수정하고 저장소에 저장하기

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

워킹 디렉토리의 모든 파일은 크게 Tracked(관리대상임)와 Untracked(관리대상이 아님)로 나눈다.

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

마지막 커밋 이후 아직 아무것도 수정하지 않은 상태에서 어떤 파일이 수정되면 Git은 그 즉시 파일을 Modified 상태로 인식한다. 그리고 이 수정한 파일을 Stage 하고 Staged 상태인 파일을 커밋한다. 이 라이프 사이클을 계속 반복한다.



파일의 상태 확인하기

파일의 상태를 확인하려면 보통 git status 명령을 사용한다.

Clone 한 후에 바로 이 명령을 실행하게 되면 아래와 같은 메시지를 볼 수 있다.

$ git status

On branch master

nothing to commit, working directory clean

위의 내용은 파일을 하나도 수정하지 않았다는 것을 말해준다. Tracked나 Modified 상태인 파일이 없다는 의미이다. Untracekd 파일은 아직 없어서 목록에 나타나지 않는다.

그리고 현재 작업 중인 브랜치를 알려준다.

기본 브랜치가 master이기 떄문에 현재 master로 나오는 것이다. 브랜치 관련 내용은 차차 알아가자, 다음 장에서 브랜치와 레퍼런스에 대해 자세히 다룬다.

프로젝트에 README 파일을 만들어보자, README 파일은 새로 만든 파일이기 떄문에 git status를 실행하면 Untracked files에 들어있다.

$ vim README

$ git status

On branch master

Untracked files:

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

README

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

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

파일을 새로 추적하기

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

$ git add README

git status 명령을 다시 실행하면 README 파일이 Tracked 상태이면서 Staged 상태라는 것을 확인 할 수 있다.

$ git status

On branch master

Changes to be committed:

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

new file: README

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

Modified 상태의 파일을 Stage 하기

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

$ git status

On branch master

Changes to be committed:

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

new file: README

Changes not staged for commit:

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

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

modified: benchmarks.rb

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

$ vim benchmarks.rb

$ git status

On branch master

Changes to be committed:

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

new file: README

modified: benchmarks.rb

Changes not staged for commit:

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

modified: benchmarks.rb

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

$ vim benchmarks.rb

$ git status

On branch master

Changes to be committed:

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

new file: README

modified: benchmarks.rb

Changes not staged for commit:

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

modified: benchmarks.rb

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

$ git add benchmarks.rb

$ git status

On branch master

Changes to be committed:

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

new file: README

modified: benchmarks.rb

파일 무시하기

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

$ cat .gitignore

\*.[oa]

\*~

첫 번째 줄은 확장자가 .o나 .a인 파일을 Git이 무시하라는 것이고 둘째 줄은 ~로 끝나는 모든 파일을 무시하라는 것이다, .o와 .a는 각각 빌드 시스템이 만들어내는 오브젝트와 아카이브파일이고 ~로 끝나는 파일은 Emacs나 VI 같은 텍스트 편집기가 임시로 만들어내는 파일이다. 또 log, tmp, pid 같은 디렉토리나, 자동으로 생성하는 문서 같은 것들도 추가할 수 있다.

.gitignore 파일은 보통 처음에 만들어 두는 것이 편리하다. 그래서 Git 저장소에 커밋하고 싶지 않은 파일을 실수로 커밋하는 일을 방지할 수 있다.

.gitignore 파일에 입력하는 패턴은 아래 규칙을 따른다.

* 아무것도 없는 줄이나 #로 시작하는 줄은 무시한다.
* 표준 Glob 패턴을 사용한다.
* 디렉토리는 슬래시를 끝에 사용하는 것으로 표현한다.
* 느낌표로 시작하는 패턴의 파일은 무시하지 않는다.

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

다음은 .gitignore 파일의 예이다.

# a comment - 이 줄은 무시한다.

# 확장자가 .a인 파일 무시

\*.a

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

!lib.a

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

/TODO

# build/ 디렉토리에 있는 모든 파일은 무시한다.

build/

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

doc/\*.txt

# `doc` 디렉토리 아래의 모든 .txt 파일을 무시한다.

doc/\*\*/\*.txt

\*\*/ 스타일의 문법은 Git1.8.2 버전부터 사용할 수 있다.

Staged와 Unstaged 상태의 변경 내용을 보기

단순히 파일이 변경됐다는 사실이 아니라 어떤 내용이 변경됐는지 살펴보기엔 git status 명령이 아니라 git diff 명령을 사용해야 한다. 보통 우리는 수정했지만 아직 Staged 파일이 아닌 것과 어떤 파일이 Staged 상태인지 가 궁금하기 때문에 git status 명령으로도 충분하다.

git diff는 Patch 처럼 어떤 라인을 추가했고 삭제했는지가 궁금할 때에 사용한다. git diff는 나중에 더 자세히 다룬다.

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

$ git status

On branch master

Changes to be committed:

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

new file: README

Changes not staged for commit:

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

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

modified: benchmarks.rb

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

$ git diff

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

index 3cb747f..da65585 100644

--- a/benchmarks.rb

+++ b/benchmarks.rb

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

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

end

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

+ git.commits.size

+ end

+

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

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

log.size

이 명령은 워킹 디렉토리에 있는 거소가 Staging Area에 있는 것을 비교한다. 그래서 수정하고 아직 Stage 하지 않은 것을 보여준다.

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

$ git diff --cached

diff --git a/README b/README

new file mode 100644

index 0000000..03902a1

--- /dev/null

+++ b/README2

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

+grit

+ by Tom Preston-Werner, Chris Wanstrath

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

+

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

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

Benchmarks.rb 파일을 Stage 한 후에 다시 수정해도 git diff 명령을 사용할 수 있다. 이 때는 Staged 상태인 것과 Unstaged 상태인 것을 비교한다.

$ git add benchmarks.rb

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

$ git status

On branch master

Changes to be committed:

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

modified: benchmarks.rb

Changes not staged for commit:

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

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

modified: benchmarks.rb

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

$ git diff

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

index e445e28..86b2f7c 100644

--- a/benchmarks.rb

+++ b/benchmarks.rb

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

main()

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

+# test line

Staged 상태인 파일은 git diff –cached 옵션으로 확인한다.

변경 사항 커밋하기

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

$ git commit

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

편집기는 아래와 같은 내용을 표시한다. (아래 예제는 Vim 편집기)

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

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

# On branch master

# Changes to be committed:

# new file: README

# modified: benchmarks.rb

#

~

~

~

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

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

메시지를 인라인으로 첨부할 수도 있다. Commit 명령을 실행할 떄 아래와 같이 -m옵션을 사용한다.

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

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

2 files changed, 3 insertions(+)

create mode 100644 README

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

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

Staging Area 생략하기

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

$ git status

On branch master

Changes not staged for commit:

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

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

modified: benchmarks.rb

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

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

[master 83e38c7] added new benchmarks

1 files changed, 5 insertions(+)

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

파일을 삭제하기

GIt에서 파일을 제거하려면 git rm 명령으로 Tracked 상태의 파일을 삭제한 후에 (정확하게는 Staging Area에서 삭제하는 것) 커밋해야 한다. 이 명령은 워킹 디렉토리에 있는 파일도 삭제하기 때문에 실제로 지워진다. 만약 Git 없이 그냥 파일을 삭제하고 git status 명령으로 상태를 확인하면 Changes not staged for commit 즉 (Unstaged)에 속한다는 것을 확인할 수 있다.

$ rm grit.gemspec

$ git status

On branch master

Changes not staged for commit:

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

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

deleted: grit.gemspec

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

그리고 git rm 명령을 실행하면 삭제한 파일은 staged 상태가 된다.

$ git rm grit.gemspec

rm 'grit.gemspec'

$ git status

On branch master

Changes to be committed:

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

deleted: grit.gemspec

커밋하면 파일은 삭제되고 Git은 이 파일을 더는 추적하지 않는다. 이미 파일을 수정했거나 Index에 (역주, Stagine Area를 Get Index라고도 부른다.) 추가했다면 -f옵션을 주어 강제로 삭제해야 한다. 이 점은 실수로 데이터를 삭제하지 못하도록 하는 안전장치이다. 한 번도 커밋한 적 없는 데이터는 Git으로 복구할 수 없다.

또 Stating Area에서만 제거하고 워킹 디렉토리에 있는 파일은 지우지 않고 남겨둘 수 있다. 다시 말해서 하드디스크에있는 파일은 그대로 두고 Git만 추적하지 않게 한다. 이것은.gitignore파일에 추가하는 것을 빼먹었거나 대용량 로그 파일이나 컴파일된 파일인 .a파일 같은 것을 실수로 추가했을 때 뜬다. –cached 옵션을 사용하여 명령을 실행한다.

$ git rm --cached readme.txt

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

$ git rm log/\\*.log

파일 이름 변경하기

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

이렇게 말하고 Git에 mv명령이 있는게 좀 이상하겠지만, 아래와 같이 파일 이름을 변경할 수 있다.

$ git mv file\_from file\_to

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

$ git mv README.txt README

$ git status

On branch master

Changes to be committed:

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

renamed: README.txt -> README

사실 git mv명령은 아래 명령어들을 수행한 것과 완전히 똑같다.

$ mv README.txt README

$ git rm README.txt

$ git add README

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