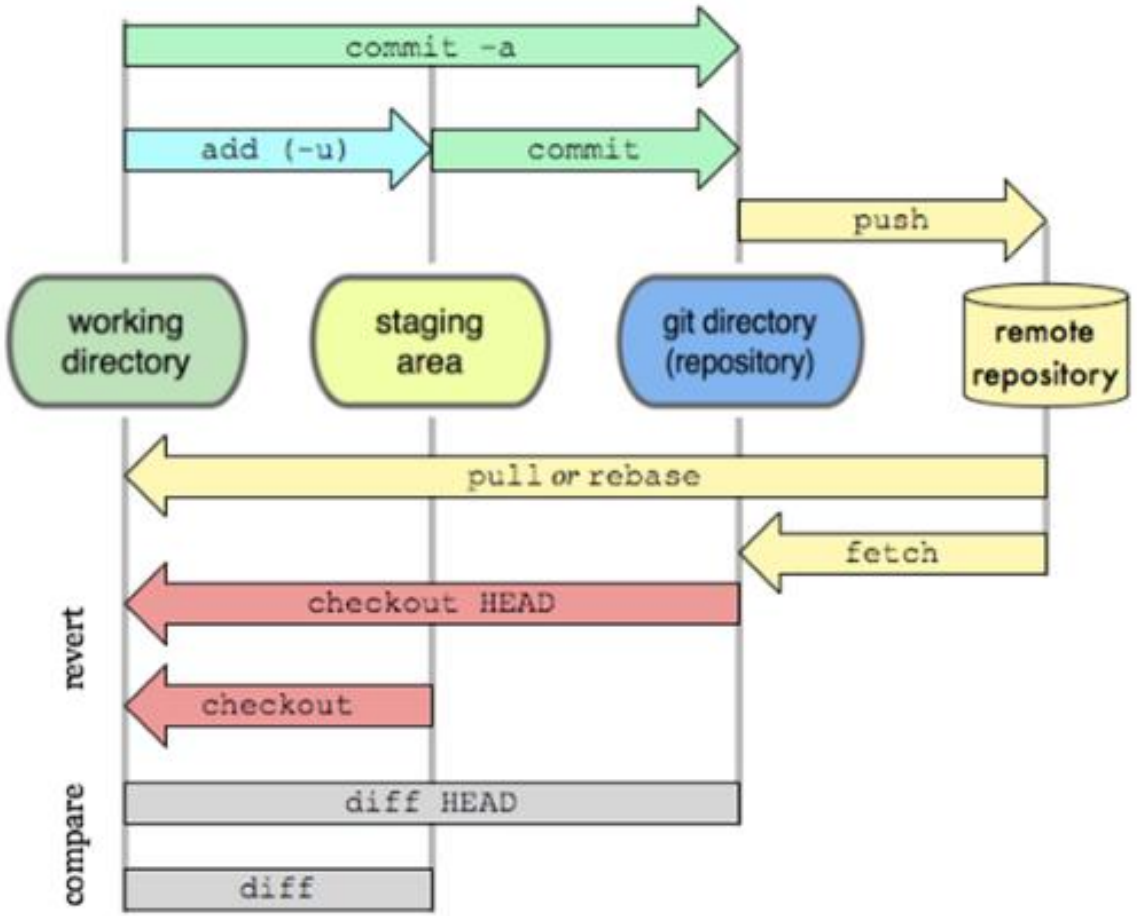
Git의 작업 (1/2)

- Git의 세가지 상태
  - 제출된 상태(Committed)
  - 수정된 상태(Modified)
  - 준비 영역에 추가된 상태(Staged)
- Git 디렉터리(Local Repository)
  - Git 프로젝트의 메타데이터와 객체 DB가 저장된 곳
- Working 디렉터리
  - 프로젝트의 특정 버전을 Checkout
- Staging Area(준비영역)
  - 곧 commit할 파일에 대한 정보



Git의 작업 (2/2)

- GitHub와 같은 원격 저장소가 있는 경우
  - https://github.com
  - 프로젝트를 안전하게 보존하고 여러 사용자가 공동으로 관리 (Remote Repository)
  - push와 fetch명령이 추가된다.



Q/A

5. 서버 구축 실무 II

5.1 FTP와 NFS

5.2 웹서버 구축과 운영

5.3 버전, 형상관리의 개념

5.4 Git 서버의 구축과 운영

5.5 지속적 통합(CI)

5.6 배포와 자동화

Git 서버 구축 (1/3)

- 깃 설치
  - sudo apt-get install git
  - git --version 버전 정보 확인
- 사용자 정보 설정
  - git를 사용하기 전에 사용자 정보를 설정한다.
  - git config --global user.name "Youngdeok Hwang"
  - git config --global user.email "medusakiller@gmail.com"
  - git config --global --list 설정 내용 확인

Git 서버 구축 (2/3)

- 저정소의 설정
  - mkdir ~/gittest; cd ~/gittest
  - git init → .git에 모든 파일을 생성하며 git이 관리하게 된다.
  - git status → 디렉터리 상태 확인
  - vim hello.c
  - git status → hello.c가 untracked 상태이다.
  - git add hello.c → 저장소에 파일을 추가하여 관리대상(staged) 상태가 됨
  - git commit -m 'initial project version' → 변경 내용의 확정
    - 최초의 master 브랜치가 생성됨

Git 서버 구축 (3/3)

- 로그의 확인
  - git log
    - -p patch 로그를 보여줌
    - --word-diff 단어단위로 확인을 위해
    - --stat 변경 통계
    - -2 최근 두개의 결과
    - --pretty=oneline 한줄에 로그요약

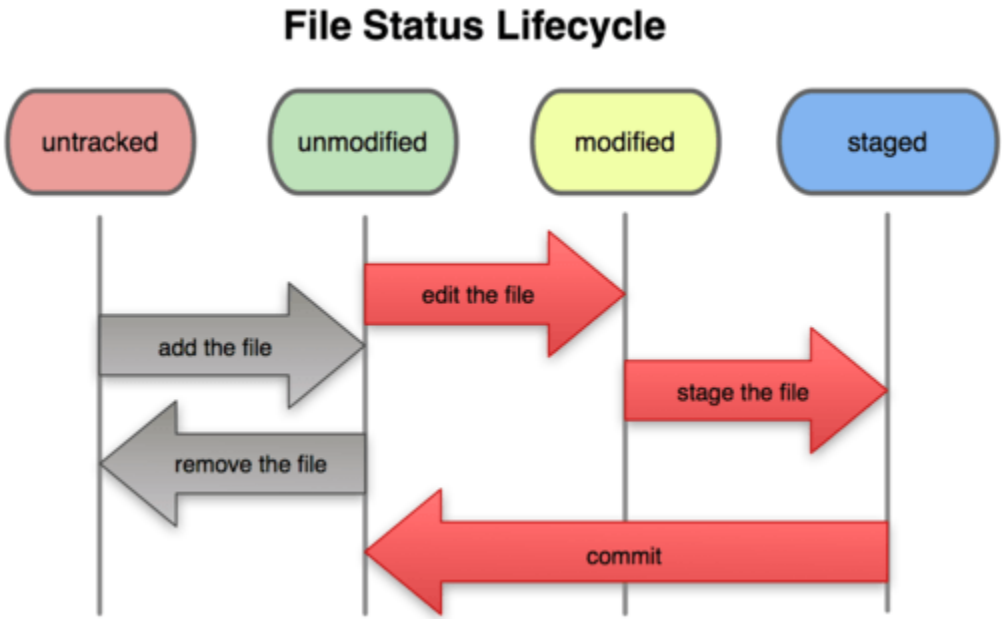
```
edu@mail:~/gittest$ git log -p
commit 7853c7250b23cd119b19e39ce4c9639bf2b2dbd6
Author: Youngdeok Hwang <medusakiller@gmail.com>
Date:   Thu Mar 26 13:04:33 2015 +0900

    initial project version

diff --git a/hello.c b/hello.c
new file mode 100644
index 0000000..90a3df7
--- /dev/null
+++ b/hello.c
@@ -0,0 +1,7 @@
+#include <stdio.h>
+
+int main(void) {
+       printf("Hello World!~\n");
+       return 0;
+}
+
edu@mail:~/gittest$
```

Git 서버의 기본적인 이용 (1/2)

- 파일의 수정과 상태 확인
  - vim hello.c
  - git status
  - git add .
  - git commit -m 'added new line'
  - git status
  - git log
  - git log -p



Git 서버의 기본적인 이용 (2/2)

- 파일 무시하기
  - hello.c을 컴파일하거나 빌드하면 오브젝트(*.o)파일등의 파일은 추가하거나 Untracked 파일이라고 보여줄 필요가 없다. 파일을 무시하려면 .gitignore파일에 무시할 패턴을 적는다.
  - cd ~/gittest; vim .gitignore

```
*.[oa]
*~
*.out
```

    - 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.

Git 의 확인과 삭제

- 수정 사항의 확인과 삭제
  - hello.c 파일을 수정하고 다음 명령을 내려보자
  - git diff
  - git add . ; git commit
    - add 가 귀찮은 경우 git commit -a -m 'modified hello.c' 과 같이 -a 옵션을 사용
  - git rm filename → 파일의 삭제
  - git mv filename → 파일 이름의 변경
- Git의 시각화 도구
  - 로그를 시각화하여 GUI로 보여 준다.
  - sudo apt-get install gitk
  - gitk &

Remote 저장소의 프로젝트 이용하기 (1/3)

- Remote 저장소를 clone 하기
  - 다른 프로젝트에 참여하거나 Git 저장소를 복사하고 싶을 때
  - Git와 Subversion이 다른 큰 차이점은 서버에 있는 모든 데이터를 복사한다는 것
  - Ruby용 Git 라이브러리를 클론 해 보자
  - cd; git clone git://github.com/schacon/grit.git
    - grit 디렉터리가 자동으로 생성되며 그 안에 .git 디렉터리를 만든다.
    - 그리고 저장소의 데이터를 모두 가져와서 가장 최신 버전을 Checkout 해 놓는다.
  - 특정 이름으로 clone 하려고 할때
    - git clone git://github.com/schacon/grit.git **mygrit**

Remote 저장소의 프로젝트 가져오기

- 리모트 저장소의 확인
  - cd ~/grit
  - git remote -v 현재 프로젝트에 등록된 리모트 저장소의 확인
- 리모트 저장소 추가하기
  - git remote add origin https://github.com/youngdeok/gittest.git
  - git remote -v

```
$ git remote add origin git://github.com/youngdeok/gittest.git
$ git remote -v
origin   git://github.com/youngdeok/gittest.git (fetch)
origin   git://github.com/youngdeok/gittest.git (push)
```

  - 이제 이름 대신 스트링 origin을 사용할 수 있다. 로컬 저장소에는 없지만 origin 의 저장소에 있는 것을 가져오려면 아래과 같이 실행한다.
  - git fetch origin → git fetch {remote name}

Remote 저장소의 프로젝트 이용하기

- 리모트 저장소 생성
  - github에 gittest 저장소를 만든다.
  - 프로젝트를 공유하고 싶을 때 리모트 저장소에 Push할 수 있다.
- push 하기
  - git push [리모트 저장소 이름] [브랜치 이름]으로 단순하다. master 브랜치를 origin 서버에 Push하려면(Clone하면 보통 자동으로 origin 이름이 생성된다) 아래와 같이 서버에 Push한다.
  - git push -u origin master
  - git remote show origin → 원격의 브랜치들을 보여줌
  - git branch → 현재의 브랜치

푸시 과정

- 저장소에 push 하기
  - 프로젝트를 공유하고 싶을 때 리모트 저장소에 Push할 수 있다.
  - git push [리모트 저장소 이름] [브랜치 이름]
  - git push origin master
    - Clone한 리모트 저장소에 쓰기 권한이 있고, Clone하고 난 이후 아무도 리모트 저장소에 Push하지 않았을 때만 사용할 수 있다.
    - 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.

저장소 살펴보기와 삭제

- 저장소 보기
  - git remote show [리모트 저장소 이름]
  - git remote show origin
  - 리모트 저장소의 URL과 추적하는 브랜치를 출력한다. 이 명령은 git pull 명령을 실행할 때 master 브랜치와 머지할 브랜치가 무엇인지 보여 준다.

- 리모트저장소 변경/삭제
  - git remote rename 명령으로 리모트 저장소의 이름을 변경할 수 있다.
  - git remote rename pb paul → pb를 paul로 변경
  - git remote rm paul → paul 삭제

Remote 저장소의 프로젝트 이용하기

- 리모트 저장소의 이름 변경/삭제
  - git remote ***rename*** origin medusakiller
  - git remote
  - git remote ***rm*** medusakiller
  - git remote

클론과 머지

- 머지 과정 (pull)
  - fetch 명령은 리모트 저장소의 데이터를 모두 로컬로 가져오지만, 자동으로 머지하지 않는다.
  - 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.
  - 그리고 git pull 명령은 Clone한 서버에서 데이터를 가져오고 그 데이터를 자동으로 현재 작업하는 코드와 머지시킨다.

원격 저장소의 있는 파일 가져오기

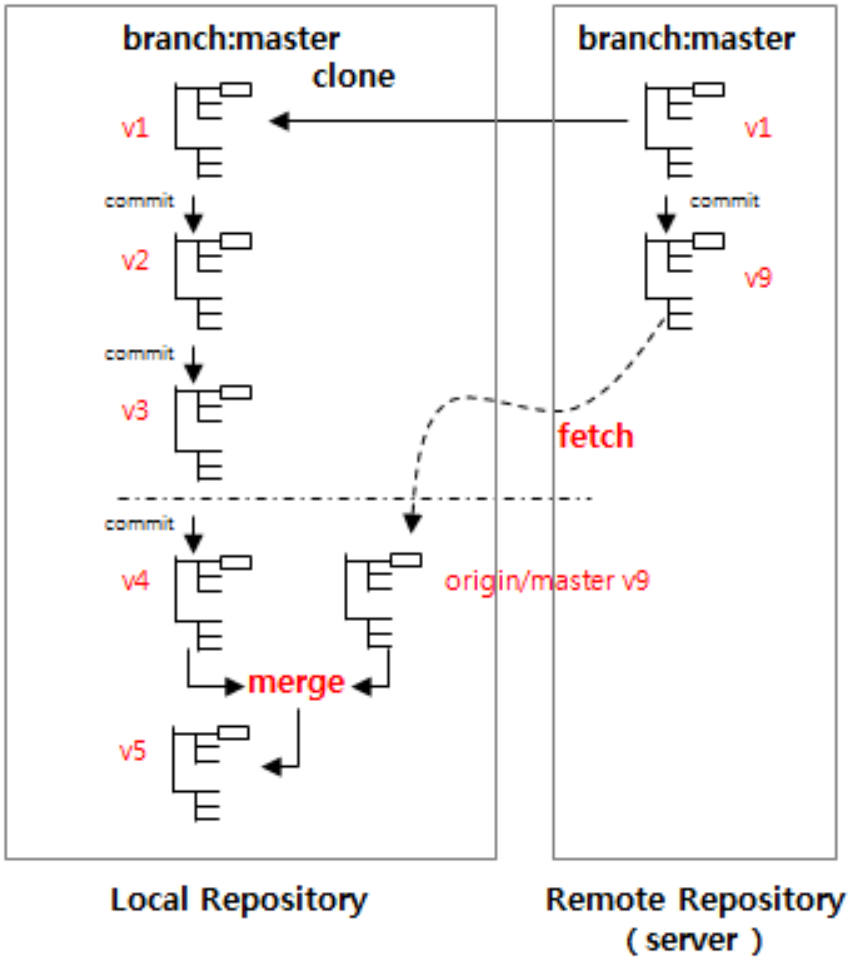
- 가져오기
  - 로컬에 commit 한 파일들을 리모트 저장소에 업로드 하기전에 먼저 리모트 저장소에 있는 파일들을 다운 받아서 동기화 시켜야 한다.
  - git fetch origin master → (리모트 저장소의 별명) (리모트 브랜치)
  - git pull origin master →(리모트 저장소의 별명) (리모트 브랜치)
    - git pull 은 git fetch 명령을 실행하고 자동으로 merge(병합) 한다.
  - 확인 후 다시 push한다.
  - git push origin master

fetch의 과정

- 그림 설명
  - 원격 저장소의 v1 버전에서 clone을 받아서 로컬 저장소에서 개발을 시작하였다. 로컬에서 여러번의 commit을 통해서, v4 버전까지 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.

fetch의 과정

- 그림


branch:master
clone
branch:master
v1
v1
commit
commit
v2
v9
commit
v3
fetch
commit
v4
origin/master v9
merge
v5
Local Repository
Remote Repository
(server)

브랜치의 생성

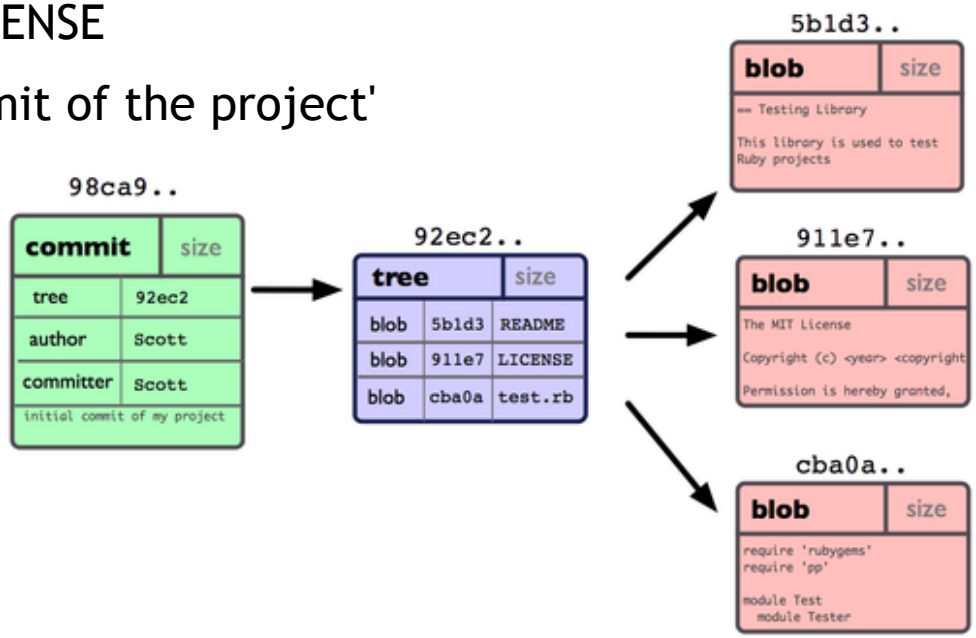
- 새로운 브랜치의 생성
  - 새로 생성한 로컬 브랜치에서 파일을 만들고 commit을 한 다음 아래와 같이 push를 하면
  - git branch coffee
  - git push origin coffee → 새로운 브랜치가 원격에 생성된다.
  - 원하는 브랜치를 변경하려면 체크아웃 한다.
  - git checkout master
  - git checkout coffee
- 브랜치에서 작업을 끝내고 master로 머지하기
  - git merge coffee → 모든 변경사항은 master에 추가된다.

Remote 저장소의 태그 사용

- 태그 붙이기
  - Annotated 태그: 만든사람 이름, 이메일 날자등을 저장, GPG로 서명할 수 있다.
  - Lightweight 태그: 브랜치와 비슷하며 특정 커밋에 대한 포인터 이다.
- Annotated 태그 생성
  - git tag **-a** v1.4 -m 'my version 1.4'
  - git tag → 태그의 조회
  - git show v1.4
- Lightweight 태그 생성
  - git tag v1.4-lw
  - git tag
  - git show v1.4-lw → 단순 커밋 정보만 보여준다.

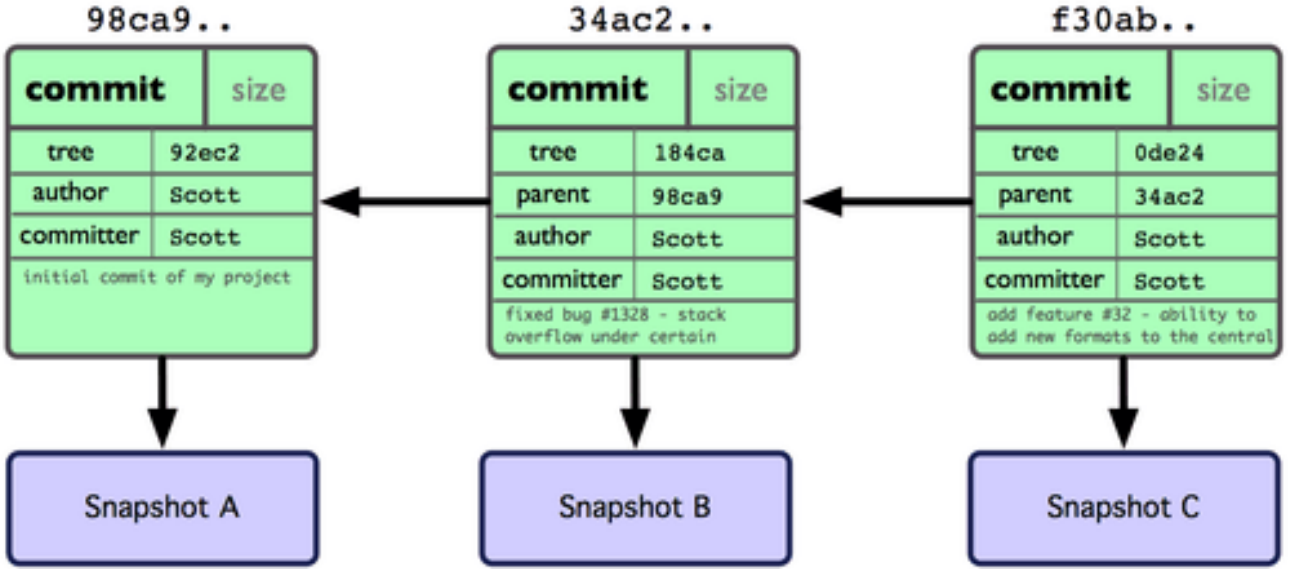
Git 브랜치 (1/8)

- 브랜치란 무엇인가?
  - 원본의 흐름과 다른 버전의 갈래를 branch라 하며 기본적으로 master branch가 있고 커밋을 할때마다 이 브랜치는 자라난다. 각 커밋은 ID로 구분한다.
- 브랜치의 생성 과정
  - 파일 3개를 만들어 브랜치 개념을 살펴보자.
    - git add README test.rb LICENSE
    - git commit -m 'initial commit of the project'



Git 브랜치 (2/8)

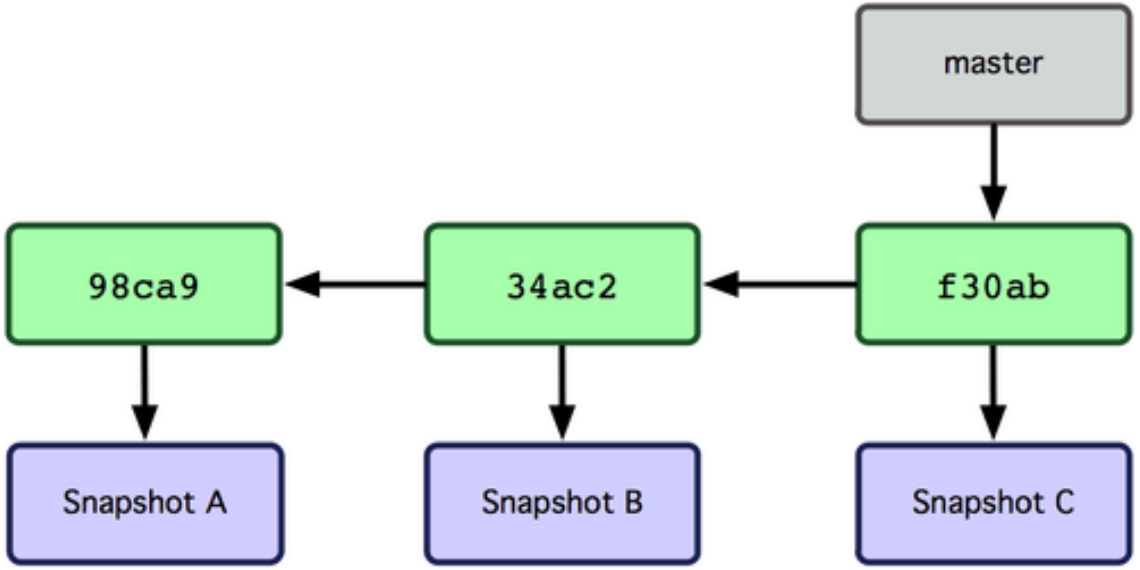
- 브랜치의 생성 과정 (Cont.)
  - 다시 파일을 수정하고 커밋하면 이전 커밋이 무엇인지도 저장한다. 커밋을 두 번 더 하면 그림과 같이 저장된다.



  - Git의 브랜치는 커밋 사이를 가볍게 이동할 수 있는 어떤 포인터 같은 것이다.

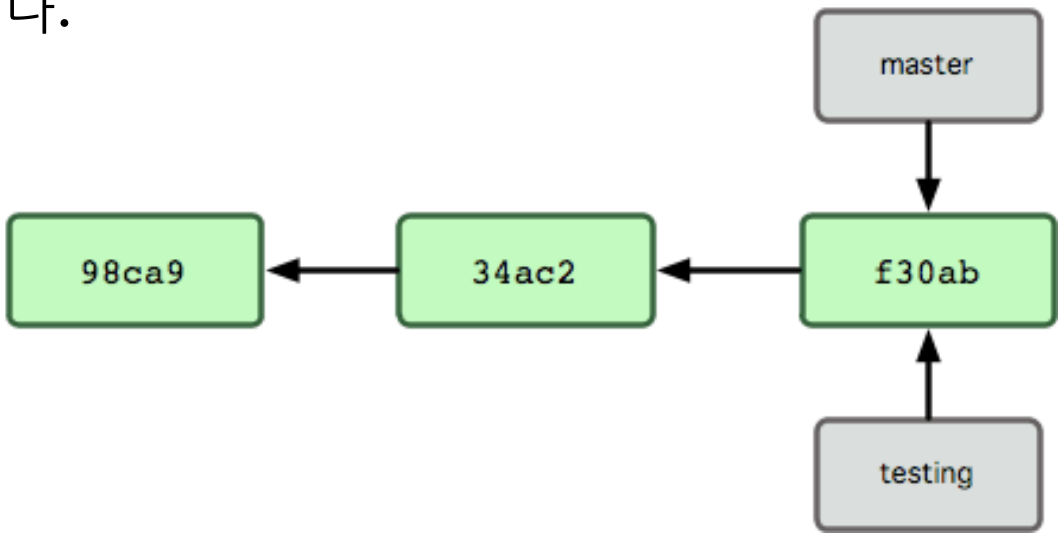
Git 브랜치 (3/8)

- 브랜치의 생성 과정 (Cont.)
  - 기본적으로 Git은 master 브랜치를 만든다. 최초로 커밋하면 Git은 master라는 이름의 브랜치를 만들어서 자동으로 가장 마지막 커밋을 가리키게 한다.



Git 브랜치 (4/8)

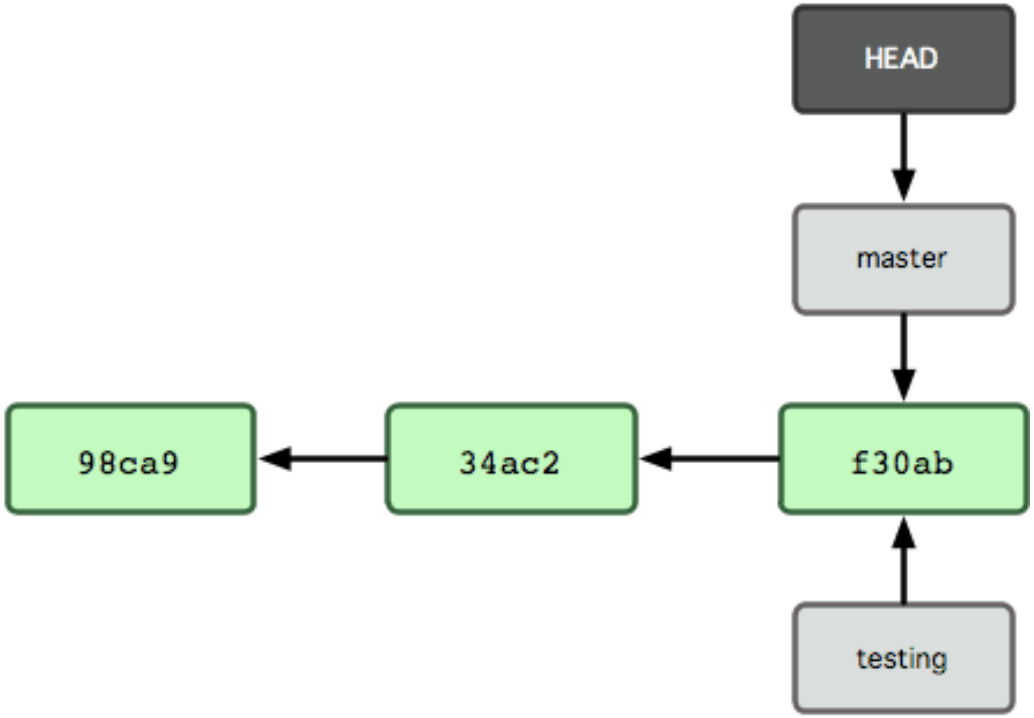
- 브랜치의 생성 과정 (Cont.)
  - 다음과 같이 git branch 명령으로 testing 브랜치를 만들면 그림과 같이 마지막 커밋을 가리킨다.



  - git branch testing → 새로운 브랜치의 생성
  - git branch -v → 상태 확인

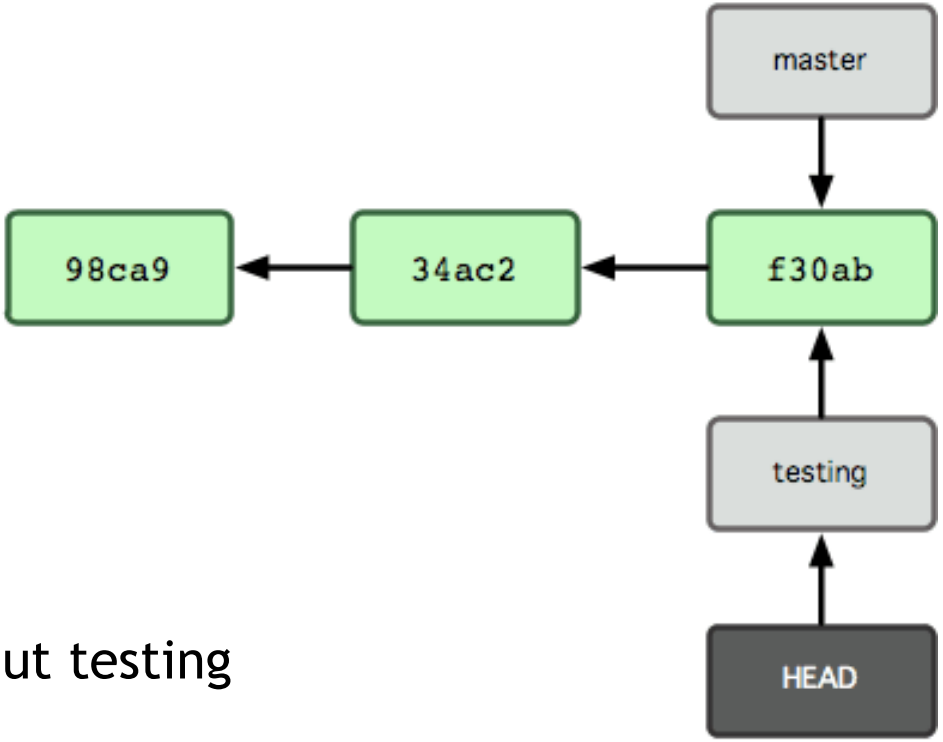
Git 브랜치 (5/8)

- 브랜치의 생성 과정 (Cont.)
  - Git은 'HEAD'라는 특수한 포인터가 있다. 이 포인터는 지금 작업하는 로컬 브랜치를 가리킨다. 브랜치를 새로 만들었지만, Git은 아직 master 브랜치를 가리키고 있다. git branch 명령은 브랜치를 만들기만 하고 브랜치를 옮기지 않는다.



Git 브랜치 (6/8)

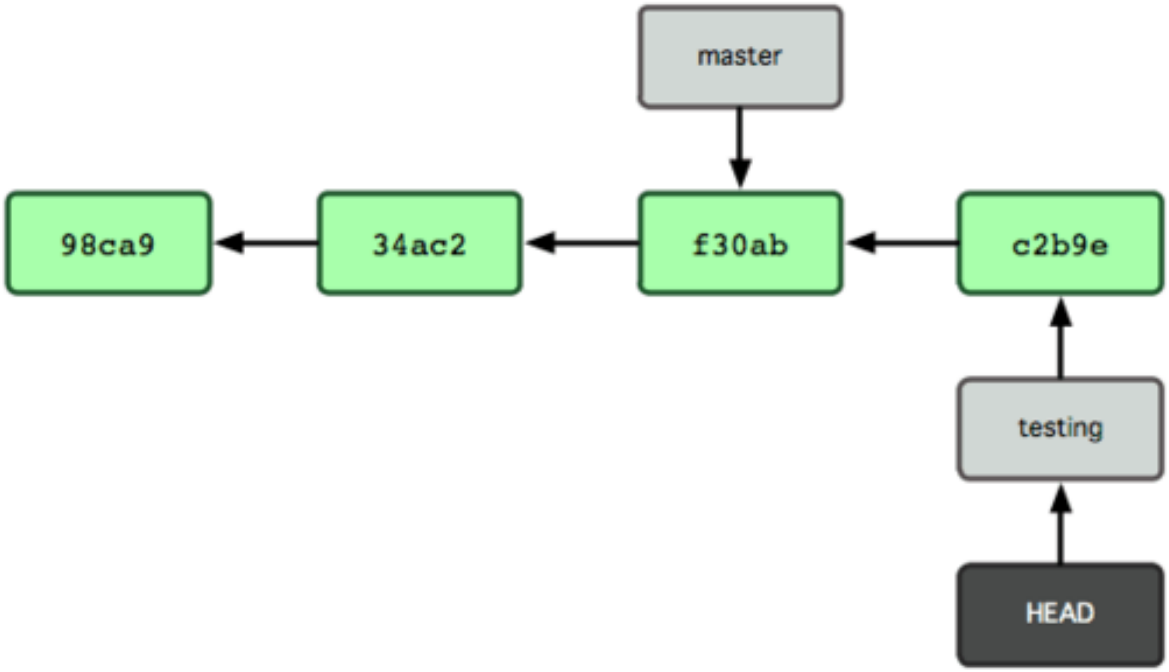
- 브랜치의 생성 과정 (Cont.)
  - git checkout 명령으로 새로 만든 브랜치로 이동할 수 있다. testing 브랜치로 이동하려면 다음과 같이 한다.
  - git checkout testing



Git 브랜치 (7/8)

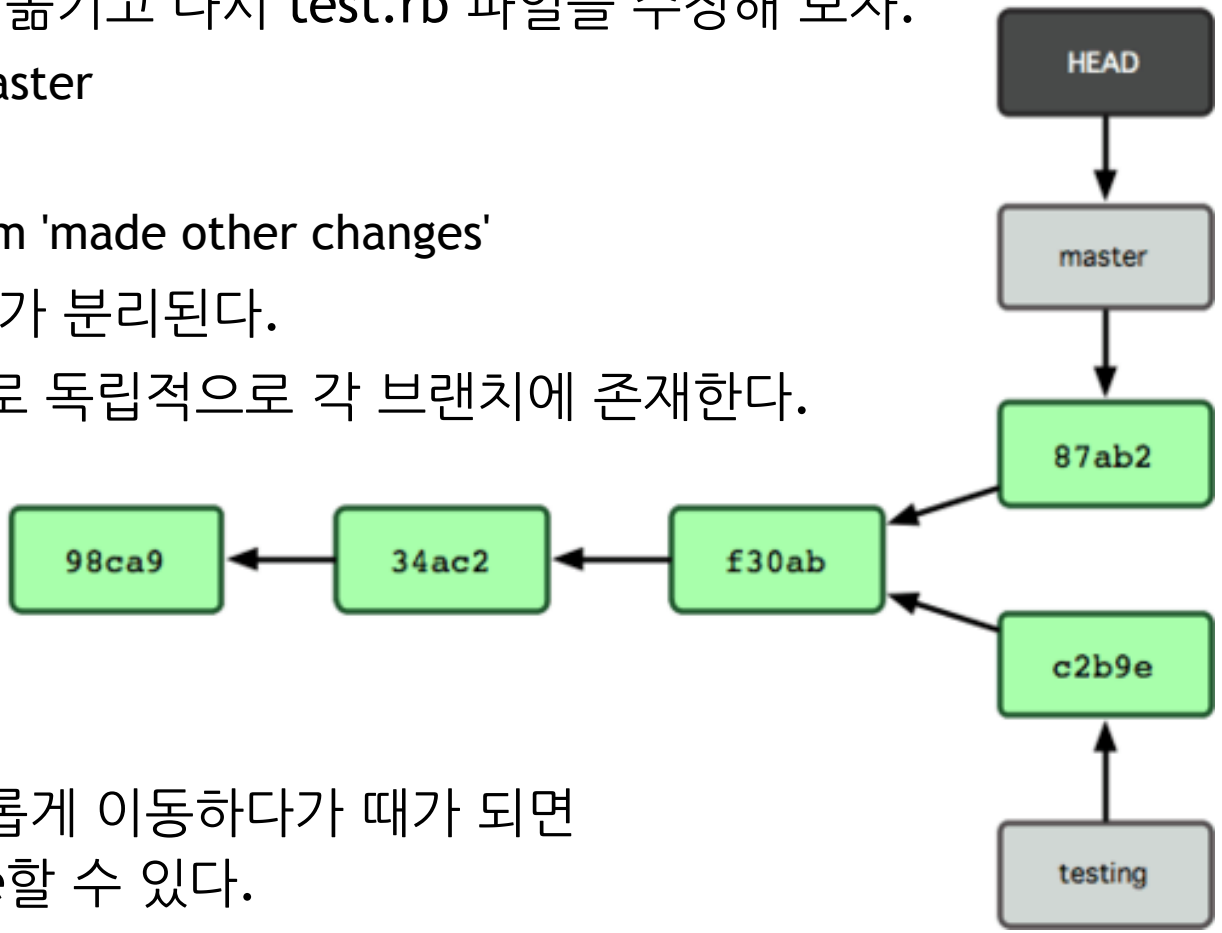
- 브랜치의 생성 과정 (Cont.)
  - 파일을 수정하고 새로 커밋을 해보자.

```
$ vim test.rb
$ git commit -a -m 'made a change'
```



Git 브랜치 (8/8)

- 브랜치의 생성 과정 (Cont.)
  - HEAD를 master로 옮기고 다시 test.rb 파일을 수정해 보자.
    - git checkout master
    - vim test.rb
    - git commit -a -m 'made other changes'
  - 프로젝트 히스토리가 분리된다.
  - 두 작업 내용은 서로 독립적으로 각 브랜치에 존재한다.



  - 커밋 사이를 자유롭게 이동하다가 때가 되면 두 브랜치를 Merge할 수 있다.

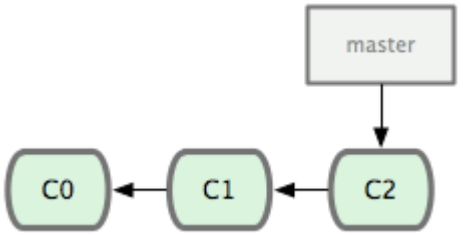
Git 브랜치와 Merge의 기초

- 실제 개발 과정의 예
  - 작업 중인 웹사이트가 있다.
  - 새로운 이슈를 처리할 새 Branch를 하나 생성.
  - 새로 만든 Branch에서 작업 중.

- 이때 중요한 문제가 발생할 경우
  - 새로운 이슈를 처리하기 이전의 운영(Production) 브랜치로 이동.
  - Hotfix 브랜치를 새로 하나 생성.
  - 수정한 Hotfix 테스트를 마치고 운영 브랜치로 Merge.
  - 다시 작업하던 브랜치로 옮겨가서 하던 일 진행.

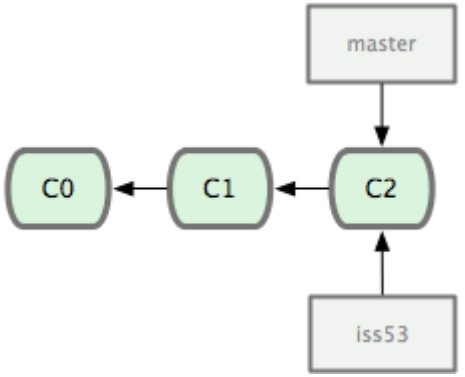
개발 시나리오 (1/8)

- 개발의 진행
  - 몇번 커밋을 한 상태를 가정하자.



  - 이슈 관리 시스템에 등록된 53번 이슈를 처리한다고 하면 이 이슈에 집중할 수 있는 브랜치를 새로 하나 만든다. 브랜치를 만들면서 Checkout까지 한 번에 하려면 git checkout 명령에 -b라는 옵션을 준다.

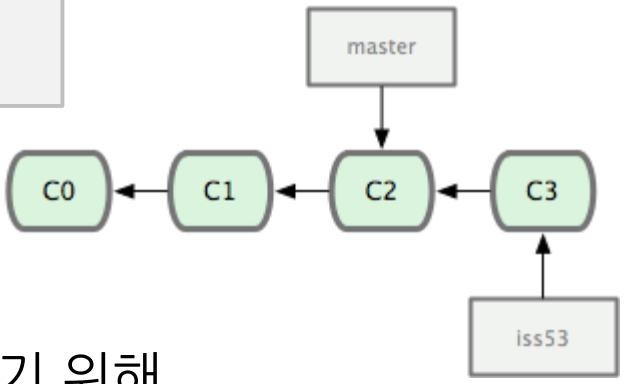
```
$ git checkout -b iss53
Switched to a new branch 'iss53'
```



개발 시나리오 (2/8)

- 개발의 진행 (Cont.)
  - iss53 브랜치를 Checkout했기 때문에(즉, HEAD는 iss53 브랜치를 가리킨다) 뭔가 일을 하고 커밋하면 iss53 브랜치가 앞으로 진행한다.

```
$ vim index.html
$ git commit -a -m 'added a new footer [issue 53]'
```



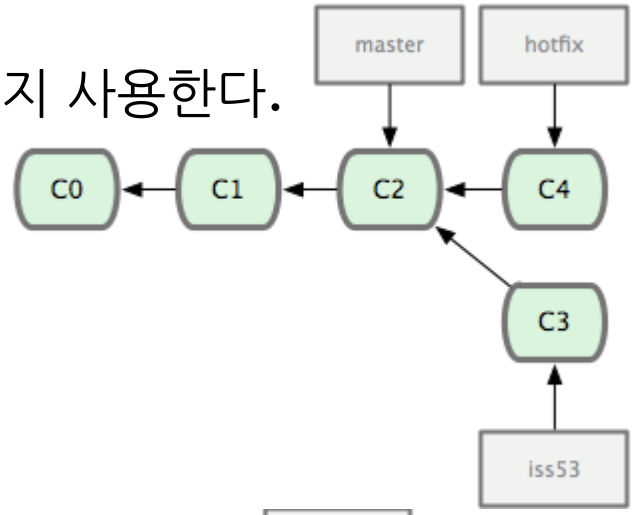
  - 만드는 사이트에 문제가 생겨서 즉시 고쳐야 한다.
  - 버그를 해결한 Hotfix에 'iss53'이 섞이는 것을 방지하기 위해 현재 작업을 커밋하고 master 브랜치로 옮긴다.

```
$ git checkout master
Switched to branch 'master'
```

개발 시나리오 (3/8)

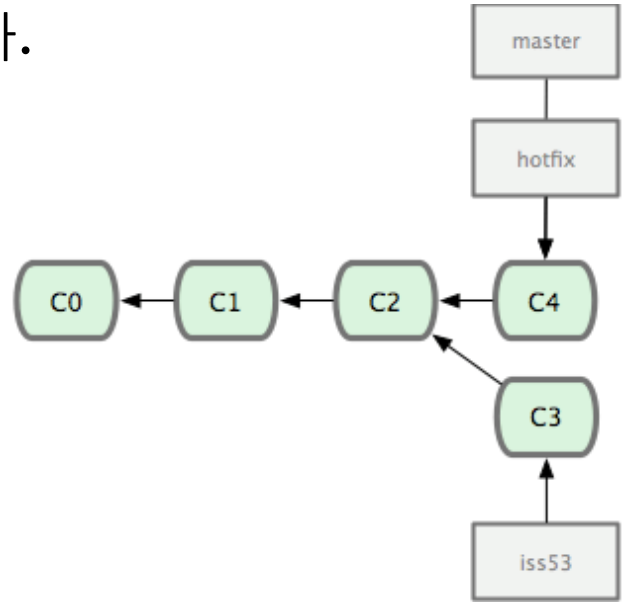
- 개발의 진행 (Cont.)
  - hotfix라는 브랜치를 만들고 새로운 이슈를 해결할 때까지 사용한다.

```
$ git checkout -b hotfix
Switched to a new branch 'hotfix'
$ vim index.html
$ git commit -a -m 'fixed the broken email address'
[hotfix 3a0874c] fixed the broken email address
1 files changed, 1 deletion(-)
```



  - 문제가 해결되면 master 브랜치에 합쳐야 한다.

```
$ git checkout master
$ git merge hotfix
Updating f42c576..3a0874c
Fast-forward
 README | 1 -
 1 file changed, 1 deletion(-)
```



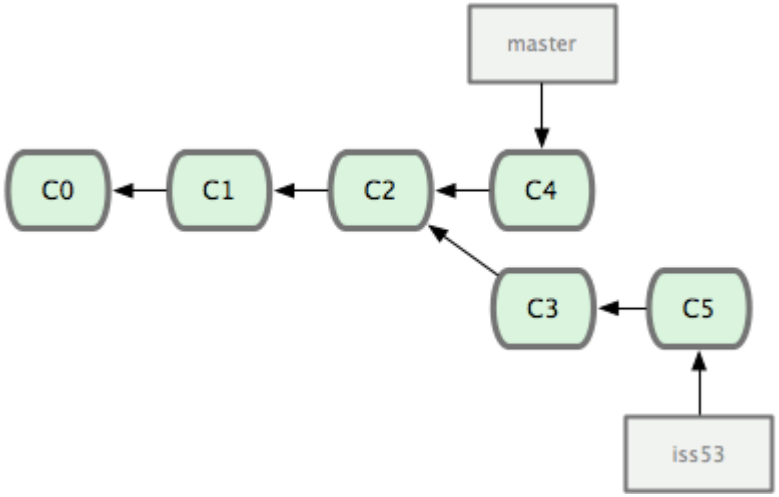
개발 시나리오 (4/8)

- 개발의 진행 (Cont.)
  - 문제를 급히 해결하고 master 브랜치에 적용하고 나면 다시 일하던 브랜치로 돌아가야 한다. 하지만, 그전에 필요없는 hotfix 브랜치를 삭제한다.

```
$ git branch -d hotfix
Deleted branch hotfix (was 3a0874c).
```

  - 이제 이슈 53번을 처리하던 환경으로 되돌아가서 하던 일을 계속 한다.

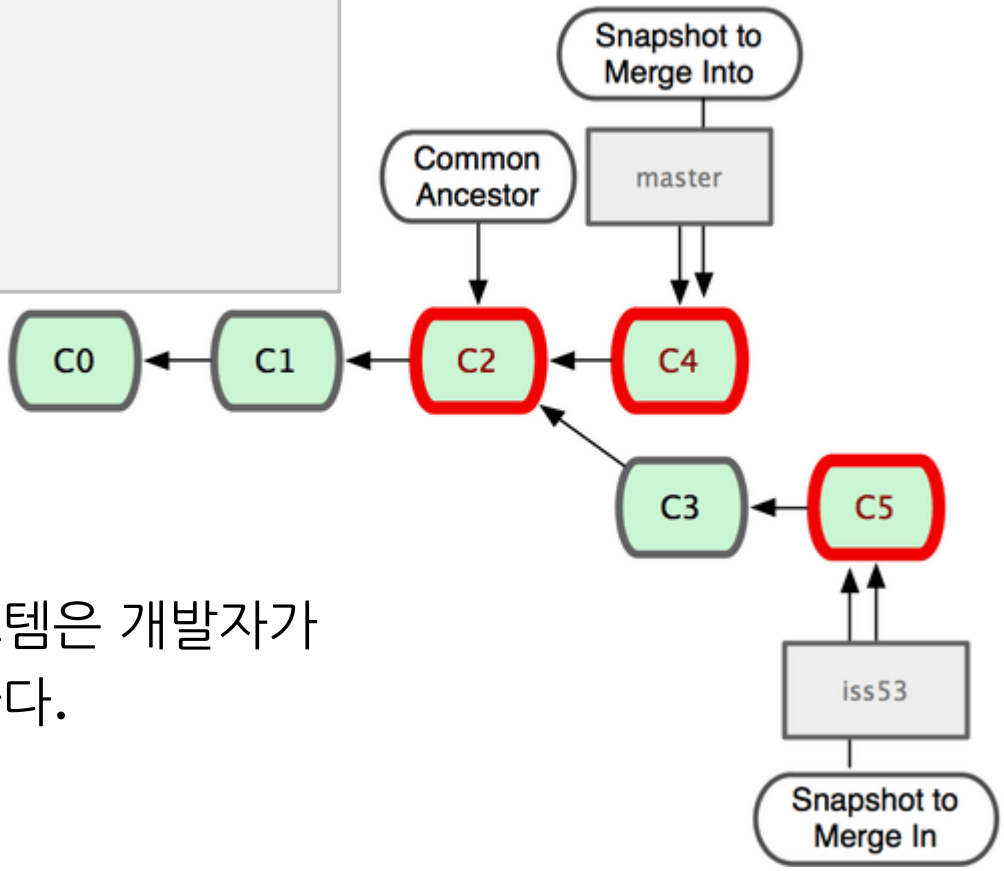
```
$ git checkout iss53
Switched to branch 'iss53'
$ vim index.html
$ git commit -a -m 'finished the new footer [issue 53]'
[iss53 ad82d7a] finished the new footer [issue 53]
 1 file changed, 1 insertion(+)
```



개발 시나리오 (5/8)

- 개발 브랜치의 Merge
  - 53번 이슈를 다 구현하고 master 브랜치에 Merge하는 과정.

```
$ git checkout master
$ git merge iss53
Auto-merging README
Merge made by the 'recursive' strategy.
 README | 1 +
 1 file changed, 1 insertion(+)
```

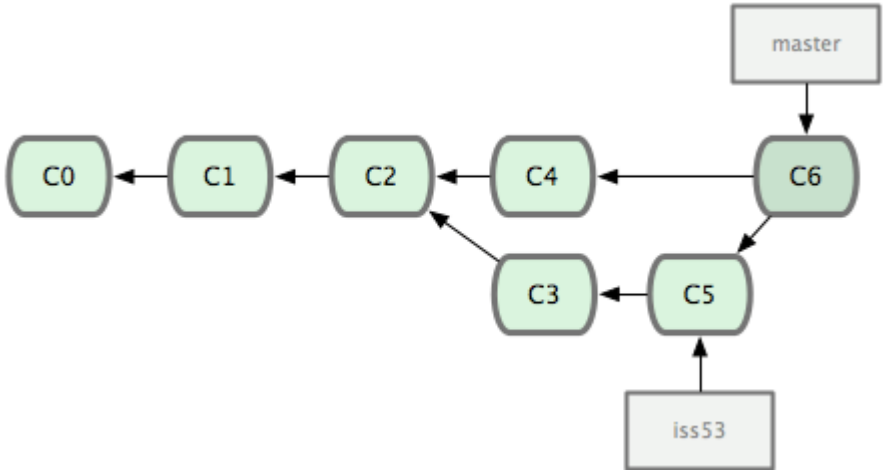


  - Git은 각 브랜치가 가리키는 커밋 두 개와 공통 조상 하나를 사용하여 3-way Merge를 한다.
  - CVS나 Subversion 같은 버전 관리 시스템은 개발자가 직접 공통 조상을 찾아서 Merge해야 한다. Git은 다른 시스템보다 Merge가 쉽다.

개발 시나리오 (6/8)

- Merge가 완료된 브랜치
  - iss53 브랜치를 master에 Merge하고 나면 더는 iss53 브랜치가 필요 없다. 다음 명령으로 브랜치를 삭제하고 이슈의 상태를 처리 완료로 표시한다.

```
$ git branch -d iss53
```



개발 시나리오 (7/8)

- 충돌의 처리
  - 가끔씩 3-way Merge가 실패할 때도 있다. Merge하는 두 브랜치에서 같은 파일의 한 부분을 동시에 수정하고 Merge하면 Git은 해당 부분을 Merge하지 못한다.
  - 예를 들어, 53번 이슈와 hotfix가 같은 부분을 수정했다면 Git은 Merge하지 못하고 다음과 같은 충돌(Conflict) 메시지를 출력한다.

```
$ git merge iss53
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
Automatic merge failed; fix conflicts and then commit the result.
```

  - Merge 충돌이 일어났을 때 Git이 어떤 파일을 Merge할 수 없었는지 살펴보려면 git status 명령을 이용한다.

```
Unmerged paths:
  (use "git add <file>..." to mark resolution)

        both modified:      index.html
```

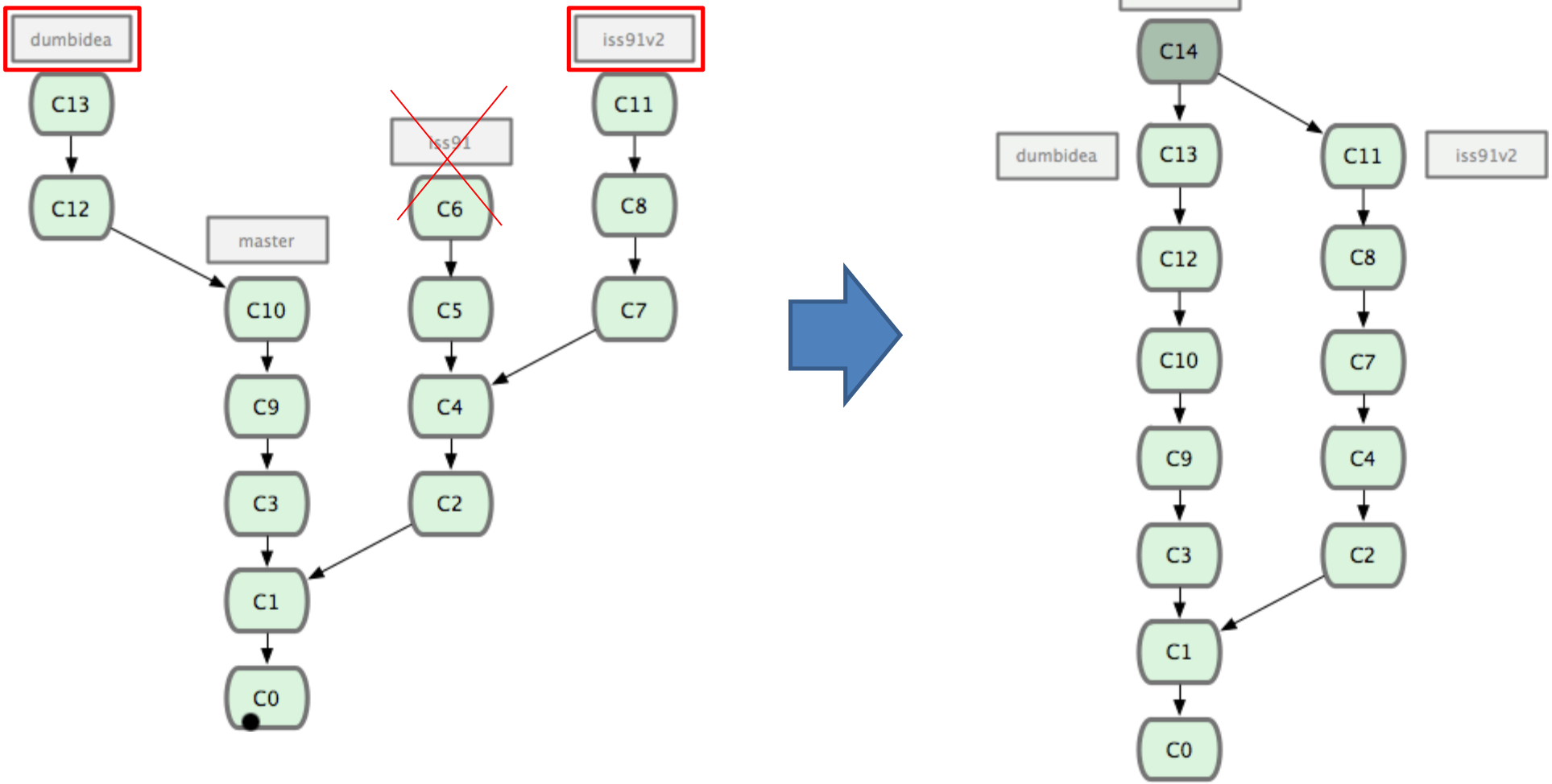
개발 시나리오 (8/8)

- 충돌의 처리 (Cont.)
  - 충돌난 메시지 부분을 참조하여 수동으로 해당 부분을 수정하여야 한다.

```
<<<<<<< HEAD
<div id='footer'>contact : email.support@github.com</div>
=======
<div id='footer'>
  please contact us at support@github.com
</div>
>>>>>>> iss53
```

  - <<<<<<<, =======, >>>>>>> 가 포함된 행을 삭제하고 새로 작성하여 해결하고 git add 명령으로 다시 Git에 저장한다.
  - 충돌을 쉽게 해결하기 위해 git mergetool 명령을 이용할 수 있다.
  - git status 명령으로 충돌이 해결된 상태인지 다시 한번 확인해볼 수 있다.
  - git commit 명령으로 Merge 한 것을 커밋한다.

머지의 또다른 예


dumbidea
C13
C12
master
C10
C9
C3
C1
C0
iss91
C6
C5
C4
C2
iss91v2
C11
C8
C7
master
C14
dumbidea
C13
C12
C10
C9
C3
C1
C0
C11
C8
C7
C4
C2
iss91v2

중앙 저장소를 활용한 공동 작업 시나리오의 사례

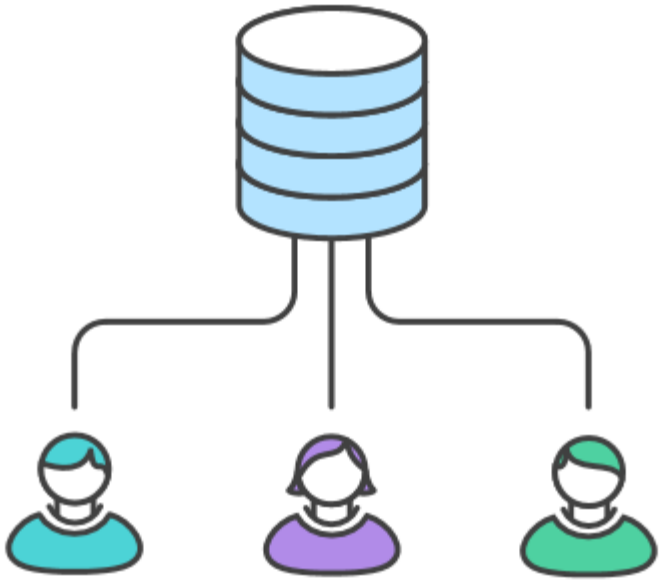
- 철이와 미애, 두 명의 개발자로 구성된 작은 팀이 Centralized Workflow를 이용하여 어떻게 협업하는 지 살펴보자.

- 기존 프로젝트라면 가져오면 되고, 새로운 프로젝트라면 빈 저장소를 만들면 된다.

```
$ ssh user@host git init --bare /path/to/repo.git
```

중앙 저장소를 활용한 공동 작업 시나리오의 사례

- 모든 팀 구성원이 git clone 명령으로 중앙 저장소를 복제해서 로컬 저장소를 만든다.



```
$ git clone ssh://user@host/path/to/repo.git
```

중앙 저장소를 활용한 공동 작업 시나리오의 사례

- 철이의 작업
  - 로컬에서 파일을 add 하고 추적 가능한 상태에서 commit로 스테이지 (임시 공간)에 둔다.

```
$ git status                # 로컬 저장소의 상태 확인
$ git add <some-file>       # 스테이징 영역에 some-file 추가
$ git commit                # some-file의 변경 내역을 커밋
```

중앙 저장소를 활용한 공동 작업 시나리오의 사례

- 미애의 작업
  - 미애도 철이처럼 로컬 저장소를 만들고, 자신이 맡은 기능을 개발하고, 스테이징하고 커밋한다.

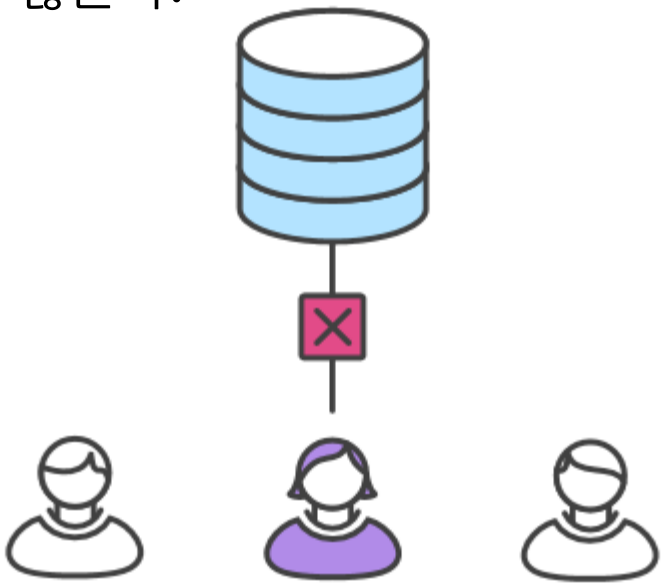
중앙 저장소를 활용한 공동 작업 시나리오의 사례

- 철이의 작업 내용 발행
  - 철이는 **git push** 명령으로 자신의 로컬 커밋 이력을 중앙 저장소에 올려 다른 팀 구성원과 공유하려 한다. 아무도 중앙 저장소를 변경하지 않았기 때문에, 철이의 푸시는 충돌없이 순조롭게 진행될 것이다.

```
$ git push origin master
```

중앙 저장소를 활용한 공동 작업 시나리오의 사례

- 미애의 작업 내용 발행
  - 미애의 커밋 이력은 중앙 저장소의 최신 커밋 이력을 포함하고 있지 않아(diverge), 미애의 푸시를 받아 주지 않는다.

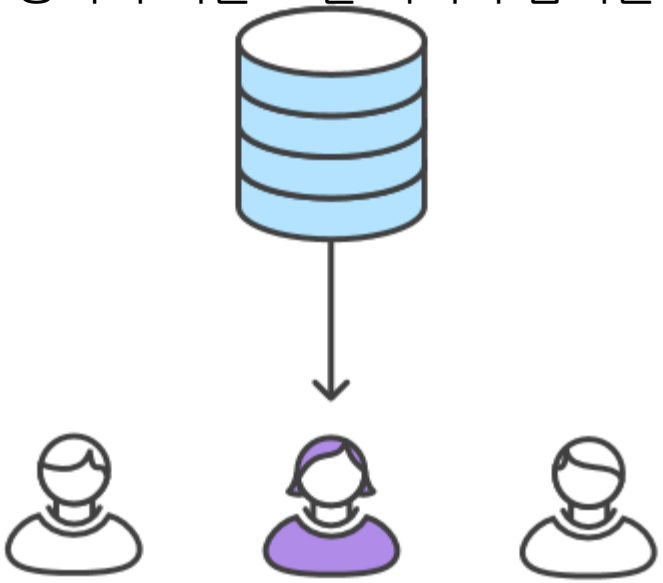


```
$ git push origin master
```

  - 미애는 철이의 커밋 이력을 로컬로 받아온 후, 자신의 로컬 커밋 이력과 통합한 후, 다시 푸시해야 한다.

중앙 저장소를 활용한 공동 작업 시나리오의 사례

- 미애의 리베이스
  - git pull 명령으로 중앙 저장소의 변경 이력을 로컬 저장소로 내려 받는다. 이 명령은 중앙 저장소의 최신 이력을 내려 받는 동작과 이를 로컬 이력과 합치는 동작을 한 번에 한다.

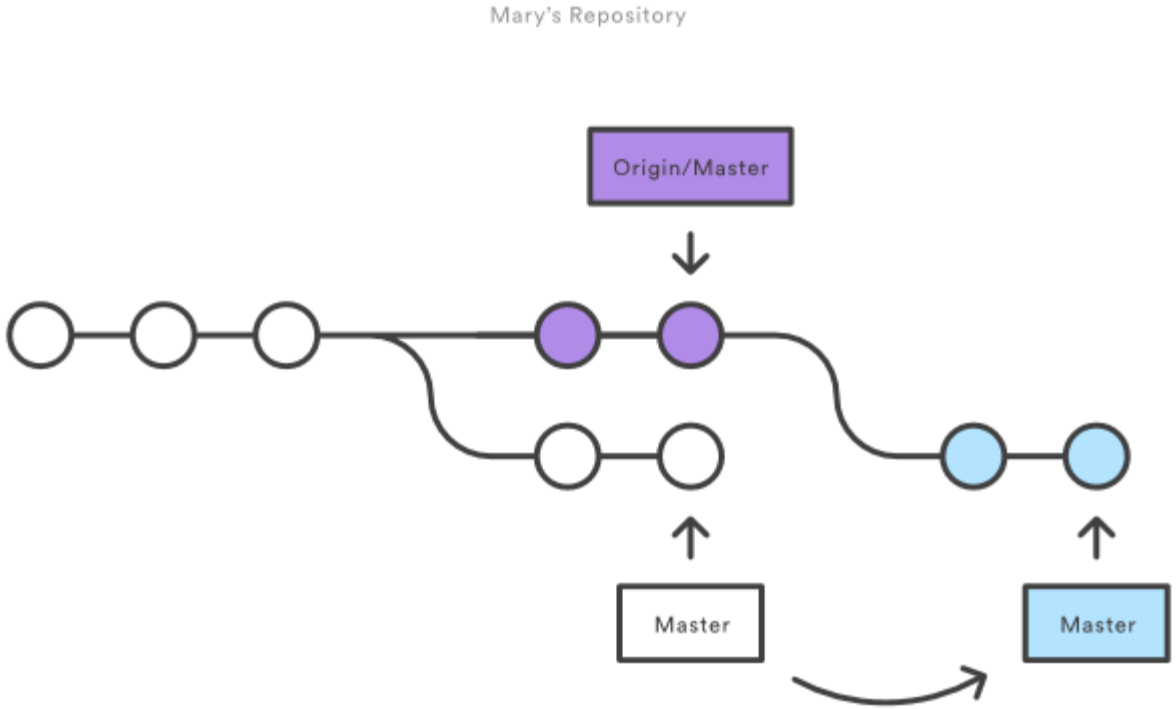


```
$ git pull --rebase origin master
```

--rebase 옵션을 주면 중앙 저장소의 커밋 이력을 미애의 커밋 이력 앞에 끼워 넣는다.

중앙 저장소를 활용한 공동 작업 시나리오의 사례

- --rebase 옵션 없이 쓸 수도 있지만, 불필요한 병합 커밋을 한 번 더해야 하는 번거로움이 있으므로 --rebase 옵션을 쓰는 것이 좋다.



중앙 저장소를 활용한 공동 작업 시나리오의 사례

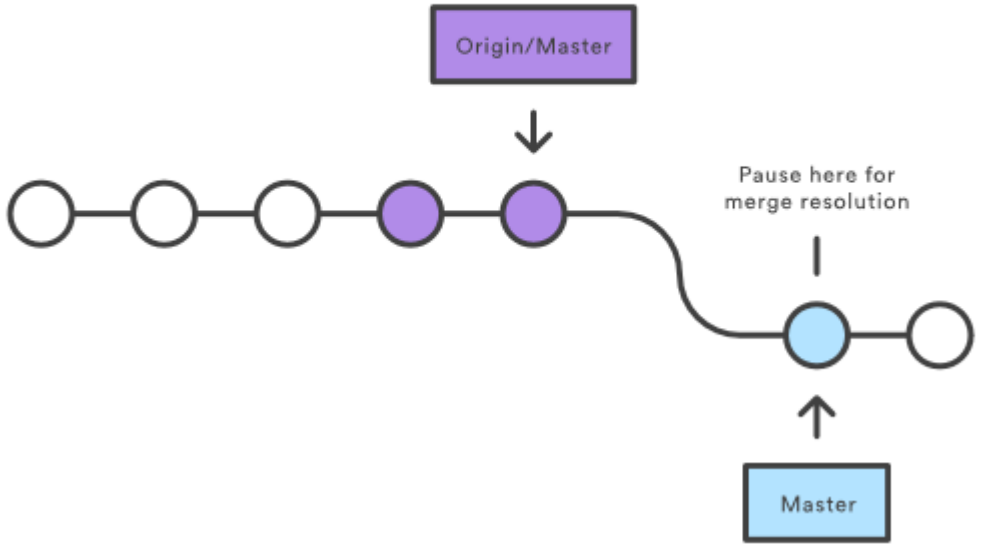
- 미애의 충돌 해결
  - 리베이스는 미애의 로컬 커밋을 새로 내려 받은 master 브랜치에 하나 하나 대입하고 대조해 가면서 커밋 이력을 재배열한다.

  - 이런 동작 특성때문에 커밋 이력도 깔끔하게 유지할 수 있을 뿐만아니라, 경우에 따라 버그를 발견하기도 한다.

중앙 저장소를 활용한 공동 작업 시나리오의 사례

- 리베이스 과정에 충돌이 발생하면, Git은 현재 커밋에서 리베이스를 멈추고 다음과 같은 메시지를 뿜어 낸다.

```
# CONFLICT (content): Merge conflict in <some-file>
```



- 미애는 git status 명령으로, Unmerged paths: 부분에서 충돌이 발생한 파일을 찾을 수 있다.

중앙 저장소를 활용한 공동 작업 시나리오의 사례

```
$ git status
# Unmerged paths:
# (use "git reset HEAD <some-file>..." to unstage)
# (use "git add/rm <some-file>..." as appropriate to mark resolution)
#
# both modified: <some-file>
```

- 이제 some-file을 열어 충돌을 해결하고, 스테이징 영역에 변경된 파일을 추가한 후, 리베이스를 계속 하면 된다.

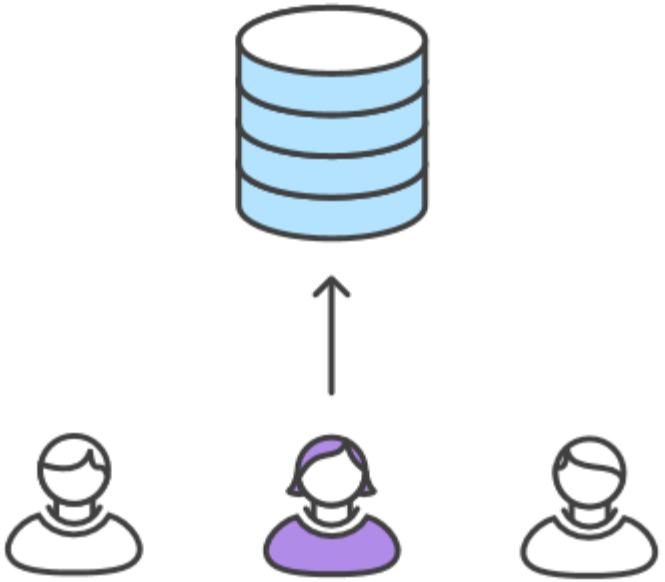
```
$ git add <some-file>
$ git rebase --continue
```

- 리베이스는 다음 커밋으로 넘어가고, 더 이상 충돌이 없다면 리베이스는 성공적으로 끝난다. 리베이스 중에 뭔가 잘못되었다면, 다음 명령으로 git pull --rebase 명령을 내리기 이전 상태로 되돌릴 수 있다.

```
$ git rebase --abort
```

중앙 저장소를 활용한 공동 작업 시나리오의 사례

- 미애의 작업 내용 재발행
  - 중앙 저장소의 커밋 이력과 로컬 커밋 이력을 모두 합쳤으므로, 이제 중앙 저장소에 올리기만 하면 된다.



```
$ git push origin master
```

Ref. http://blog.appkr.kr/learn-n-think/comparing-workflows/

Git 서버의 구축 (1/4)

- 사용 프로토콜
  - Local, SSH, Git, HTTP의 프로토콜을 사용할 수 있다.

- 로컬 프로토콜
  - 공유된 디렉터리에서 clone하여 사용할 수 있다.
    - git clone /opt/git/project.git 또는 → 하드링크를 생성하는 방식
    - git clone file:///opt/git/project.git → 별도 프로세스 생성 방식

- SSH 프로토콜
  - SSH를 통해 접근하면 모든 데이터는 암호화되어 인증된 상태로 전송되므로 안전하다. 마지막으로 전송 시 데이터를 가능한 압축하기 때문에 효율적이다.
    - git clone ssh://user@server/project.git 또는
    - git clone user@server:project.git

Git 서버의 구축 (2/4)

- Git 프로토콜
  - Git 프로토콜은 Git에 포함된 데몬을 사용하는 방법이다. 포트는 9418이며 SSH 프로토콜과 비슷한 서비스를 제공하지만, 인증 메커니즘이 없다.
  - 이 저장소는 누구나 Clone할 수 있거나 아무도 Clone할 수 없거나 둘 중의 하나만 선택할 수 있다. 일반적으로 SSH 프로토콜과 함께 사용한다. 소수의 개발자만 Push할 수 있고 대다수 사람은 git://을 사용하여 읽을 수만 있게 한다.

- HTTP/S 프로토콜
  - 설정이 간단하다. HTTP 도큐먼트 루트 밑에 Bare 저장소를 두고 post-update 훅을 설정하는 것이 기본적으로 해야 하는 일의 전부다

Git 서버의 구축 (3/4)

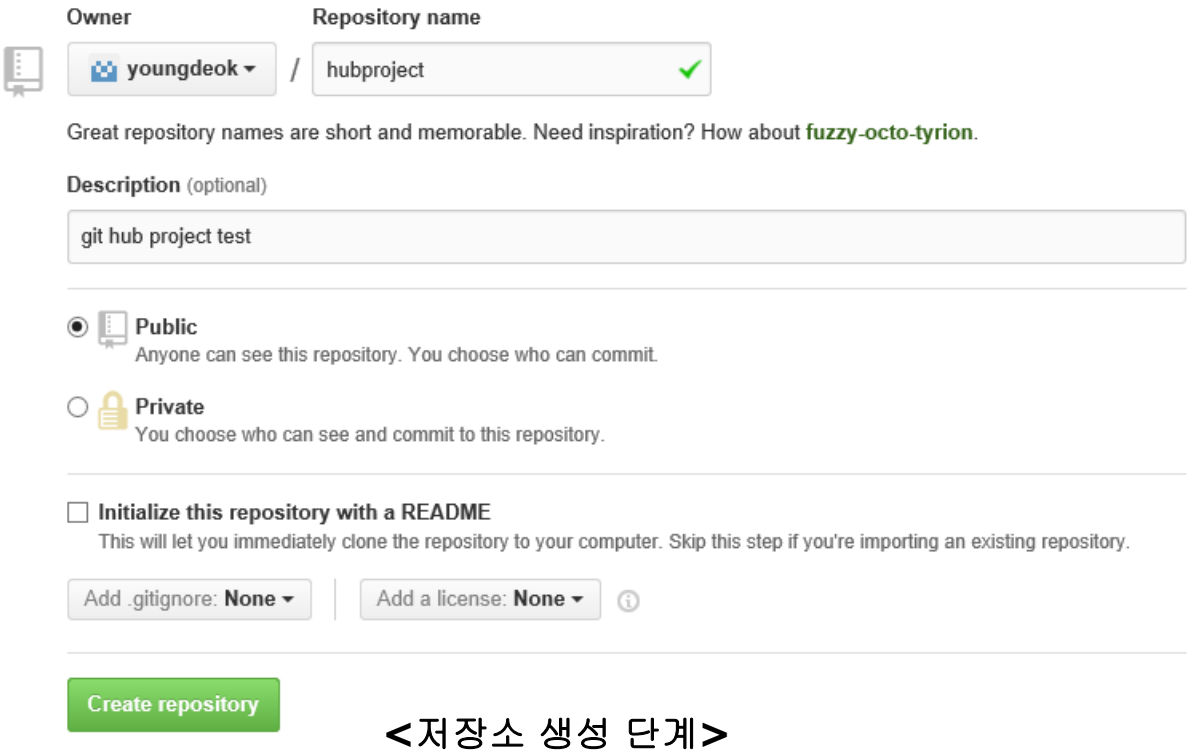
- Git 저장소를 서버에 넣기
  - SSH를 이용하여 서버 디렉터리 /opt/git에 저장소를 만든다.
    - git clone --bare gittest gittest.git
    - mkdir /opt/git; chmod 777 /opt/git
    - sudo scp -r gittest.git edu@example.com:/opt/git
  - 이제 다른 사용자들은 SSH로 서버에 접근해서 저장소를 Clone할 수 있다.
    - su kildong
    - cd ~
    - git clone kildong@example.com:/opt/git/gittest.git
  - 이 서버에 SSH로 접근할 수 있는 사용자가 /opt/git/gittest.git 디렉터리에 쓰기 권한까지 가지고 있으면 바로 Push할 수 있다.

Git 서버의 구축 (4/4)

- Git 저장소를 서버에 넣기 (Cont.)
  - git init 명령에 --shared 옵션을 추가하면 Git은 자동으로 그룹 쓰기 권한을 추가한다.
    - ssh edu@example.com
    - cd /opt/git/gittest.git
    - git init --bare -- shared
  - Git 서버를 구축하는데 할 일은 별로 없다. SSH로 접속할 수 있도록 서버에 계정을 만들고 Bare 저장소를 사람들이 읽고 쓸 수 있는 곳에 넣어 두기만 하면 된다. 다른 것은 아무것도 필요 없다.

GitHub 호스팅 업체 사용 (1/3)

- GitHub
  - GitHub은 가장 큰 오픈소스 Git 호스팅 사이트이고 공개(Public) 프로젝트와 비공개(Private) 프로젝트에 대한 호스팅 서비스를 제공한다.
  - https://github.com/



<저장소 생성 단계>

GitHub 호스팅 업체 사용 (2/3)

- GitHub의 프로젝트 등록
  - 작업하던 gittest 프로젝트를 등록해 보자.
    - su edu
    - cd ~/gittest
    - git **remote add** origin https://github.com/youngdeok/hubproject.git
    - git **push** origin master → 원격 저장소로 전송

```
edu@mail:~/gittest$ git push origin master
Username for 'https://github.com': youngdeok
Password for 'https://youngdeok@github.com':
Counting objects: 22, done.
Compressing objects: 100% (14/14), done.
Writing objects: 100% (22/22), 1.95 KiB | 0 bytes/s, done.
Total 22 (delta 2), reused 0 (delta 0)
To https://github.com/youngdeok/hubproject.git
 * [new branch]      master -> master
```

  - 이제 프로젝트가 GitHub에서 서비스되니 공유하고 싶은 사람에게 URL을 알려준다.
    - git clone git@github.com:youngdeok/hubproject.git

GitHub 호스팅 업체 사용 (3/3)

- GitHub의 원격 작업
  - 파일을 수정하고 해당 내용을 원격 저장소에 반영해 보자.
    - vim hello2.c
    - git add hello2.c
    - git commit -m 'modified hello2.c'
    - git push origin master
  - 이후 GitHub로 부터 반영 내용을 확인할 수 있다.
  - 공동 작업으로 특정 내용을 받아 오려면 git fetch 명령을 이용한다.
    - git remote add bufFix https://github.com/youngdeok/hubproject.git
    - giv remote -v
    - git fetch bufFix → 바뀐 부분을 bufFix로 받아온다.
    - git branch -va → bufFix는 remote를 가리킨다는 것을 알 수 있다.
    - git merge bufFix/master → bufFix의 커밋을 합쳐 반영한다.
    - git push origin master

GitHub의 요약

- GitHub의 장 단점
  - 누구나 손쉽게 무료 계정을 만들어 오픈소스 프로젝트를 시작할 수 있다.
  - 원하는 프로젝트를 탐색하다가 Fork하여 자신의 프로젝트로 가져와 마음대로 수정할 수 있다.
  - GitHub에 있는 개발자 커뮤니티 규모는 매우 크기 때문에 만약 GitHub에 오픈 소스 프로젝트를 만들면 다른 개발자들이 당신의 프로젝트를 복제하고 당신을 도울 것이다.
  - GitHub은 이윤을 목적으로 하는 회사이기 때문에 비공개 저장소를 만들려면 돈을 내야 한다.

- Bitbucket 서비스
  - https://bitbucket.org/
  - 무료로 비공개 저장소를 만들 수 있다.
  - 1개의 그룹에 협업하는 콜라보레이터 수가 기본 5명으로 제한
  - GitHub, Google code, SVN에서 소스를 가지고 올수도 있다.

bitbucket 저장소 생성 순서

- 생성 순서
  - 1.bitbucket에 접속한다.
  - 2.remote repository를 만든다
  - 3.local repository를 만들면서 clone한다.

로컬 디렉터리 설정

- 최초의 Git 저장소를 설정한다.

```
mkdir /work/git/helloworld
cd /work/git/helloworld
git init
git remote add origin https://medusakiller@bitbucket.org/medusakiller/helloworld.git
```

- 파일을 생성하고 커밋, 푸시 한다.

```
echo "Youngdeok Hwang" >> contributors.txt
git add contributors.txt
git commit -m 'Initial commit with contributors'
git push -u origin master
```

Q/A