성신여대 오픈소스 팀 프로젝트 실습

Summary: Web 페이지를 프로젝트화 하여 서로 협업하면서 작업을 진행, 웹 프로젝트 소스를 서로 수정함으로 발생되는 문제를 강사와 함께 해결한다.

목표

- Webpage 프로젝트를 서로 협업하면서 작업을 진행.
- Markdown으로 구성된 페이지 수정 작업을 통해 git/github 사용법을 완벽 숙지한다.

PDF 문서

- 다운로드 클릭 (https://github.com/oss-sungshin/git-lecture/raw/master/pdf/mydoc.pdf)

사용 언어

- 개발 프로젝트가 아닌 관계로 모든 작업은 Markdown을 기반으로 작성.
- Markdown 사용방법은 아래 사이트를 참고하세요.

- https://guides.github.com/features/mastering-markdown (https://guides.github.com/features/mastering-markdown)
- Markdown viwer 활용 https://jbt.github.io/markdown-editor/ (https://jbt.github.io/markdown-editor/)

실습 진행 방법

- 모든 실습자들은 웹 프로젝트(https://github.com/oss-sungshin/git-lecture (https://github.com/oss-sungshin/git-lecture))의 소스를 수정하면서 발생되는 문제를 2명의 강사와 함께 해결한다.
- 실습 진행 도중 여러 이벤트(브랜치 작업, 개발 도중 머지) 들의 미션이 발생되며, 강사와 함께 문제점을 같이 해결한다.

Git 내용 참고 사이트

- https://git-scm.com/book/ko/v1 (https://git-scm.com/book/ko/v1)

2월 9일(목)시간표

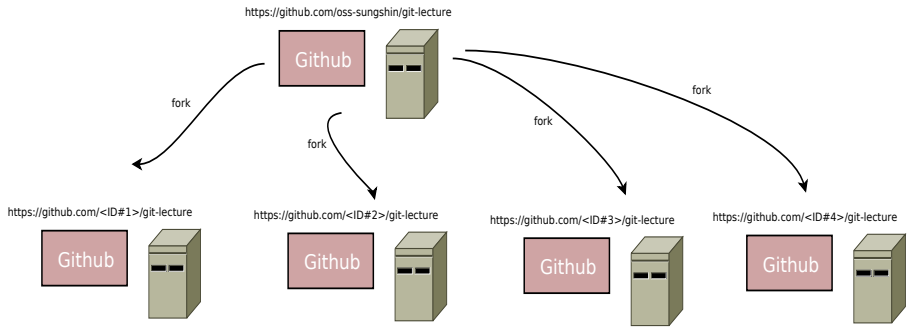
- 10:00 ~ 11:50 : 실습 진행 방법 설명, 팀 빌딩, 실습 진행
- 11:00 : 브랜치 후 작업 진행 (강사 개별 지도)
- Event #1(11:20) : 1차 머지(팀원 push -> 팀장 pull request -> 서버)
    - 원격 저장소(oss-sungshin)를 등록한다.
    - 개별 브랜치 및 머지 충돌을 실습한다.
    - 팀장은 팀 저장소를 업데이트 한다.
- Event #2(15:10) : 2차 머지(팀원 push -> 팀장 pull request -> 서버)
    - 개별 브랜치 및 머지 충돌을 실습한다.
    - gitk를 통해 커밋 로그를 분석한다.
- Event #3(16:00) : 커밋 가져오기
    - 최근 oss-sungshin 커밋을 팀장 github 저장소에 적용하기(git cherry-pick)
- Event #4(16:30) : 팀별 커밋 내용 공유 및 설명(강사 진행)
    - 내용 공유

2월 10일(금) 시간표

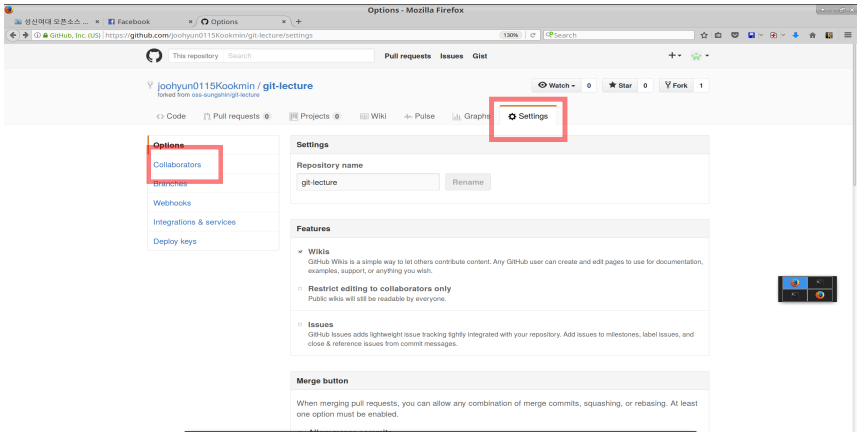
- 13:00 ~ 15:50 : 실습 진행 및 시상
- Event #5(13:20) : 4차 머지(팀원 push -> 팀장 pull request -> 서버)
    - 머지 문제 해결
- Event #5(14:00) : reset 실습
    1. 현재 커밋 백업 (git format-patch -2)
    2. 과거 커밋으로 이동 (git reset COMMIT_ID)
    3. 백업 커밋 적용 (git am 이용)
- Event #5(15:00) : 최종 머지 및 작업 내용 발표
    - 작업 한 내용 서로 공유
- Event #5(15:30) : 종합 및 시상식

팀장이 해야 할 일

- 자신의 github에 oss-sungshin 프로젝트 fork 하기



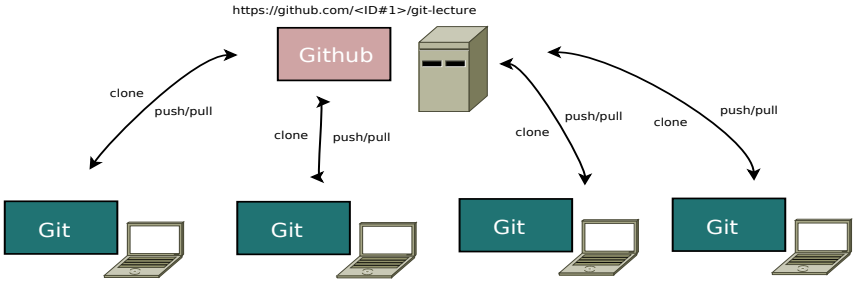
- 팀원 github 계정을 프로젝트에 등록하기



- 통합은 pull request를 사용하여 병합 할 예정

팀원이 해야 할 일

- 팀장의 github 프로젝트를 local PC에 clone 하기



팀별 결과 공유 및 시상

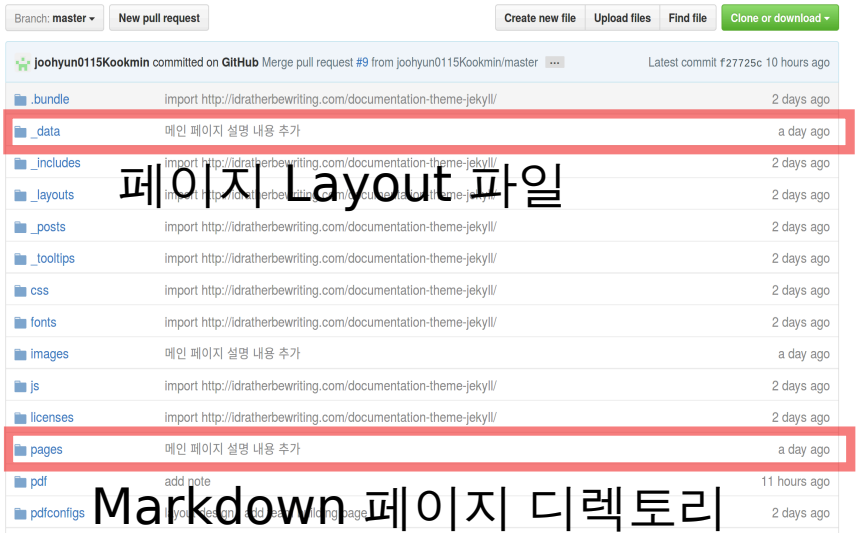
최종 결과물은 아래 페이지에 영구 보존되며, 우수 팀은 팀별 경품이 있습니다.

소스 트리 구조

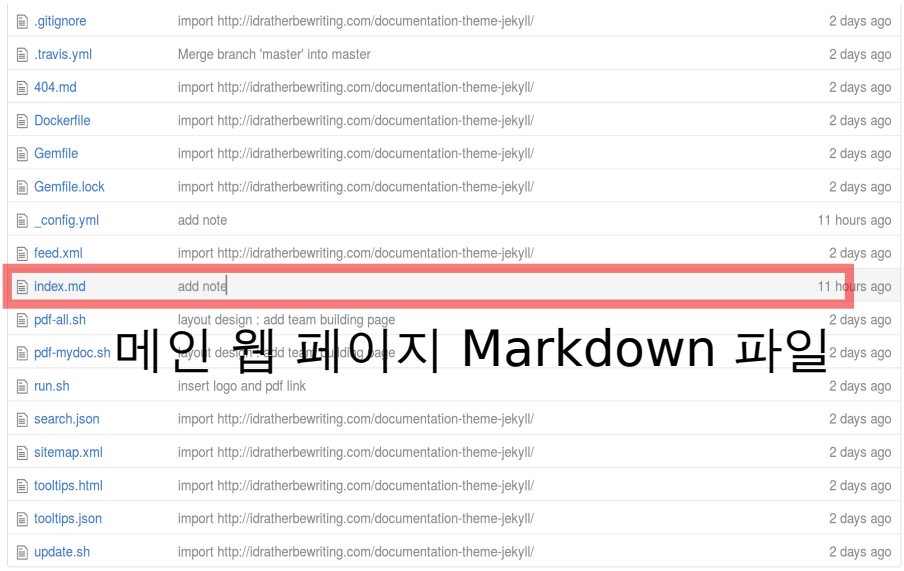
Summary: 문서 소스 트리 구조에 대한 설명

소스 트리 구조

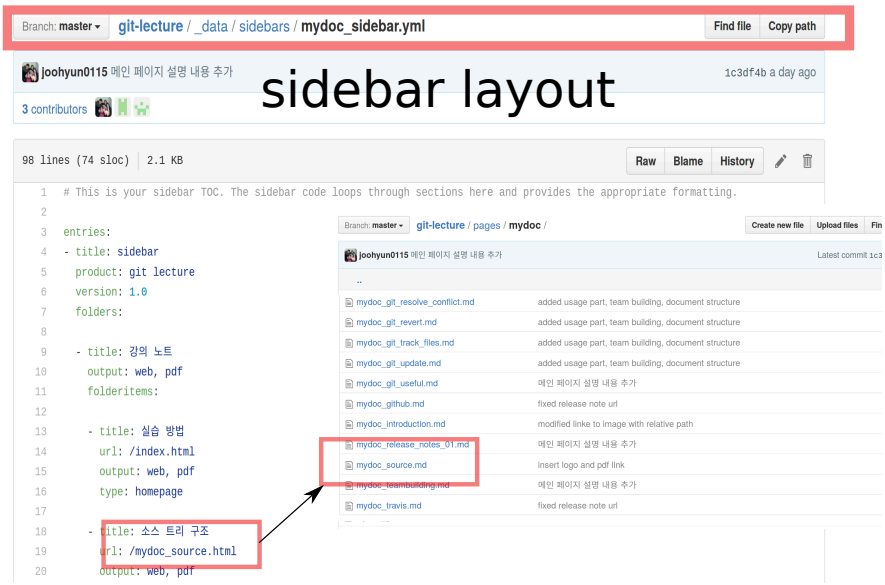
- _data 폴더는 side바 메뉴 설정 시 사용 & pages 폴더는 페이지 생성 및 수정



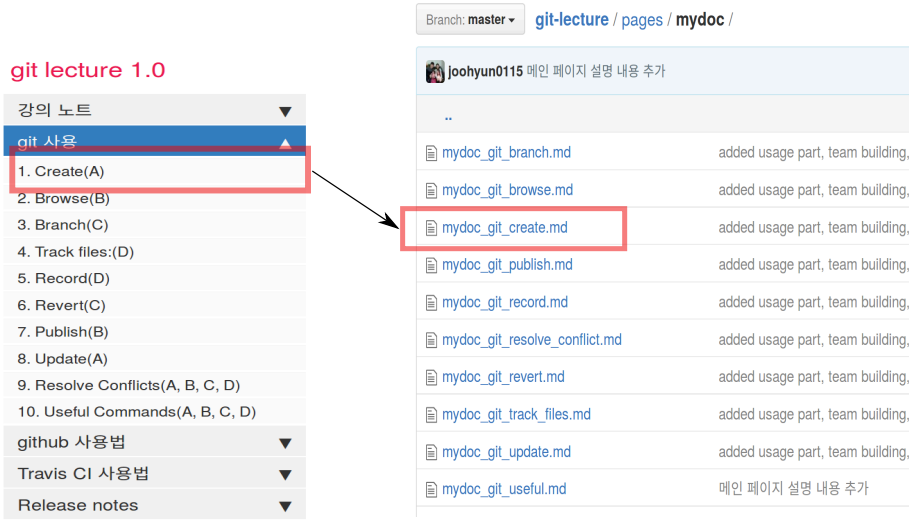
- 메인 웹페이지 Markdown 페이지 index.md



- 사이드바 메뉴는 _data/sidebars/mydoc_sidebar.yml에서 설정



- 메뉴와 markdown 파일은 1:1 매핑 관계임


git lecture 1.0
강의 노트
git 사용
1. Create(A)
2. Browse(B)
3. Branch(C)
4. Track files:(D)
5. Record(D)
6. Revert(C)
7. Publish(B)
8. Update(A)
9. Resolve Conflicts(A, B, C, D)
10. Useful Commands(A, B, C, D)
github 사용법
Travis CI 사용법
Release notes
Branch: master
git-lecture / pages / mydoc /
joohyun0115 메인 페이지 설명 내용 추가
..
mydoc_git_branch.md added usage part, team building,
mydoc_git_browse.md added usage part, team building,
mydoc_git_create.md added usage part, team building,
mydoc_git_publish.md added usage part, team building,
mydoc_git_record.md added usage part, team building,
mydoc_git_resolve_conflict.md added usage part, team building,
mydoc_git_revert.md added usage part, team building,
mydoc_git_track_files.md added usage part, team building,
mydoc_git_update.md added usage part, team building,
mydoc_git_useful.md 메인 페이지 설명 내용 추가

팀 빌딩

Overview

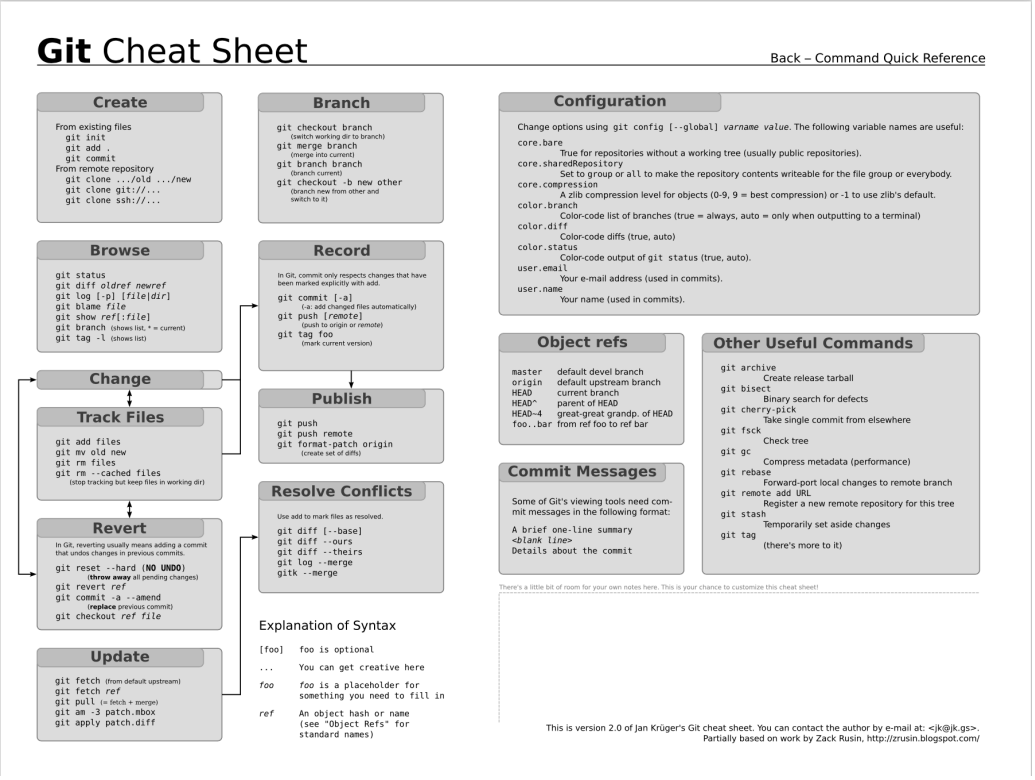
실습 수업 진행을 위한 팀 빌딩: 한 조당 5명, 5분동안 정하세요.

team 빌딩 결과

팀명	조원이름
A	김경민, 이주영, 조서원, 이수진, 이효은
B	이정희, 이한나, 유조영, 최진희, 김은채

문서 구조

참고 자료



사진이 잘 보이지 않는 경우 마우스 오른쪽 버튼을 이용해서 사진을 저장해서 윈도우즈 사진 뷰어를 통해 열어서 보시면 됩니다.

문서구조

NOTE!: 기본 명령어 및 옵션에 대한 사용법을 예제와 함께 작성해주시면 됩니다. 필요에 따라 검색을 하셔도 좋습니다.

1. Create (page 11): 실제 수정해야할 파일: $(GIT_LECTURE)/pages/mydoc/mydoc_git_create.md

2. Browse (page 13): 실제 수정해야할 파일: $(GIT_LECTURE)/pages/mydoc/mydoc_git_browse.md

3. Branch (page 14): 실제 수정해야할 파일: $(GIT_LECTURE)/pages/mydoc/mydoc_git_branch.md

4. Track_files (page 17): 실제 수정해야할 파일: $(GIT_LECTURE)/pages/mydoc/mydoc_git_track_files.md

5. Record (page 18): 실제 수정해야할 파일: $(GIT_LECTURE)/pages/mydoc/mydoc_git_record.md

6. Revert (page 19): 실제 수정해야할 파일: $(GIT_LECTURE)/pages/mydoc/mydoc_git_revert.md

7. Publish (page 22): 실제 수정해야할 파일: $(GIT_LECTURE)/pages/mydoc/mydoc_git_publish.md

8. Update (page 23): 실제 수정해야할 파일: $(GIT_LECTURE)/pages/mydoc/mydoc_git_update.md

9. Resolve_conflict (page 26): 실제 수정해야할 파일: $(GIT_LECTURE)/pages/mydoc/mydoc_git_resolve_conflict.md

10. Useful_command (page 46): 실제 수정해야할 파일: $(GIT_LECTURE)/pages/mydoc/mydoc_git_useful.md

Create

Summary: Writing git manual

Create

Image of branch

```
From existing files
   git init
   git add .
   git commit

From remote repository
   git clone .../old .../new
   git clone git://...
   git clone ssh://...
```

- git init - Create a Git repository

이 명령은 .git이라는 하위 디렉토리를 만든다.

.git 디렉토리에는 저장소에 필요한 뼈대 파일(Skeleton)이 들어 있다.

- git add . - to add files to Git

Git이 파일을 관리하게 하려면 저장소에 파일을 추가하고 커밋해야 한다.

git add 명령으로 파일을 추가하고 커밋한다:

- git commit - to commit those files to your local copy

Git은 생성하거나 수정하고 나서 git add 명령으로 추가하지 않은 파일은 커밋하지 않는다.

그 파일은 여전히 Modified 상태로 남아 있다.

커밋하기 전에 git status 명령으로 모든 것이 Staged 상태인지 확인할 수 있다. 그리고 git commit을 실행하여 커밋한다:

- git clone - to copy an existing repository with git clone

github에서 소스를 최초로 내려받을때, git clone명령어를 사용하면 된다.

이는 저장소를 복제한다는 뜻이다.

git browse-related command

Summary: Writing git manual

git browse

- git status 파일의 상태 확인하기 파일의 상태를 확인하려면 보통 git status 명령을 사용한다. Clone 한 후에 바로 이 명령을 실행하면 아래과 같은 메시지를 볼 수 있다.

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working directory clean
```

yoyo yoyo

- git diff oldref newref
- git log
- git blame file
- git show ref
- git branch
- git tag

git branch-related command

Summary: Writing git manual

##Branch

코드를 통째로 복사하고 나서 원래 코드와는 상관 없이 독립적으로 개발을 진행하는 것이다.

```
git checkout branch
  [switch working dir to branch]
```

```
git merge branch
  [merge into current]
```

```
git branch branch
  [branch current]
```

```
git checkout -b new other
  [branch new from other and switch to it]
```

브랜치

브랜치란 커밋 사이를 가볍게 이동할 수 있는 어떤 포인터 같은 것이다.

최초로 커밋하면 Git은 master라는 이름의 브랜치를 만들어서 자동으로 가장 마지막 커밋을 가리키게 한다.

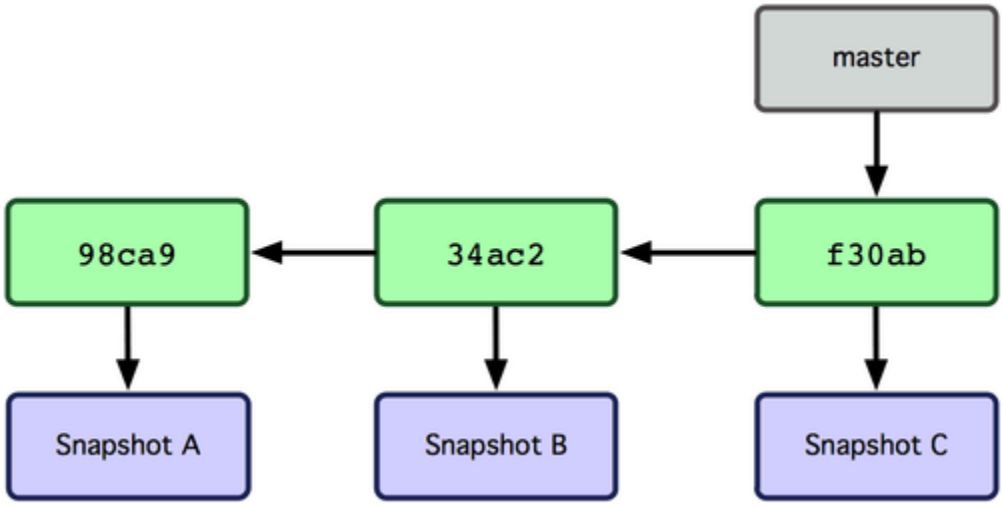


그림 1. 가장 최근 커밋 정보를 가리키는 브랜치

이해를 돕기 위해 testing 브랜치를 만든다.

```
$ git branch testing
```

위의 새로 만든 브랜치도 지금 작업하고 있던 마지막 커밋을 가리킨다.

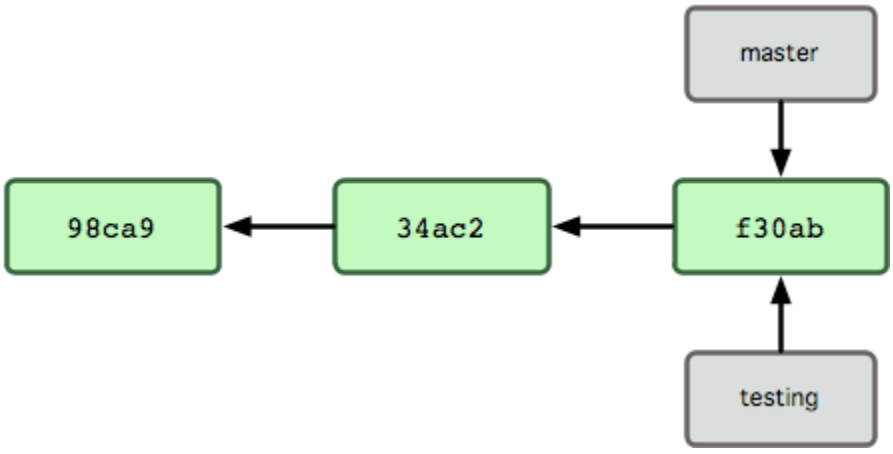


그림 2. 커밋 개체를 가리키는 두 브랜치

Git은 Head라는 특수한 포인터로 지금 작업하는 로컬 브랜치를 가리킨다. git branch 명령은 브랜치를 만들기만 하고 브랜치를 옮기지 않는다. git checkout 명령으로 새로 만든 브랜치로 이동할 수 있다.

```
$ git checkout testing
```

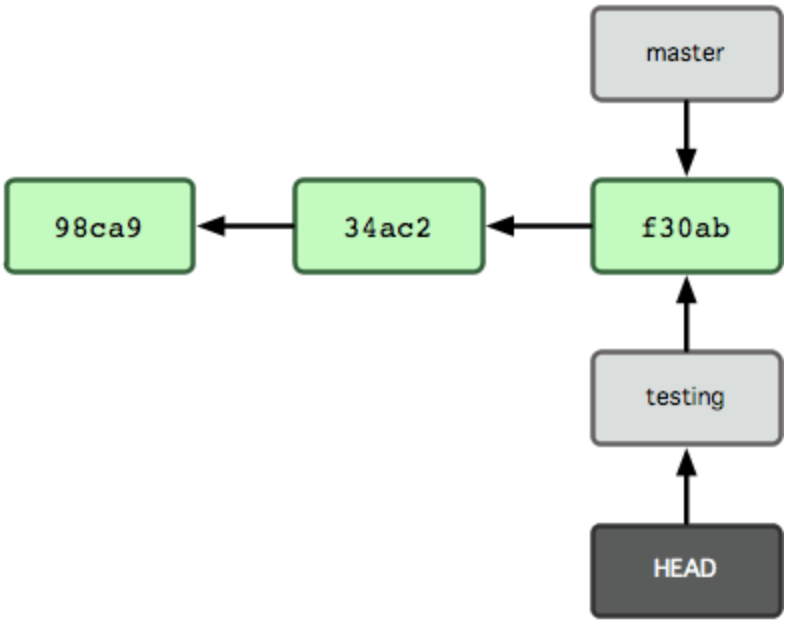


그림 3. HEAD는 옮겨간 다른 브랜치를 가리킨다.

git track-related command

Summary: Writing git manual

- 작업폴더의 파일을 깃이 추적하게 하거나 커밋을 위한 준비상태로 만드는 명령어
    - git add files
    - git add .
    - git add -i (git 대화모드로 파일 staging)
- 파일명 수정
    - git mv old_name new_name
- 파일 또는 폴더를 삭제
    - git rm files
- 깃의 추적 중단시키기 (파일은 남기고 깃디렉토리에서만 삭제)
    - git rm –cached files

Record

Summary: Git record-related command

Git record-related command

- In Git, commit only respects changes that have been marked explicitly with add.

```
git commit [-a]
git push [romote]
git tag foo
```

- git add 명령으로 파일들을 커밋할 목록에 추가하면 git commit 명령으로 커밋(히스토리의 한단위)을 만든다.
    - git commit -a 옵션을 이용하면 바뀐 파일을 자동으로 추가할 수 있다.
- git push 명령으로 Github에 밀어 넣는다.
- git tag 명령으로 이미 만들어진 태그가 있는지 확인할 수 있다. 알파벳 순서로 태그를 보여준다.

Git 상태확인 명령어

```
git show
git log
git shortlog
git diff
git status
```

git revert-related command

Summary: Writing git manual

되돌리기

1. add를 잘못한 경우

stage 영역에 추가하고 싶지 않은 파일을 잘못 추가했다면 git reset을 이용한다.

```
$ git reset 파일이름
```

2. commit을 잘못한 경우

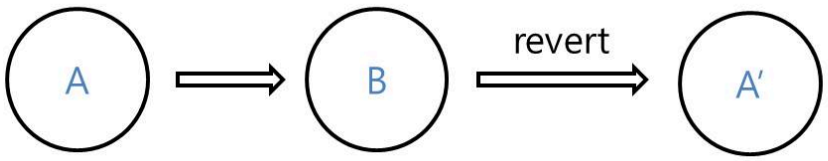
다시 커밋하고 싶으면 –amend 옵션을 사용한다.

```
$ git commit --amend
```

이는 이전 commit을 현재 stage영역의 상태로 덮어쓴다. 따라서 방금 한 commit을 다시 하고 싶으면, 파일을 수정한 뒤 add로 stage에 올리고 commit –amend를 해주면 된다.

amend 외에도 revert 명령어를 이용할 수 있다.

```
$ git revert HEAD
```



commit –amend와 다른점은 revert는 commit을 삭제한 이력을 다른 commit으로 남긴다는 점이다. 이미 push를 했다면 amend나 reset을 사용하기 어렵기 때문에 revert를 해야할 수 밖에 없는 상황도 있다. 만약 하나 이상의 commit을 revert하고 싶다면 merge하는 것과 비슷하게 conflict를 해결해줘야한다.

3. commit을 많이 잘못했다 or 수정을 하기 전으로 되돌리고 싶다.

어디서부터 잘못했는지 모르겠다. 적어도 내가 아는데부터 다시 시작하고 싶다. 또는 수정사항을 만들기 전으로 복원시키고 싶다. 이 경우에는 git reset의 다른 형태를 사용하면 된다.

```
$ git reset [option] commitId
```

reset은 사용자의 저장소와 작업 디렉토리를 특정 시점 상태로 변경한다. 메카니즘을 들여다보면 HEAD 참조를 지정된 commit 시점으로 변경한 후, 인덱스를 변경하여 해당 commit을 반영한다. 옵션에 따라 HEAD, 인덱스 뿐만아니라 작업 디렉토리 내의 모든 파일이 변경될 수 있기때문에 반드시 주의하여 사용하여야 한다.

주요 옵션

- –soft

  git reset –soft [commit id] –soft 옵션은 지정된 commit을 가리키도록 HEAD 참조를 변경한다. 인덱스와 작업 디렉토리의 내용은 그대로 유지된다. 이 옵션은 새 커밋을 가리키도록 심볼릭 참조의 상태만을 변경한다.

- –mixed

  git reset –mixed [commit id] –mixed 옵션은 지정된 commit을 가리키도록 HEAD를 변경한다. 따라서 commit에 맞게 인덱스의 내용도 변경된다. reset의 기본옵션이다.

- –hard

  git reset –hard [commit id] –hard 옵션은 HEAD, 인덱스 뿐만아니라 작업 디렉토리까지 commit과 같은 상태로 변경한다. 굉장히 위험한 옵션이니 사용시 주의가 필요하다.

따라서 다 지우고 다시 시작하고 싶은 때는 hard를 옵션으로 주면 된다. 이를 이용해 잘못 수정한 파일들을 HEAD 상태로 다시 복원할 수 있다.

```
$ git reset --hard HEAD
```

출처: http://llnntms.tistory.com/5 [IGNITER] 출처: http://donggov.tistory.com/29 [동고랩]

Publish

Summary: Writing git manual

GIT Publish-related command

Publish는 변경 내용을 발행하는 것이다. 현재의 변경내용은 아직 로컬 저장소의 head 안에 있다. 프로젝트를 공유하고 싶을 때 리모트 저장소에 Push할 수 있다. 이 명령은 git push [리모트 저장소 이름] [브랜치 이름]로 단순하다. master 브랜치를 origin 서버에 Push하려면 아래와 같이 서버에 Push한다.

- publish의 명령어

```
$ git push
$ git push remote
$ git format-patch origin
```

$ git push

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

$ git format-patch origin

이 명령은 소스코드가 동일하지만 둘 이상의 브랜치에 적용해야 하는 경우 한 곳에서 작업한 내용을 다른 곳에 쉽게 반영하려면 먼저 방금 커밋한 내용에 대한 패치 파일을 생성한다.

git update-related command

Summary: Writing git manual

Update

공유하고 업데이트하기

Git에는 네트워크가 필요한 명령어가 많지 않다. 거의 로컬 데이터베이스만으로 동작한다. 코드를 공유하거나 가져올 때 필요한 명령어가 몇 개 있다. 이런 명령어는 모두 리모트 저장소를 다루는 명령어다.

- git fetch

git fetch 명령은 로컬 데이터베이스에 있는 것을 뺀 리모트 저장소의 모든 것을 가져온다.

“리모트 저장소를 Pull 하거나 Fetch 하기”에서 이 명령을 설명하고 “리모트 브랜치”에 보면 참고할 수 있는 예제가 더 있다.

“프로젝트에 기여하기”에도 좋은 예제가 많다.

Ref를 한 개만 가져오는 방법은 “Pull Request의 Ref”에서 설명하고 번들에서 가져오는 방법은 “Bundle”에서 설명한다.

Fetch 하는 기본 Refspec을 수정하는 방법은 “Refspec”에서 설명한다. 원하는 대로 수정할 수 있다.

- git pull

git pull 명령은 git fetch와 git merge 명령을 순서대로 실행하는 것뿐이다. 그래서 해당 리모트에서 Fetch 하고 즉시 현 브랜치로 Merge를 시도한다.

“리모트 저장소를 Pull 하거나 Fetch 하기”에서 이 명령을 사용하는 방법을 다뤘고 정확히 무엇을 Merge 하는 지는 “리모트 저장소 살펴보기”에서 설명한다.

“Rebase 한 것을 다시 Rebase 하기”에서 그 어렵다는 Rebase를 다루는 방법을 설명한다.

저장소 URL을 주고 한 번만 Pull 해 올 수 있다는 것을 “리모트 브랜치로부터 통합하기”에서 설명한다.

–verify-signatures 옵션을 주면 Pull 할 때 커밋의 PGP 서명을 검증한다. PGP 서명은 “커밋에 서명하기”에서 설명한다.

- git push

git push 명령은 리모트에는 없지만, 로컬에는 있는 커밋을 계산하고 나서 그 차이만큼만 Push 한다. Push를 하려면 원격 저장소에 대한 쓰기 권한이 필요하고 인증돼야 한다.

리모트에서 git push 명령으로 브랜치를 원격 저장소에 Push 하는 방법을 설명한다. 조금 깊게 브랜치를 하나씩 골라서 Push 하는 방법은 “Push 하기”에서 설명한다. 자동으로 Push 하도록 트래킹 브랜치를 설정하는 방법은 다음에서 설명한다. git push –delete 명령으로 원격 서버의 브랜치를 삭제하는 방법은 “리모트 브랜치 삭제”에서 설명한다.

“프로젝트에 기여하기”에서는 git push를 주구장창 사용한다. 리모트를 여러 개 사용해서 브랜치에 작업한 내용을 공유하는 것을 보여준다.

–tags 옵션을 주고 태그를 Push 하는 방법은 “태그 공유하기”에서 설명한다.

서브모듈의 코드를 수정했을 때는 –recurse-submodules 옵션이 좋다. 프로젝트를 Push 할 때 서브모듈에 Push 할 게 있으면 서브모듈부터 Push 하므로 매우 편리하다. 이 옵션은 “서브모듈 수정 사항 공유하기”에서 설명한다.

“기타 훅”에서 pre-push 훅에 대해서 설명했다. 이 훅에 Push 해도 되는지 검증하는 스크립트를 설정하면 규칙에 따르도록 Push를 검증할 수 있다.

일반적인 이름 규칙에 따라서 Push 하는 것이 아니라 Refspec을 사용해서 원하는 이름으로 Push 하는 것도 가능하다. 이것은 “Refspec Push 하기”에서 설명한다.

- git remote

git remote 명령은 원격 저장소 설정인 리모트의 관리 도구다. 긴 URL 대신 “origin”처럼 이름을 짧게 지을 수 있다. 그리고 URL대신 짧은 리모트 이름을 사용한다. git remote 명령으로 이 리모트를 여러 개 만들어 관리할 수 있다.

이 리모트를 조회하고 추가하고 삭제하고 수정하는 방법은 “리모트 저장소”에서 잘 설명한다.

이 명령은 git remote add 형식으로 사용하고 이 책에서 자주 사용된다.

git archive git archive 명령은 프로젝트 스냅샷을 아카이브 파일로 만들어 준다.

“릴리즈 준비하기”에서 설명하는데 프로젝트를 Tarball로 만들어 공유할 때 사용한다.

- git submodule

git submodule 명령은 저장소 안에서 다른 저장소를 관리하는 데 사용한다. 라이브러리나 특정 형식의 리소스 파일을 서브모듈로 사용할 수 있다. submodule 명령에 있는 add, update, sync 등의 하위 명령어로 서브모듈을 관리할 수 있다.

이 명령은 “서브모듈”에서 설명한다.

출처:[https://git-scm.com/book/ko/v2/Git의-기초-되돌리기]

git resolve-related command

Summary: Writing git manual

고급 Merge

Git의 Merge은 진짜 가볍다. Git에서는 브랜치끼리 몇 번이고 Merge 하기가 쉽다. 오랫동안 합치지 않은 두 브랜치를 한 번에 Merge 하면 거대한 충돌이 발생한다. 조그마한 충돌을 자주 겪고 그걸 풀어나감으로써 브랜치를 최신으로 유지한다.

하지만, 가끔 까다로운 충돌도 발생한다. 다른 버전 관리 시스템과 달리 Git은 충돌이 나면 모호한 상황까지 해결하려 들지 않는다. Git의 철학은 Merge가 잘될지 아닐지 판단하는 것을 잘 하자이다. 충돌이 나도 자동으로 해결하려고 노력하지 않는다. 오랫동안 따로 유지한 두 브랜치를 Merge 하려면 몇 가지 해야 할 일이 있다.

이 절에서는 어떤 Git 명령을 사용해서 무슨 일을 해야 하는지 알아보자. 그 외에도 특수한 상황에서 사용하는 Merge 방법과 Merge를 잘 마무리하는 방법을 소개한다.

Merge 충돌

충돌의 기초에서 기초적인 Merge 충돌 해결에 대해서 다뤘다. Git은 복잡한 Merge 충돌이 났을 때 필요한 도구도 가지고 있다. 무슨 일이 일어났고 어떻게 해결하는 게 나은지 알 수 있다.

Merge 할 때는 충돌이 날 수 있어서 Merge 하기 전에 워킹 디렉토리를 깔끔히 정리하는 것이 좋다. 워킹 디렉토리에 작업하던 게 있다면 임시 브랜치에 커밋하거나 Stash 해둔다. 그래야 어떤 일이 일어나도 다시 되돌릴 수 있다. 작업 중인 파일을 저장하지 않은 채로 Merge 하면 작업했던 일부를 잃을 수도 있다.

매우 간단한 예제를 따라가 보자. 현재 ‘hello world’를 출력하는 Ruby 파일을 하나 가지고 있다.

```
#! /usr/bin/env ruby

def hello
  puts 'hello world'
end
hello()
```

저장소에 whitespace 브랜치를 생성하고 모든 Unix 개행을 DOS 개행으로 바꾸어 커밋한다. 파일의 모든 라인이 바뀌었지만, 공백만 바뀌었다. 그 후 “hello world” 문자열을 ‘`hello mundo’’로 바꾼 다음에 커밋한다.

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

$ unix2dos hello.rb
unix2dos: converting file hello.rb to DOS format ...
$ git commit -am 'converted hello.rb to DOS'
[whitespace 3270f76] converted hello.rb to DOS
 1 file changed, 7 insertions(+), 7 deletions(-)

$ vim hello.rb
$ git diff -b
diff --git a/hello.rb b/hello.rb
index ac51efd..e85207e 100755
--- a/hello.rb
+++ b/hello.rb
@@ -1,7 +1,7 @@
 #! /usr/bin/env ruby

 def hello
-  puts 'hello world'
+  puts 'hello mundo'^M
 end

 hello()

$ git commit -am 'hello mundo change'
[whitespace 6d338d2] hello mundo change
 1 file changed, 1 insertion(+), 1 deletion(-)
```

master 브랜치로 다시 이동한 다음에 함수에 대한 설명을 추가한다.

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

$ vim hello.rb
$ git diff
diff --git a/hello.rb b/hello.rb
index ac51efd..36c06c8 100755
--- a/hello.rb
+++ b/hello.rb
@@ -1,5 +1,6 @@
 #! /usr/bin/env ruby

+# prints out a greeting
 def hello
   puts 'hello world'
 end

$ git commit -am 'document the function'
[master bec6336] document the function
 1 file changed, 1 insertion(+)
```

이때 whitespace 브랜치를 Merge 하면 공백변경 탓에 충돌이 난다.

```
$ git merge whitespace
Auto-merging hello.rb
CONFLICT (content): Merge conflict in hello.rb
Automatic merge failed; fix conflicts and then commit the resul
t.
```

Merge 취소하기

Merge 중에 발생한 충돌을 해결하는 방법은 몇 가지가 있다. 첫 번째는 그저 이 상황을 벗어나는 것이다. 예상하고 있던 일도 아니고 지금 당장 처리할 일도 아니라면 git merge –abort 명령으로 간단히 Merge 하기 전으로 되돌린다.

```
$ git status -sb
## master
UU hello.rb

$ git merge --abort

$ git status -sb
## master
```

git merge –abort 명령은 Merge 하기 전으로 되돌린다. 완전히 뒤로 되돌리지 못하는 유일한 경우는 Merge 전에 워킹 디렉토리에서 Stash 하지 않았거나 커밋하지 않은 파일이 존재하고 있었을 때뿐이다. 그 외에는 잘 돌아간다.

어떤 이유로든 Merge를 처음부터 다시 하고 싶다면 git reset –hard HEAD 명령으로 되돌릴 수 있다. 이 명령은 워킹 디렉토리를 그 시점으로 완전히 되돌려서 저장하지 않은 것은 사라진다는 점에 주의하자.

공백 무시하기

공백 때문에 충돌이 날 때도 있다. 단순한 상황이고 실제로 충돌난 파일을 살펴봤을 때 한 쪽의 모든 라인이 지워지고 다른 쪽에는 추가됐기 때문에 간단하다고 할 수 있다. 기본적으로 Git은 이런 모든 라인이 변경됐다고 인지하여 Merge 할 수 없다.

기본 Merge 전략은 공백의 변화는 무시하도록 하는 옵션을 주는 것이다. Merge 할 때 무수한 공백 때문에 문제가 생기면 그냥 Merge를 취소한 다음 -Xignore-all-space나 -Xignore-space-change 옵션을 주어 다시 Merge 한다. 첫 번째 옵션은 모든 공백을 무시하고 두 번째 옵션은 뭉쳐 있는 공백을 하나로 취급한다.

```
$ git merge -Xignore-space-change whitespace
Auto-merging hello.rb
Merge made by the 'recursive' strategy.
 hello.rb | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```

위 예제는 모든 공백 변경 사항을 무시하면 실제 파일은 충돌 나지 않고 모든 Merge가 잘 실행된다.

팀원 중 누군가 스페이스를 탭으로 바꾸거나 탭을 스페이스로 바꾸는 짓을 했을 때 이 옵션이 그대를 구원해 준다.

수동으로 Merge 하기

Merge 작업할 때 공백 처리 옵션을 사용하면 Git이 꽤 잘해준다. 하지만, Git이 자동으로 해결하지 못하는 때도 있다. 이럴 때는 외부 도구의 도움을 받아 해결한다. 예를 들어 Git이 자동으로 해결해주지 못하는 상황에 부닥치면 직접 손으로 해결해야 한다.

파일을 dos2unix로 변환하고 Merge 하면 된다. 이걸 Git에서 어떻게 하는지 살펴보자.

먼저 Merge 충돌 상태에 있다고 치자. 현 시점의 파일과 Merge 할 파일, 공통 조상의 파일이 필요하다. 이 파일들로 어쨌든 잘 Merge 되도록 수정하고 다시 Merge를 시도해야 한다.

우선 세 가지 버전의 파일을 얻는 건 쉽다. Git은 세 버전의 모든 파일에 “stages” 숫자를 붙여서 Index에 다 가지고 있다. Stage 1는 공통 조상 파일, Stage 2는 현재 개발자의 버전에 해당하는 파일, Stage 3은 `MERGE_HEAD`가 가리키는 커밋의 파일이다.

git show 명령으로 각 버전의 파일을 꺼낼 수 있다.

```
$ git show :1:hello.rb > hello.common.rb
$ git show :2:hello.rb > hello.ours.rb
$ git show :3:hello.rb > hello.theirs.rb
```

좀 더 저수준으로 파고들자면 ls-files -u 명령을 사용한다. 이 명령은 Plumbing 명령으로 각 파일을 나타내는 Git Blob의 SHA를 얻을 수 있다.

```
$ git ls-files -u
100755 ac51efdc3df4f4fd328d1a02ad05331d8e2c9111 1        hell
o.rb
100755 36c06c8752c78d2aff89571132f3bf7841a7b5c3 2        hell
o.rb
100755 e85207e04dfdd5eb0a1e9febbc67fd837c44a1cd 3        hell
o.rb
```

:1:hello.rb는 그냥 Blob SHA-1를 지칭하는 줄임말이다.

이제 워킹 디렉토리에 세 버전의 파일을 모두 가져왔다. 공백 문제를 수동으로 고친 다음에 다시 Merge 한다. Merge 할 때는 git merge-file 명령을 이용한다.

```
$ dos2unix hello.theirs.rb
dos2unix: converting file hello.theirs.rb to Unix format ...

$ git merge-file -p \
    hello.ours.rb hello.common.rb hello.theirs.rb > hello.rb

$ git diff -b
diff --cc hello.rb
index 36c06c8,e85207e..0000000
--- a/hello.rb
+++ b/hello.rb
@@@ -1,8 -1,7 +1,8 @@@
  #! /usr/bin/env ruby

 +# prints out a greeting
  def hello
-   puts 'hello world'
+   puts 'hello mundo'
  end

  hello()
```

이렇게 해서 멋지게 Merge가 완료된 파일을 얻었다. 사실 이것이 ignore-all-space 옵션을 사용하는 것보다 더 나은 방법이다. 왜냐면 공백을 무시하지 않고 실제로 고쳤기 때문이다. ignore-all-space 옵션을 사용한 Merge 에서는 여전히 DOS의 개행 문자가 남아서 한 파일에 두 형식의 개행문자가 뒤섞인다.

Merge 커밋을 완료하기 전에 양쪽 부모에 대해서 무엇이 바뀌었는지 확인하려면 git diff를 사용한다. 이 명령을 이용하면 Merge 의 결과로 워킹 디렉토리에 무엇이 바뀌었는지 알 수 있다. 한번 자세히 살펴보자.

Merge 후의 결과를 Merge 하기 전의 브랜치와 비교하려면, 다시 말해 무엇이 합쳐졌는지 알려면 git diff –ours 명령을 실행한다.

```
$ git diff --ours
* Unmerged path hello.rb
diff --git a/hello.rb b/hello.rb
index 36c06c8..44d0a25 100755
--- a/hello.rb
+++ b/hello.rb
@@ -2,7 +2,7 @@

 # prints out a greeting
 def hello
-  puts 'hello world'
+  puts 'hello mundo'
 end

 hello()
```

위의 결과에서 Merge를 했을 때 현재 브랜치에서는 무엇을 추가했는지를 알 수 있다.

Merge 할 파일을 가져온 쪽과 비교해서 무엇이 바뀌었는지 보려면 git diff –theirs를 실행한다. 아래 예제에서는 공백을 빼고 비교하기 위해 -b 옵션을 같이 써주었다.

```
$ git diff --theirs -b
* Unmerged path hello.rb
diff --git a/hello.rb b/hello.rb
index e85207e..44d0a25 100755
--- a/hello.rb
+++ b/hello.rb
@@ -1,5 +1,6 @@
 #! /usr/bin/env ruby

+# prints out a greeting
 def hello
   puts 'hello mundo'
 end
```

마지막으로 `git diff --base` 를 사용해서 양쪽 모두와 비교하여 바뀐 점을 알아본다.

```
$ git diff --base -b
* Unmerged path hello.rb
diff --git a/hello.rb b/hello.rb
index ac51efd..44d0a25 100755
--- a/hello.rb
+++ b/hello.rb
@@ -1,7 +1,8 @@
 #! /usr/bin/env ruby

+# prints out a greeting
 def hello
-  puts 'hello world'
+  puts 'hello mundo'
 end

 hello()
```

수동 Merge를 위해서 만들었던 각종 파일은 이제 필요 없으니 git clean 명령을 실행해서 지워준다.

```
$ git clean -f
Removing hello.common.rb
Removing hello.ours.rb
Removing hello.theirs.rb
```

충돌 파일 Checkout

앞서 살펴본 여러가지 방법으로 충돌을 해결했지만 바라던 결과가 아닐 수도 있고 심지어 결과가 잘 동작하지 않아 충돌을 직접 수동으로 더 많은 정보를 살펴보며 해결해야 하는 경우도 있다.

예제를 조금 바꿔보자. 이번 예제에서는 긴 호흡의 브랜치 두 개가 있다. 각 브랜치에는 몇 개의 커밋이 있는데 양쪽은 Merge 할 때 반드시 충돌이 날 만한 내용이 들어 있다.

```
$ git log --graph --oneline --decorate --all
* f1270f7 (HEAD, master) update README
* 9af9d3b add a README
* 694971d update phrase to hola world
| * e3eb223 (mundo) add more tests
| * 7cff591 add testing script
| * c3ffff1 changed text to hello mundo
|/
* b7dcc89 initial hello world code
```

master에만 있는 세 개의 커밋과 mundo 브랜치에만 존재하는 또 다른 세 개의 커밋이 있다. master 브랜치에서 mundo 브랜치를 Merge 하면 충돌이 난다.

```
$ git merge mundo
Auto-merging hello.rb
CONFLICT (content): Merge conflict in hello.rb
Automatic merge failed; fix conflicts and then commit the resul
t.
```

해당 파일을 열어서 충돌이 발생한 내용을 보면 아래와 같다.

```
#! /usr/bin/env ruby

def hello
  puts 'hola world'
  puts 'hello mundo'
end

hello()
```

양쪽 브랜치에서 추가된 부분이 이 파일에 다 적용됐다. 적용한 커밋 중 파일의 같은 부분을 수정해서 위와 같은 충돌이 생긴다.

충돌을 해결하는 몇 가지 도구에 대해 알아보자. 어쩌면 이 충돌을 어떻게 해결해야 하는지 명확하지 않을 수도 있다. 맥락을 좀 더 살펴봐야 하는 상황 말이다.

git checkout 명령에 –conflict 옵션을 붙여 사용하는 게 좋은 방법이 될 수 있다. 이 명령은 파일을 다시 Checkout 받아서 충돌 표시된 부분을 교체한다. 충돌 난 부분은 원래의 코드로 되돌리고 다시 고쳐보려고 할 때 알맞은 도구다.

–conflict 옵션에는 diff3나 `merge를 넘길 수 있고 merge가 기본 값이다.
–conflict 옵션에 diff3를 사용하면 Git은 약간 다른 모양의 충돌 표시를 남긴다.
“ours”나 “theirs”말고도 “base”버전의 내용까지 제공한다.

```
$ git checkout --conflict=diff3 hello.rb
```

위 명령을 실행하면 아래와 같은 결과가 나타난다.

```
#! /usr/bin/env ruby

def hello
  puts 'hola world'
||||||| base
  puts 'hello world'

  puts 'hello mundo'
end

hello()
```

이런 형태의 충돌 표시를 계속 보고 싶다면 기본으로 사용하도록 merge.conflictstyle 설정 값을 `diff3` 로 설정한다.

```
$ git config --global merge.conflictstyle diff3
git checkout 명령도 --ours와 --theirs 옵션을 지원한다. 이 옵션은 Mer
ge 하지 않고 둘 중 한쪽만을 선택할 때 사용한다.
```

이 옵션은 바이너리 파일이 충돌 나서 한쪽을 선택해야 하는 상황이나 한쪽 브랜치의 온전한 파일을 원할 때 사용할 수 있다. 일단 Merge 하고 나서 특정 파일만 Checkout 한 후에 커밋하는 방법도 있다.

Merge 로그

git log 명령은 충돌을 해결할 때도 도움이 된다. 로그에는 충돌을 해결할 때 도움이 될만한 정보가 있을 수 있다. 과거를 살짝 들춰보면 개발 당시에 같은 곳을 고쳐야만 했던 이유를 밝혀내는 데 도움이 된다.

“Triple Dot” 문법을 이용하면 Merge 에 사용한 양 브랜치의 모든 커밋의 목록을 얻을 수 있다. 자세한 문법은 Triple Dot를 참고한다.

```
$ git log --oneline --left-right HEAD...MERGE_HEAD
< f1270f7 update README
< 9af9d3b add a README
< 694971d update phrase to hola world
> e3eb223 add more tests
> 7cff591 add testing script
> c3ffff1 changed text to hello mundo
```

위와 같이 총 6개의 커밋을 볼 수 있다. 커밋이 어떤 브랜치에서 온 것인지 보여준다.

맥락에 따라 필요한 결과만 추려 볼 수도 있다. git log 명령에 –merge 옵션을 추가하면 충돌이 발생한 파일이 속한 커밋만 보여준다.

```
$ git log --oneline --left-right --merge
< 694971d update phrase to hola world
> c3ffff1 changed text to hello mundo
`--merge`대신 `-p`를 사용하면 충돌 난 파일의 변경사항만 볼 수 있다. 이건 왜 충돌이 났는지 또 이를 해결하기 위해 어떻게 해야 하는지 이해하는 데 진짜로 유용하다.
```

Combined Diff 형식

Merge가 성공적으로 끝난 파일은 Staging Area에 올려놓았다. 이 상태에서 충돌 난 파일들이 그대로 있을 때 git diff 명령을 실행하면 충돌 난 파일이 무엇인지 알 수 있다. 어떤 걸 더 고쳐야 하는지 아는 데에 도움이 된다.

Merge 하다가 충돌이 났을 때 git diff 명령을 실행하면 꽤 생소한 Diff 결과를 보여준다.

```
$ git diff
diff --cc hello.rb
index 0399cd5,59727f0..0000000
--- a/hello.rb
+++ b/hello.rb
@@@ -1,7 -1,7 +1,11 @@@
  #! /usr/bin/env ruby

  def hello
++<<<<<<< HEAD
 +  puts 'hola world'
++=======
+   puts 'hello mundo'
++>>>>>>> mundo
  end

  hello()
```

이런 형식을 “Combined Diff’라고 한다. 각 라인은 두 개의 컬럼으로 구분할 수 있다. 첫 번째 컬럼은 “ours” 브랜치와 워킹 디렉토리의 차이(추가 또는 삭제)를 보여준다. 두 번째 컬럼은 ``theirs’와 워킹 디렉토리사이의 차이를 나타낸다.

이 예제에서 ««««<와>>>>>>> 충돌 마커 표시는 어떤 쪽에도 존재하지 않고 추가된 코드라는 것을 알 수 있다. 이 표시는 Merge 도구가 만들어낸 코드이기 때문이다. 물론 이 표시는 지워야 하는 라인이다.

충돌을 다 해결하고 git diff 명령을 다시 실행하면 아래와 같이 보여준다. 이 결과도 유용하다.

```
$ vim hello.rb
$ git diff
diff --cc hello.rb
index 0399cd5,59727f0..0000000
--- a/hello.rb
+++ b/hello.rb
@@@ -1,7 -1,7 +1,7 @@@
  #! /usr/bin/env ruby

  def hello
-   puts 'hola world'
 -  puts 'hello mundo'
++  puts 'hola mundo'
  end

  hello()
```

이 결과는 세 가지 사실을 보여준다. “hola world”는 Our 브랜치에 있었지만 워킹 디렉토리에는 없다. “hello mundo”는 Their 브랜치에 있었지만 워킹 디렉토리에는 없다. “hola mundo”는 어느 쪽 브랜치에도 없고 워킹 디렉토리에는 있다. 충돌을 해결하고 마지막으로 확인하고 나서 커밋하는 데 유용하다.

이 정보를 git log 명령을 통해서도 얻을 수 있다. Merge 후에 무엇이 어떻게 바뀌었는지 알아야 할 때 유용하다. Merge 커밋에 대해서 git show 명령을 실행하거나 git log -p에 –cc 옵션을 추가해도 같은 결과를 얻을 수 있다. git log -p 명령은 기본적으로 Merge 커밋이 아닌 커밋의 Patch를 출력한다.

```
$ git log --cc -p -1
commit 14f41939956d80b9e17bb8721354c33f8d5b5a79
Merge: f1270f7 e3eb223
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri Sep 19 18:14:49 2014 +0200

    Merge branch 'mundo'

    Conflicts:
        hello.rb

diff --cc hello.rb
index 0399cd5,59727f0..e1d0799
--- a/hello.rb
+++ b/hello.rb
@@@ -1,7 -1,7 +1,7 @@@
  #! /usr/bin/env ruby

  def hello
-   puts 'hola world'
-  puts 'hello mundo'
++  puts 'hola mundo'
  end

  hello()
```

Merge 되돌리기

지금까지 Merge 하는 방법을 배웠으나 Merge 할 때 실수할 수도 있다. Git에서는 실수해도 된다. 실수해도 (대부분 간단하게) 되돌릴 수 있다.

Merge 커밋도 예외는 아니다. 토픽 브랜치에서 일을 하다가 master로 잘못 Merge 했다고 생각해보자. 커밋 히스토리는 아래와 같다.

우발적인 Merge 커밋. 접근 방식은 원하는 결과에 따라 두 가지로 나눌 수 있다.

Refs 수정

실수로 생긴 Merge 커밋이 로컬 저장소에만 있을 때는 브랜치를 원하는 커밋을 가리키도록 옮기는 것이 쉽고 빠르다. 잘못 Merge 하고 나서 git reset –hard HEAD~ 명령으로 브랜치를 되돌리면 된다.

git reset –hard HEAD~ 실행 후의 히스토리. reset에 대해서는 이미 앞의 «_git_reset»에서 다뤘었기 때문에 이 내용이 그리 어렵진 않을 것이다. 간단하게 복습해보자. reset –hard 명령은 아래의 세 단계로 수행한다.

1. HEAD의 브랜치를 지정한 위치로 옮긴다. 이 경우엔 master 브랜치를 Merge 커밋(C6) 이전으로 되돌린다.
2. Index를 HEAD의 내용으로 바꾼다.
3. 워킹 디렉토리를 Index의 내용으로 바3꾼다.

이 방법의 단점은 히스토리를 다시 쓴다는 것이다. 다른 사람들과 공유된 저장소에서 히스토리를 덮어쓰면 문제가 생길 수 있다. 무슨 문제가 일어나는지 알고 싶다면 Rebase 의 위험성를 참고하자. 간단히 말해 다시 쓰는 커밋이 이미 다른 사람들과 공유한 커밋이라면 reset 하지 않는 게 좋다. 이 방법은 Merge 하고 나서 다른 커밋을 생성했다면 제대로 동작하지 않는다. HEAD를 이동시키면 Merge 이후에 만든 커밋을 잃어버린다.

커밋 되돌리기

브랜치를 옮기는 것을 할 수 없는 경우는 모든 변경사항을 취소하는 새로운 커밋을 만들 수도 있다. Git에서 이 기능을 “revert” 라고 부른다. 지금의 경우엔 아래처럼 실행한다.

```
$ git revert -m 1 HEAD
[master b1d8379] Revert "Merge branch 'topic'"
```

-m 1 옵션은 부모가 보호되어야 하는 “mainline” 이라는 것을 나타낸다. HEAD로 Merge를 했을 때(git merge topic1) Merge 커밋은 두 개의 부모 커밋을 가진다. 첫 번째 부모 커밋은 HEAD(C6)이고 두 번째 부모 커밋은 Merge 대상 브랜치(C4)이다. 두 번째 부모 커밋(C4)에서 받아온 모든 변경사항을 되돌리고 첫 번째 부모(C6)로부터 받아온 변경사항은 남겨두고자 하는 상황이다.

변경사항을 되돌린 커밋은 히스토리에서 아래와 같이 보인다.

`git revert -m 1` 실행 후의 히스토리. Figure 140. git revert -m 1 실행 후의 히스토리 새로 만든 커밋 ^M은 C6과 내용이 완전히 똑같다. 잘못 Merge 한 커밋까지 HEAD의 히스토리에서 볼 수 있다는 것 말고는 Merge 하지 않은 것과 같다. topic 브랜치를 master 브랜치에 다시 Merge 하면 Git은 아래와 같이 어리둥절해 한다.

```
$ git merge topic
Already up-to-date.
```

이미 Merge 했던 topic 브랜치에는 더는 master 브랜치로 Merge 할 내용이 없다. 상황을 더 혼란스럽게 하는 경우는 topic에서 뭔가 더 일을 하고 다시 Merge를 하는 경우이다. Git은 Merge 이후에 새로 만들어진 커밋만 가져온다.

좋지 않은 Merge가 있는 히스토리. 이러면 가장 좋은 방법은 되돌렸던 Merge 커밋을 다시 되돌리는 것이다. 이후에 추가한 내용을 새 Merge 커밋으로 만드는 게 좋다.

```
$ git revert ^M
[master 09f0126] Revert "Revert "Merge branch 'topic'""
$ git merge topic
```

되돌린 Merge를 다시 Merge 한 후의 히스토리. 위 예제에서는 M과 ^M이 상쇄됐다. ^^M는 C3와 C4의 변경 사항을 담고 있고 C8은 C7의 내용을 훌륭하게 Merge 했다. 이리하여 현재 topic 브랜치를 완전히 Merge 한 상태가 됐다.

다른 방식의 Merge

지금까지 두 브랜치를 평범하게 Merge 하는 방법에 대해 알아보았다. Merge는 보통 “recursive” 전략을 사용한다. 브랜치를 한 번에 Merge 하는 방법은 여러 가지다. 그 중 몇 개만 간단히 알아보자.

Our/Their 선택하기

먼저 일반적인 “recursive” 전략을 사용하는 Merge 작업을 할 때 유용한 옵션을 소개한다. 앞에서 ignore-all-space와 ignore-space-change 기능을 -X 옵션에 붙여 쓰는 것을 보았다. 이 -X 옵션은 충돌이 났을 때 어떤 한 쪽을 선택할 때도 사용한다.

아무 옵션도 지정하지 않고 두 브랜치를 Merge 하면 Git은 코드에 충돌 난 곳을 표시하고 해당 파일을 충돌 난 파일로 표시해준다. 충돌을 직접 해결하는 게 아니라 미리 Git에게 충돌이 났을 때 두 브랜치 중 한쪽을 선택하라고 알려줄 수 있다. merge 명령을 사용할 때 -Xours나 Xtheirs 옵션을 추가하면 된다.

Git에 이 옵션을 주면 충돌 표시가 남지 않는다. Merge가 가능하면 Merge 될 것이고 충돌이 나면 사용자가 명시한 쪽의 내용으로 대체한다. 바이너리 파일도 똑같다.

“hello world” 예제로 돌아가서 다시 Merge를 해보자. Merge를 하면 충돌이 나는 것을 볼 수 있다.

```
$ git merge mundo
Auto-merging hello.rb
CONFLICT (content): Merge conflict in hello.rb
Resolved 'hello.rb' using previous resolution.
Automatic merge failed; fix conflicts and then commit the resul
t.
```

하지만 -Xours나 -Xtheirs 옵션을 주면 충돌이 났다는 소리가 없다.

```
$ git merge -Xours mundo
Auto-merging hello.rb
Merge made by the 'recursive' strategy.
 hello.rb | 2 +-
 test.sh  | 2 ++
 2 files changed, 3 insertions(+), 1 deletion(-)
 create mode 100644 test.sh
```

한쪽 파일에는 “hello mundo”가 있고 다른 파일에는 “hola world”가 있다. 이 Merge에서 충돌 표시를 하는 대신 간단히 “hola world”를 선택한다. 충돌 나지 않은 나머지는 잘 Merge 된다.

이 옵션은 git merge-file 명령에도 사용할 수 있다. 앞에서 이미 git merge-file –ours 같이 실행해서 파일을 따로따로 Merge 했다.

이런 식의 동작을 원하지만 애초에 Git이 Merge 시도조차 하지 않는 자비 없는 옵션도 있다. “ours” Merge 전략이다. 이 전략은 Recursive Merge 전략의 “ours” 옵션과는 다르다.

이 작업은 기본적으로 거짓으로 Merge 한다. 그리고 양 브랜치를 부모로 삼는 새 Merge 커밋을 만든다. 하지만, Their 브랜치는 참고하지 않는다. Our 브랜치의 코드를 그대로 사용하고 Merge 한 것처럼 기록할 뿐이다.

```
$ git merge -s ours mundo
Merge made by the 'ours' strategy.
$ git diff HEAD HEAD~
$
```

지금 있는 브랜치와 Merge 결과가 다르지 않다는 것을 알 수 있다.

이 ours 전략을 이용해 이미 Merge가 되었다고 Git을 속이고 실제로는 Merge를 나중에 수행한다. 예를 들어 release 브랜치을 만들고 여기에도 코드를 추가했다. 언젠가 이것을 master 브랜치에도 Merge 해야 하지만 아직은 하지 않았다. 그리고 master 브랜치에서 bugfix 브랜치를 만들어 버그를 수정하고 이것을 release 브랜치에도 적용(Backport)해야 한다. bugfix 브랜치를 release 브랜치로 Merge 하고 이미 포함된 master 브랜치에도 merge -s ours 명령으로 Merge 해 둔다. 이렇게 하면 나중에 release 브랜치를 Merge 할 때 버그 수정에 대한 커밋으로 충돌이 일어나지 않게끔 할 수 있다.

서브트리 Merge

서브트리 Merge 의 개념은 프로젝트 두 개가 있을 때 한 프로젝트를 다른 프로젝트의 하위 디렉토리로 매핑하여 사용하는 것이다. Merge 전략으로 서브트리(Subtree)를 사용하는 경우 Git은 매우 똑똑하게 서브트리를 찾아서 메인 프로젝트로 서브프로젝트의 내용을 Merge 한다.

한 저장소에 완전히 다른 프로젝트의 리모트 저장소를 추가하고 데이터를 가져와서 Merge 까지 하는 과정을 살펴보자.

먼저 Rack 프로젝트 현재 프로젝트에 추가한다. Rack 프로젝트의 리모트 저장소를 현재 프로젝트의 리모트로 추가하고 Rack 프로젝트의 브랜치와 히스토리를 가져와(Fetch) 확인한다.

```
$ git remote add rack_remote https://github.com/rack/rack
$ git fetch rack_remote --no-tags
warning: no common commits
remote: Counting objects: 3184, done.
remote: Compressing objects: 100% (1465/1465), done.
remote: Total 3184 (delta 1952), reused 2770 (delta 1675)
Receiving objects: 100% (3184/3184), 677.42 KiB | 4 KiB/s, don
e.
Resolving deltas: 100% (1952/1952), done.
From https://github.com/rack/rack
 * [new branch]      build      -> rack_remote/build
 * [new branch]      master     -> rack_remote/master
 * [new branch]      rack-0.4   -> rack_remote/rack-0.4
 * [new branch]      rack-0.9   -> rack_remote/rack-0.9
$ git checkout -b rack_branch rack_remote/master
Branch rack_branch set up to track remote branch refs/remotes/r
ack_remote/master.
Switched to a new branch "rack_branch"
```

(역주 - git fetch rack_remote 명령의 결과에서 warning: no common commits 메시지를 주목해야 한다.) Rack 프로젝트의 브랜치인 rack_branch를 만들었다. 원 프로젝트는 master 브랜치에 있다. checkout 명령으로 두 브랜치를 이동하면 전혀 다른 두 프로젝트가 한 저장소에 있는 것처럼 보인다.

```
$ ls
AUTHORS         KNOWN-ISSUES   Rakefile      contrib         li
b
COPYING         README         bin           example         te
st
$ git checkout master
Switched to branch "master"
$ ls
README
```

상당히 요상한 방식으로 Git을 활용한다. 저장소의 브랜치가 꼭 같은 프로젝트가 아닐 수도 있다. Git에서는 전혀 다른 브랜치를 쉽게 만들 수 있다. 물론 이렇게 사용하는 경우는 드물다.

Rack 프로젝트를 master 브랜치의 하위 디렉토리로 만들 수 있다. 이는 git read-tree 명령을 사용한다. read-tree 명령과 같이 저수준 명령에 관련된 많은 내용은 Git의 내부에서 다룬다. 간단히 말하자면 read-tree 명령은 어떤 브랜치로부터 루트 트리를 읽어서 현재 Staging Area나 워킹 디렉토리로 가져온다. master 브랜치로 다시 Checkout 하고 rack_branch 브랜치를 rack이라는 master 브랜치의 하위 디렉토리로 만들어보자.

`$ git read-tree --prefix=rack/ -u rack_branch ``

이제 커밋하면 Rack 프로젝트의 모든 파일이 Tarball 압축파일을 풀어서 소스코드를 포함한 것 같이 커밋에 새로 추가된다. 이렇게 쉽게 한 브랜치의 내용을 다른 브랜치에 Merge 시킬 수 있다는 점이 흥미롭지 않은가? Rack 프로젝트가 업데이트되면 Pull 해서 master 브랜치도 적용할 수 있을까?

```
$ git checkout rack_branch
$ git pull
```

위의 명령을 실행하고 업데이트된 결과를 master 브랜치로 다시 Merge 한다. Recursive Merge 전략 옵션인 -Xsubtree 옵션과 –squash 옵션을 함께 사용하면 동일한 커밋 메시지로 업데이트할 수 있다. (Recursive 전략이 기본 전략이지만 설명을 위해서 사용한다)

```
$ git checkout master
$ git merge --squash -s recursive -Xsubtree=rack rack_branch
Squash commit -- not updating HEAD
Automatic merge went well; stopped before committing as request
ed
```

위 명령을 실행하면 Rack 프로젝트에서 변경된 모든 부분이 master 브랜치로 반영되고 커밋할 준비가 완료된다. 반대로 rack 하위 디렉토리에서 변경한 내용을 rack_branch로 Merge 하는 것도 가능하다. 변경한 것을 메인테이너에게 보내거나 Upstream에 Push 한다.

이런 방식은 서브모듈(서브모듈에서 자세하게 다룬다)을 사용하지 않고 서브모듈을 관리하는 또 다른 워크플로이다. 한 저장소 안에 다른 프로젝트까지 유지하면서 서브트리 Merge 전략으로 업데이트도 할 수 있다. 프로젝트에 필요한 코드를 한 저장소에서 관리할 수 있다. 다만, 이렇게 저장소를 관리하는 방법은 저장소를 다루기 좀 복잡하고 통합할 때 실수하기 쉽다. 엉뚱한 저장소로 Push 해버릴 가능성도 있다.

diff 명령으로 rack 하위 디렉토리와 rack_branch의 차이를 볼 때도 이상하다. Merge 하기 전에 두 차이를 보고 싶어도 diff 명령을 사용할 수 없다. 대신 git diff-tree 명령이 준비돼 있다.

```
$ git diff-tree -p rack_branch
```

혹은 rack 하위 디렉토리가 Rack 프로젝트의 리모트 저장소의 master 브랜치와 어떤 차이가 있는지 살펴보고 싶을 수도 있다. 마지막으로 Fetch 한 리모트의 master 브랜치와 비교하려면 아래와 같은 명령을 사용한다.

```
$ git diff-tree -p rack_remote/master
```

출처 : https://git-scm.com/book/ko/v2/Git-%EB%8F%84%EA%B5%AC-%EA%B3%A0%EA%B8%89-Merge

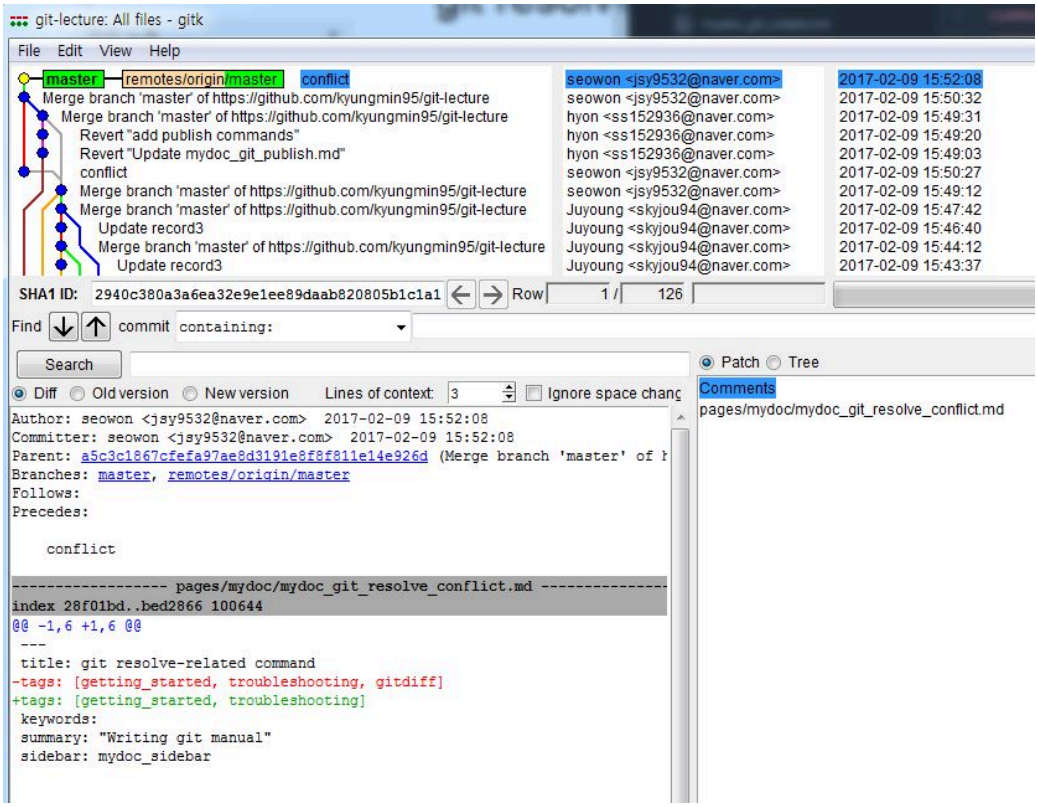
Reslove Conflicts Use add to mark files as resolved.

예시는 아래와 같다.

```
git diff [--base]

git diff --ours

git diff --theirs

git log --merge

gitk --merge
```

- git diff는 두 커밋간이나 HEAD와 워킹 디렉토리의 차이점을 보여주는 명령어이다.
- git diff 명령어를 사용하면 수정된 라인의 전, 후를 비교할 수 있다.
- git diff 명령어는 수정사항이 있을 때 많이 사용한다.
- git diff HEAD...원하는커밋아이디 를 써주면 처음부터 커밋아이디까지의 변경사항을 볼 수 있다.
- gitk를 사용하면 지금까지 작업의 히스토리가 보여진다.

예시 화면은 아래와 같다.



git useful command

Summary: Writing git manual

Super Useful Need To Know Git Commands

Setup

- git clone url Clone a repository specified by url. This is similar to “checkout” in some other version control systems such as Subversion and CVS.
- git init Create an new git repository in the current directory or reinitialize an existing one.
- git init –bare Create an new git repository in the current directory without an associated working tree. This is useful for repositories that serve as mirrors.
- git update-server-info Allow a git repository to act as a dumb server (for remote access).

Adding/Deleting

- git add file1 file2 … Add file1, file2, etc. to the project.
- git add dir Add all files under directory dir to the project, including subdirectories.
- git add . Add all files under the current directory to the project, including subdirectories.
- git rm file1 file2 … Remove file1, file2, etc. from the project (and the filesystem).

Commiting

- git commit file1 file2 … [-m msg] Commit changes in file1, file2, etc., optionally using commit message msg or otherwise opening

editor for commit message entry.

- git commit -a [-m msg] Commit changes made to all tracked files since the last commit, optionally using commit message msg or otherwise opening editor for commit message entry.
- git commit –amend file1 file2 … [-m msg] Re-commit previous commit, including file1, file2, etc., using previous commit message or, optionally, a new one given by msg.

Sharing

- git push [remote] Update the remote repository named remote with commits across all branches that are common between your local repository and remote. If remote is not specified, but a remote named “origin” is defined, then remote defaults to “origin”. Local branches that were never pushed to the server in the first place are not shared.
- git push remote branch Update the remote repository named remote (e.g. “origin”) with commits made to branch since the last push. This is always required for new local branches (including “master” in a new repository). After the first explicit push, “git push” by itself is sufficient.
- git pull remote Update the current branch with changes from the remote named remote (defaults to “origin” if not given). Note that for this to work, “.git/config” must define merge configuration variables for the current branch.

Information

Changes and Differences

- git status Show files added to the index, files with changes, and untracked files.
- git diff Show unstaged changes made since your last commit. git diff –cached Show changes staged for commit (i.e., difference between index and last commit).
- git diff HEAD Show changes (staged and unstaged) in working directory since last commit.
- git diff rev [path(s)] Show differences between working directory and revision rev, optionally limiting comparison to files found in

one or more space-separated file paths or subdirectories given by path(s).

- git diff rev1..rev2 [path(s)] Show differences between two revisions, rev1 and rev2, optionally limiting comparison to files found in one or more space-separated file paths or subdirectories given by path(s).
- git diff rev1...rev2 [path(s)] Show differences between the last common ancestor of two revisions, rev1 and rev2, optionally limiting comparison to files found in one or more space-separated file paths or subdirectories given by path(s).

File and Directory Contents

- git show rev:file Show contents of file (specified relative to the project root) from revision rev.
- git ls-files [-t] Show all tracked files ("-t" shows file status).
- git ls-files –others Show all untracked files.

Commit History

- git log Show recent commits, most recent on top.
- git log [path(s)] Show recent commits, most recent on top, limited to the file or files found on path(s) if given.
- git log -p Show recent commits, most recent on top, with full diffs.
- git log -p [path(s)] Show recent commits, most recent on top, with full diffs, limited to files found in one or more space-separated file paths or subdirectories given by path(s).
- git log -g Show recent commits, most recent on top, walking the full reflog entries instead of the commit ancestry chain up to the current HEAD. By default, "git log" reports all commits only up to the current HEAD, even if HEAD has descendents on the current branch (as, for example, might happen if you ran "git reset rev" to move HEAD to a previous point in history). The "-g" option will report the full history.
- git log –stat [path(s)] Show recent commits, with stats (files changed, insertions, and deletions), optionally limited to files found in one or more space-separated file paths or subdirectories given by path(s).

- git log –author=author Show recent commits, only by author.
- git log –after=”MMM DD YYYY” Show commits that occur after a certain date, e.g. “Jun 20 2008”.
- git log –before=”MMM DD YYYY” Show commits that occur before a certain date.
- git whatchanged file Show only the commits which affected file listing the most recent first.
- git blame file Show who authored each line in file.
- git blame file rev Show who authored each line in file as of rev (allows blame to go back in time).
- git rev-list –all List all commits.
- git rev-list rev1..rev2 List all commits between rev1 and rev2.
- git show rev Show the changeset (diff) of a commit specified by rev.
- git show rev – path(s) Show the changeset (diff) of a commit rev , optionally limited to files found in one or more space-separated file paths or subdirectories given by path(s).

Searching

Searching for Content

- git grep regexp Search working tree for text matching regular expression regexp.
- git grep -e regexp1 [–or] -e regexp2 Search working tree for lines of text matching regular expression regexp1 or regexp2.
- git grep -e regexp1 –and -e regexp2 Search working tree for lines of text matching regular expression regexp1 and regexp2, reporting file paths only.
- git grep -l –all-match -e regexp1 -e regexp2 Search working tree for files that have lines of text matching regular expression regexp1 and lines of text matching regular expression regexp2.
- git grep regexp $(git rev-list –all) Search all revisions for text matching regular expression regexp.
- git grep regexp $(git rev-list rev1..rev2) Search all revisions between rev1 and rev2 for text matching regular expression

regexp.

Searching Logs and Commit History

- git log –grep regexp Search commit logs for lines of text matching regular expression regexp.
- git log –grep regexp1 –grep regexp2 Search commit logs for lines of text matching regular expression regexp1 or regexp2.
- git log –grep regexp1 –and –grep regexp2 Search commit logs for lines of text matching regular expression regexp1 and regexp2.

Branching

Listing Branches

- git branch List all local branches.
- git branch -r List all local and remote branches.

Creating Branches

- git branch new-branch Create a new branch named new-branch, based on current branch. git branch new-branch rev Create a new branch named new-branch, based on revision specified by tree-ish rev. git branch –track new-branch remote/remote-branch Create a new tracking branch named new-branch, referencing, and pushing/pulling from, the branch named remote-branch on remote repository named remote.

Checking Out Branches/Revisions

- git checkout branch Switch to branch named branch. This updates the working tree to reflect the state of the branch named branch, and sets HEAD to “.git/refs/heads/branch”.
- git checkout rev Switch to revision specified by tree-ish rev, without explicitly branching. Running “git checkout -b new-branch” will create a branch from the checked out version.

Simultaneous Creating and Switching Branches

- git checkout -b new-branch Create a new branch named new-branch, referencing the current branch, and check it out.
- git checkout -b new-branch rev Create a new branch named new-branch based on the tree-ish rev, update the working tree to reflect its state, and check it out (switch to it).

Deleting Branches

- git branch -d branch Delete the local branch named branch (fails if branch is not reachable from the current branch).
- git branch -D branch Force delete of the local branch named branch (works even if branch is not reachable from the current branch).
- git branch -d -r remote/branch Delete a “local remote” branch, i.e. a local tracking branch. git push remote :heads/branch Delete a branch named branch from a remote repository.

Merging

In all of the following, a merge strategy can be specified by the “-s strategy” argument, which can be one of: “resolve”, “recursive”, “octopus”, “ours”, or “subtree”. If you tried a merge which resulted in a complex conflicts and would want to start over, you can recover with “git reset –hard”. If you accidently merged and want to unmerge, you can “git reset –hard ORIG_HEAD”.

- git merge branch Merge branch branch into the current branch and commit the result. This command is idempotent and can be run as many times as needed to keep the current branch up-to-date with changes in branch. git merge branch –no-commit Merge branch branch into the current branch, but do not autocommit the result. Allows for inspection or tweaking of the merge result before committing. git merge branch –squash –commit Merge branch branch into the current branch as a single commit.

Undoing

Reverting is different from resetting in that reverts usually create new history while resets usually remove existing history. The changes of a revert are applied to the current state of the repository, and, if committed, results in a new repository state descending from the current one. Reverts are safe to publish even if they revert a previously published commit, and, in fact, are the correct way of dealing with the undoing of published commits. Resetting, on the other hand, represents (a possibly selective) “rewind” to a previous state in the history “starting again” from there. Resets should never be committed if they undo commits that have been published or pushed to remote repositories, as this would result in invalid object histories and commit ID’s in the remote repositories.

Reverting

- git revert rev Revert the changes introduced by rev, and record a new commit that records it. This does not do the same thing as similarly named commands in other VCS’s such as “svn revert” or “bzr revert”. git checkout path(s) Re-checkout file or files specified by path(s), overwriting any local changes. This is most similar to “svn revert”. git checkout – path(s) As above, but use this syntax if you have a branch or tag with the same name as a path given in path(s). git checkout rev path(s) Re-checkout file or files specified by path(s) to version specified by rev (which may be specified using a SHA1 commit ID, branch name, or tag), overwriting any local changes. git checkout -f Throw away all local changes since last commit, restoring working tree to last committed state (plus untracked files) and clearing index. Unlike “git reset –hard”, does not move HEAD, and so will not, for example, cleanly forget about a failed merged: use “git reset –hard” for this.

Resetting

- git reset Resets the index (i.e., removes all changes staged for commit) but does not modify the working tree (i.e., the changes in the files are preserved), and does not change HEAD. git reset rev – path(s) Restores file or files specified by path(s) to revision

specified by rev, without changing HEAD.

- git reset rev Sets the current HEAD to the commit specified by rev (which may be specified using a SHA1 commit ID, branch name, or tag), and resets the index but not the working tree (i.e current changes in the working tree are preserved). git reset –soft rev Sets the current HEAD to the commit specified by rev, and does not modify the working tree, but keeps changes in the index for editing. For example, if something was forgotten or omitted in the previous commit, “git reset –soft HEAD^” will undo the last commit, and keep all changes in the index for editing and the next commit. git reset –hard Throw away all local changes since last commit, restoring working tree to last committed state (plus untracked files) and resetting both index and HEAD. git reset –hard rev Sets the current HEAD to the commit specified by rev, and changes the working tree to mirror the new HEAD (plus untracked files). For example, “git reset –hard ORIG_HEAD” will undo the most recent successful merge and any changes that occurred after. Useful for forgetting about the merge just done. If there are conflicts (the merge was not successful), use “git reset –hard” instead.

Stashing

Use “git stash” when you want to record the current state of the working directory and the index, but want to go back to a clean working directory. The command saves your local modifications away and reverts the working directory to match the HEAD commit.

- git stash save [msg] Save your local modifications to a new stash, and run “git reset –hard” to revert them. This is the default action when no subcommand is given. If msg is not explicitly given, then it defaults to “WIP on branch” where “branch” is the current branch name.
- git stash list List all current stashes.
- git stash apply [stash] Restore the changes recorded in the stash on top of the current working tree state. When no stash is given, applies the latest one (stash@{0}). The working directory must match the index. git stash pop [stash] Remove a single stashed state from the stash list and apply on top of the current working tree state. When no stash is given, the latest one (stash@{0}) is assumed. git stash clear Remove all the stashed states. git stash drop [stash] Remove a single stashed state from the stash list. When no stash is given, it removes the latest one. i.e.

stash@{0}. git stash branch new-branch [stash] Creates and checks out a new branch named new-branch starting from the commit at which the stash was originally created, applies the changes recorded in stash to the new working tree and index, then drops the stash if that completes successfully. When no stash is given, applies the latest one.

Cleaning

- git clean -f Remove all untracked files from working copy.
- git clean -fd Remove all untracked files and directories from working copy.
- git clean -fX Remove all ignored files from working copy.
- git clean -fXd Remove all ignored files and directories from working copy.
- git clean -fx Remove all untracked and ignored files from working copy.
- git clean -fxd Remove all untracked and ignored files and directories from working copy.

Remotes

- git remote add remote url Adds a remote named remote for the repository at url.
- git rm remote url Remove reference to remote repository named remote: all tracking branches and configuration settings for remote are removed.
- git push remote :heads/branch Delete the branch branch from the remote repository named remote.
- git remote prune remote Prune deleted remote branches from git branch listing. These branches have already been removed from the remote repository named remote, but are still locally available in “remotes/remote”.

Plumbing

test sha1-A = $(git merge-base sha1-B) Determine if merging sha1-B into sha1-A is achievable as a fast forward; non-zero exit status is false.

Configuration

You can add “–global” after “git config” to any of these commands to make it apply to all git repositories (writes to ~/.gitconfig).

git config user.email author@email.com Set email for commit messages. git config user.name ‘author name’ Set name for commit messages. git config branch.autosetupmerge true Tells git-branch and git-checkout to setup new branches so that git-pull(1) will appropriately merge from that remote branch. Recommended. Without this, you will have to add “–track” to your branch command or manually merge remote tracking branches with “fetch” and then “merge“. Environment Variables

GIT_AUTHOR_NAME, GIT_COMMITTER_NAME Full name to be recorded in any newly created commits. Overrides user.name in .git/config. GIT_AUTHOR_EMAIL, GIT_COMMITTER_EMAIL Email address to be recorded in any newly created commits. Overrides user.email in .git/config. No related posts.

출처:[http://zackperdue.com/tutorials/super-useful-need-to-know-git-commands]

Sidebar Navigation

Summary: github 사용법에 대한 문서

github 사용법

Travis CI 사용법

Summary: Travis CI 사용법에 대한 문서

Travis CI 사용법

Release notes 0.1

Summary: Version 0.1 of the Documentation

Relative links

2017-02-09 merge version