

Session 4

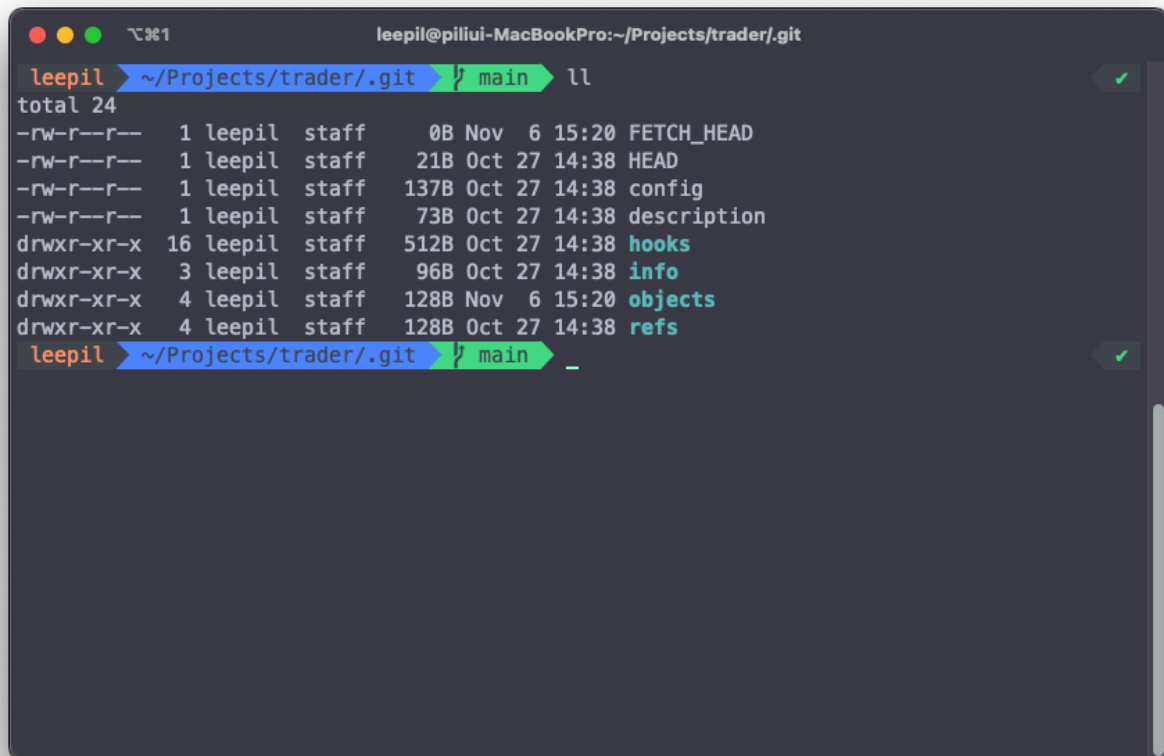
10.1 Plumbing and Porcelain

이제까지는 주로 사용자 친화적인 Porcelain 명령어에 대해 다뤘다면 10장에서는 더 낯것의 명령어들이인 Plumbing 명령어들을 다룬다.

```
$ git init
```

이렇게 빈 git 레포지토리를 생성하게 되면 `.git` 디렉토리가 생긴다. 이 디렉토리가 git이 어떻게 작동하는지에 대한 핵심 내용이다.

새로 생성된 `.git` 디렉토리는 다음과 같이 구성되어 있다.



```
leepil@piliui-MacBookPro:~/Projects/trader/.git
leepil ~/Projects/trader/.git main ll
total 24
-rw-r--r--  1 leepil  staff   0B Nov  6 15:20 FETCH_HEAD
-rw-r--r--  1 leepil  staff  21B Oct 27 14:38 HEAD
-rw-r--r--  1 leepil  staff 137B Oct 27 14:38 config
-rw-r--r--  1 leepil  staff  73B Oct 27 14:38 description
drwxr-xr-x 16 leepil  staff 512B Oct 27 14:38 hooks
drwxr-xr-x  3 leepil  staff  96B Oct 27 14:38 info
drwxr-xr-x  4 leepil  staff 128B Nov  6 15:20 objects
drwxr-xr-x  4 leepil  staff 128B Oct 27 14:38 refs
leepil ~/Projects/trader/.git main _
```

눈여겨 볼 것들은 HEAD와 index 파일 그리고 objects, ref 디렉토리 이렇게 4개이다.

Tip

index 파일은 아직 생성되지 않은 상태이다.

10.2 Git Objects

Object

git은 본질적으로 key-value 쌍으로 이루어진 자료 저장 공간이다. 원하는 어떤 파일이든, git 레포지토리에 넣게 되면 git은 그 파일에 대응하는 key 값을 반환해준다. 아래와 같이 `git hash-object` 명령어를 사용하면 `.git/object` 디렉토리에 파일이 저장되고, key를 반환받을 수 있다.

```
$ echo 'test content' | git hash-object -w --stdin
d670460b4b4aece5915caf5c68d12f560a9fe3e4
```

이제 `find` 명령어로 파일이 저장된 것을 확인할 수 있다.

```
$ find .git/objects -type f
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
```

위의 예시를 자세히 보면 `d6` 라는 디렉토리 내부에 `70460...` 이라는 파일이 존재하는 걸 볼 수 있는데, 이는 해시값의 앞 2글자를 git이 디렉토리로 만들어 사용하기 때문이다.

해당 파일의 내용을 보고 싶다면 `git cat-file -p` 명령어를 사용하면 된다.

```
$ git cat-file -p d670460b4b4aece5915caf5c68d12f560a9fe3e4
test content
```

그럼 이제 버전을 관리하는 방법을 차근차근 알아보자. 우선 첫 번째 버전의 파일을 생성해서 git 저장소에 저장한다.

```
$ echo 'version 1' > test.txt
$ git hash-object -w test.txt
83baae61804e65cc73a7201a7252750c76066a30
```

그리고 두 번째 버전을 작성해 다시 저장한다.

```
$ echo 'version 2' > test.txt
$ git hash-object -w test.txt
1f7a7a472abf3dd9643fd615f6da379c4acb3e3a
```

이제 로컬 파일 시스템에서 `test.txt` 를 삭제해도, git 저장소에서 파일을 복구할 수 있다.

```
$ rm test.txt
$ git cat-file -p 83baae > test.txt
$ cat test.txt
version 1
```

이렇게 파일명은 저장하지 않고 내용만 저장하는 파일을 `blob` 타입 파일이라고 한다.

```
$ git cat-file -t 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a
blob
```

Tree Object

git에서는 **tree**를 사용해 이전에 `blob` 파일에서 존재했던 파일명이 없는 문제를 해결하고, 여러 파일들을 유의미한 단위로 묶어서 관리할 수 있다. git에서 파일을 저장하는 방법은 UNIX 파일 시스템과 유사한데 아래와 같이 대응된다.

- `tree`: 디렉토리
- `blob`: 각 파일의 내용

아래와 같은 `tree`를 가진 git 레포지토리가 있다고 생각해보자

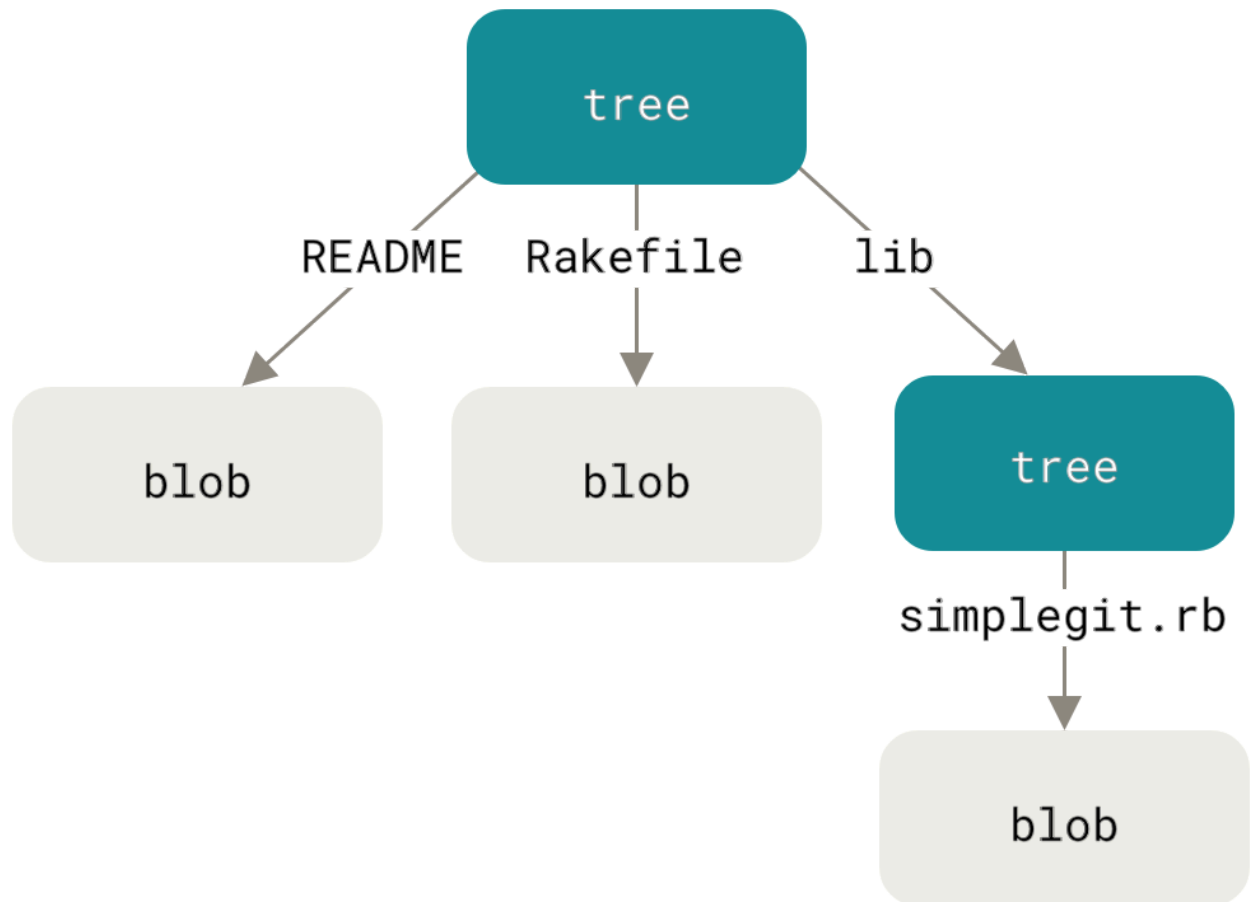
```
$ git cat-file -p master^{tree}
100644 blob a906cb2a4a904a152e80877d4088654daad0c859    README
100644 blob 8f94139338f9404f26296befa88755fc2598c289    Rakefile
040000 tree 99f1a6d12cb4b6f19c8655fca46c3ecf317074e0    lib
```

Tip

`master^{tree}` 는 현재 `master` 브랜치가 가르키고 있는 커밋의 `tree object`를 선택하는 방법이다.

`lib`의 경우에는 하위 디렉토리를 나타내는데 `blob`이 아니라 `tree`로 나오는 것을 볼 수 있다. 그림으로 보게 된다면 아래와 같은 상태이다.

```
$ git cat-file -p 99f1a6d12cb4b6f19c8655fca46c3ecf317074e0
100644 blob 47c6340d6459e05787f644c2447d2595f5d3a54b    simplegit.rb
```



`tree`는 일반적으로 `index`의 상태를 바탕으로 생성된다. 따라서 `tree`를 생성하고 싶다면 파일들을 먼저 staging 시켜야 한다. 이전의 예시에서 생성한 `test.txt`의 첫 번째 버전 `blob` 파일을 `git update-index` 명령어를 이용해 등록해보자.

```
$ git update-index --add --cacheinfo 100644 \
  83baae61804e65cc73a7201a7252750c76066a30 test.txt
```

이제 `git write-tree` 명령어를 이용해 `index`에 있는 내용을 토대로 `tree`를 생성할 수 있다.

```
$ git write-tree
d8329fc1cc938780ffdd9f94e0d364e0ea74f579
$ git cat-file -p d8329fc1cc938780ffdd9f94e0d364e0ea74f579
100644 blob 83baae61804e65cc73a7201a7252750c76066a30      test.txt
```

새로운 `new.txt` 파일을 만들고, 기존에 만들었던 `test.txt`의 두 번째 버전도 `index` 파일에 추가해준다.

```
$ echo 'new file' > new.txt
$ git update-index --cacheinfo 100644 \
  1f7a7a472abf3dd9643fd615f6da379c4acb3e3a test.txt
$ git update-index --add new.txt
```

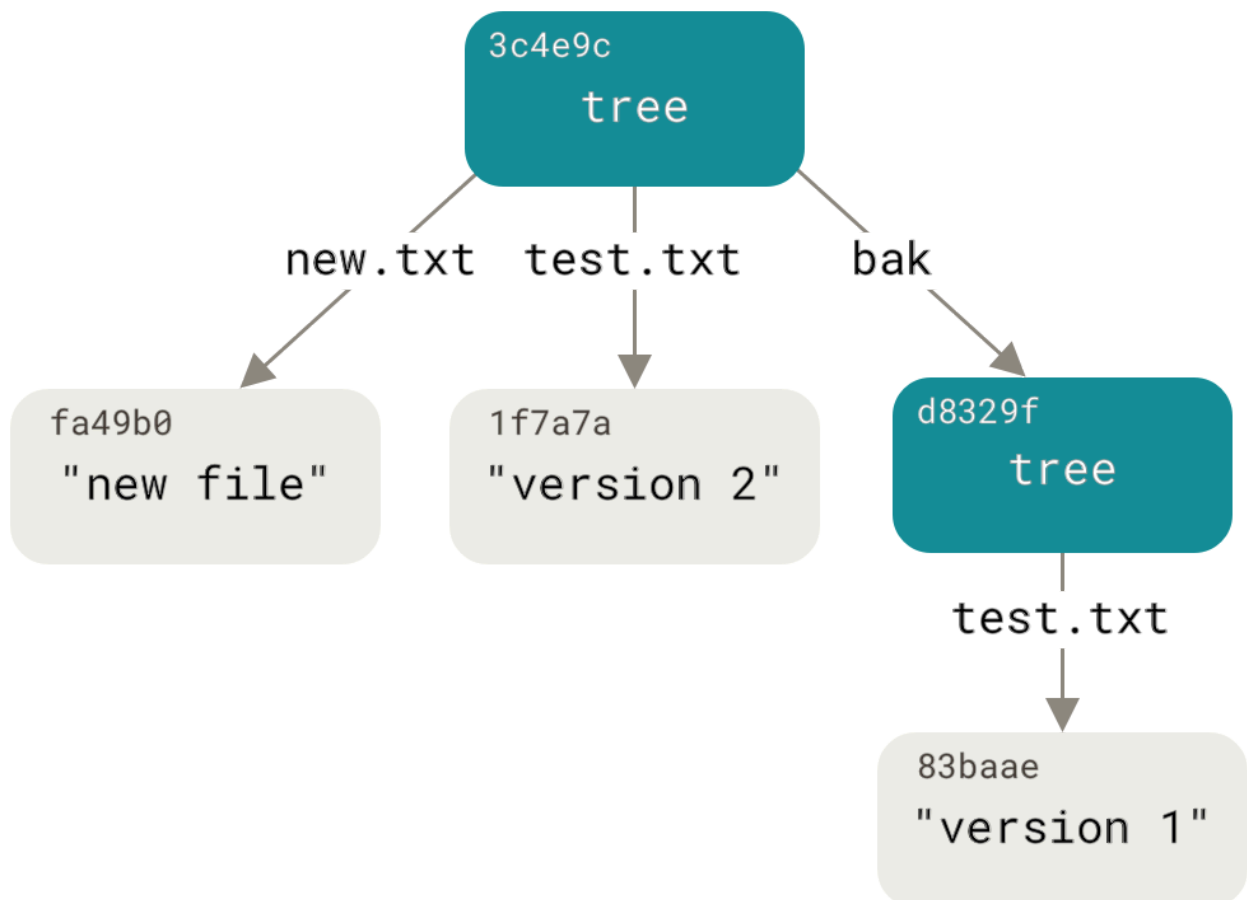
새롭게 생성된 tree에는 두 파일이 존재하는 걸 볼 수 있다.

```
$ git write-tree
0155eb4229851634a0f03eb265b69f5a2d56f341
$ git cat-file -p 0155eb4229851634a0f03eb265b69f5a2d56f341
100644 blob fa49b077972391ad58037050f2a75f74e3671e92    new.txt
100644 blob 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a    test.txt
```

`git read-tree` 명령어를 이용해 index에 tree를 가져올 수 있는데, `--prefix` 옵션을 사용하게 되면 서브디렉토리로 해당 tree를 붙일 수 있다.

```
$ git read-tree --prefix=bak d8329fc1cc938780ffdd9f94e0d364e0ea74f579
$ git write-tree
3c4e9cd789d88d8d89c1073707c3585e41b0e614
$ git cat-file -p 3c4e9cd789d88d8d89c1073707c3585e41b0e614
040000 tree d8329fc1cc938780ffdd9f94e0d364e0ea74f579    bak
100644 blob fa49b077972391ad58037050f2a75f74e3671e92    new.txt
100644 blob 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a    test.txt
```

지금 상태는 아래와 같다.



Commit Objects

이제껏 3개의 tree를 만들었지만, 단 한 개의 커밋도 만들지 않았다. 그 결과 우리는 트리의 내용을 보고 싶으면 각 트리의 SHA-1 값을 기억해야 한다. 누가 tree를 만들었는지, 언제 만들었는지와 같은 부가적인 정보 또한 알 수 없는 상태이다.

`git commit-tree` 명령을 이용해 tree의 내용을 담은 커밋 object를 생성할 수 있다.

```
$ echo 'First commit' | git commit-tree d8329f
fdf4fc3344e67ab068f836878b6c4951e3b15f3d
```

처음으로 생성된 커밋 object의 내용은 다음과 같다.

```
$ git cat-file -p fdf4fc3
tree d8329fc1cc938780ffdd9f94e0d364e0ea74f579
author Scott Chacon <schacon@gmail.com> 1243040974 -0700
committer Scott Chacon <schacon@gmail.com> 1243040974 -0700

First commit
```

이제 이렇게 생성한 첫 번째 커밋을 바탕으로 두 번째, 세 번째 커밋도 `-p` 옵션을 이용해 연결시킬 수 있다.

```
$ echo 'Second commit' | git commit-tree 0155eb -p fdf4fc3
cac0cab538b970a37ea1e769cbbde608743bc96d
$ echo 'Third commit' | git commit-tree 3c4e9c -p cac0cab
1a410efbd13591db07496601ebc7a059dd55cfe9
```

드디어 `1a410e`를 이용해 커밋 기록들을 볼 수 있게 됐다.

```
$ git log --stat 1a410e
commit 1a410efbd13591db07496601ebc7a059dd55cfe9
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri May 22 18:15:24 2009 -0700

    Third commit

    bak/test.txt | 1 +
    1 file changed, 1 insertion(+)

commit cac0cab538b970a37ea1e769cbbde608743bc96d
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri May 22 18:14:29 2009 -0700

    Second commit

    new.txt | 1 +
```

```
test.txt | 2 +-
2 files changed, 2 insertions(+), 1 deletion(-)

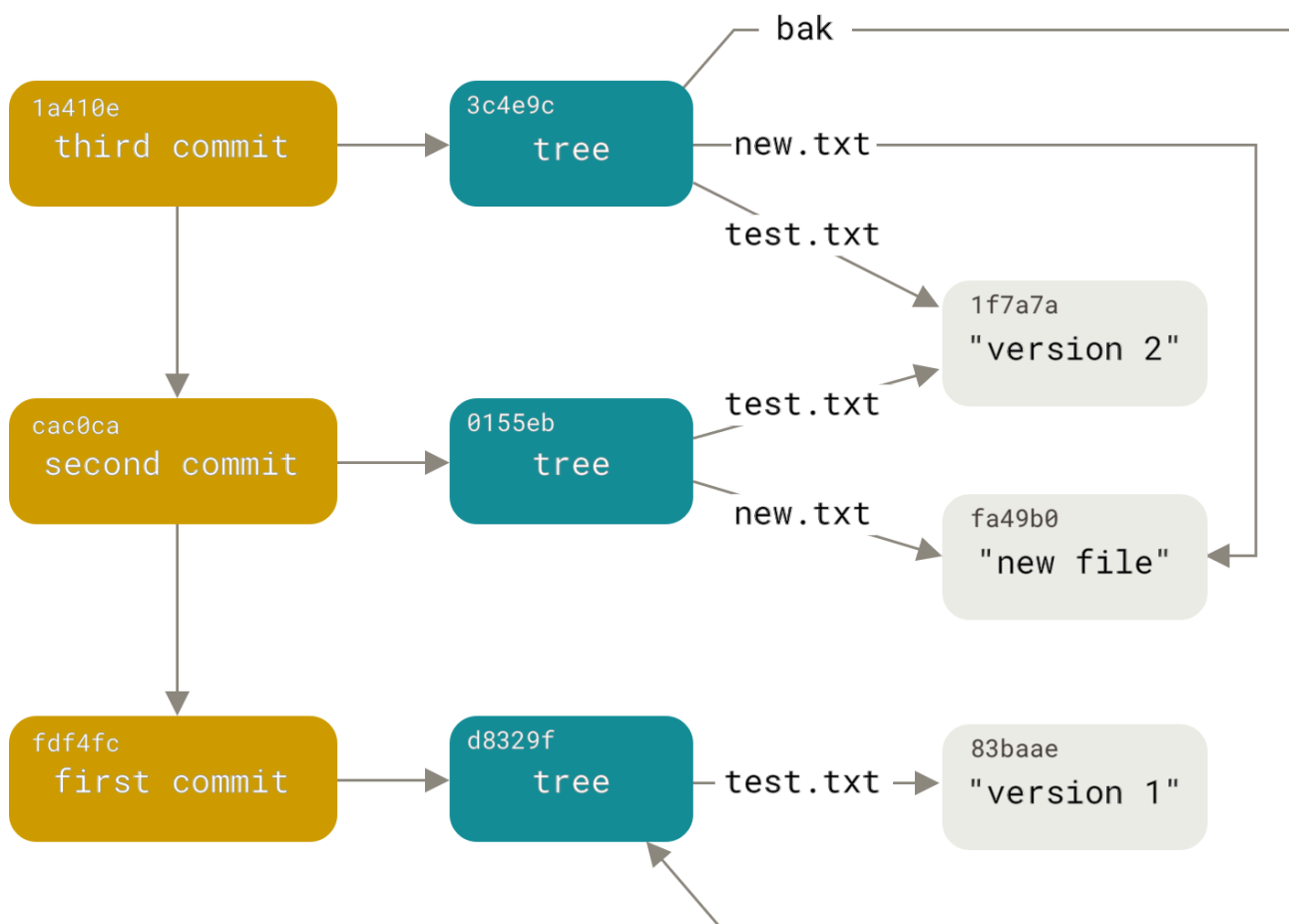
commit fdf4fc3344e67ab068f836878b6c4951e3b15f3d
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri May 22 18:09:34 2009 -0700

    First commit

test.txt | 1 +
1 file changed, 1 insertion(+)
```

git 내부에서는 우리가 `git add`와 `git commit` 명령어를 사용할 때마다 이런 방식으로 새로운 커밋을 생성하고 연결시켜준다.

이제까지 우리가 수동으로 만든 오브젝트들은 아래와 같이 연결되어 있다.



Object Storage

git이 object를 어떤 식으로 저장하는지 보여준다. blob object의 경우에는 파일의 크기와 파일의 내용으로 SHA-1 해시값을 생성하고, 해당 해시값의 경로에 압축 파일을 저장한다.

10.3 Git References

Git References

이전에 수동으로 커밋 object 들을 만들고 기록을 볼 때, `git log 1a410e` 처럼 해시값을 통해 기록을 볼 수 있었다. ref는 해시를 기억하지 않고 기억하기 쉬운 이름으로 object에 접근할 수 있도록 만들어준다.

ref 또는 reference라고 불리는데 작동 원리는 간단하다. `.gits/refs` 디렉토리에 각 파일명마다 가르키고 싶은 SHA-1 해시값을 저장하면 된다.

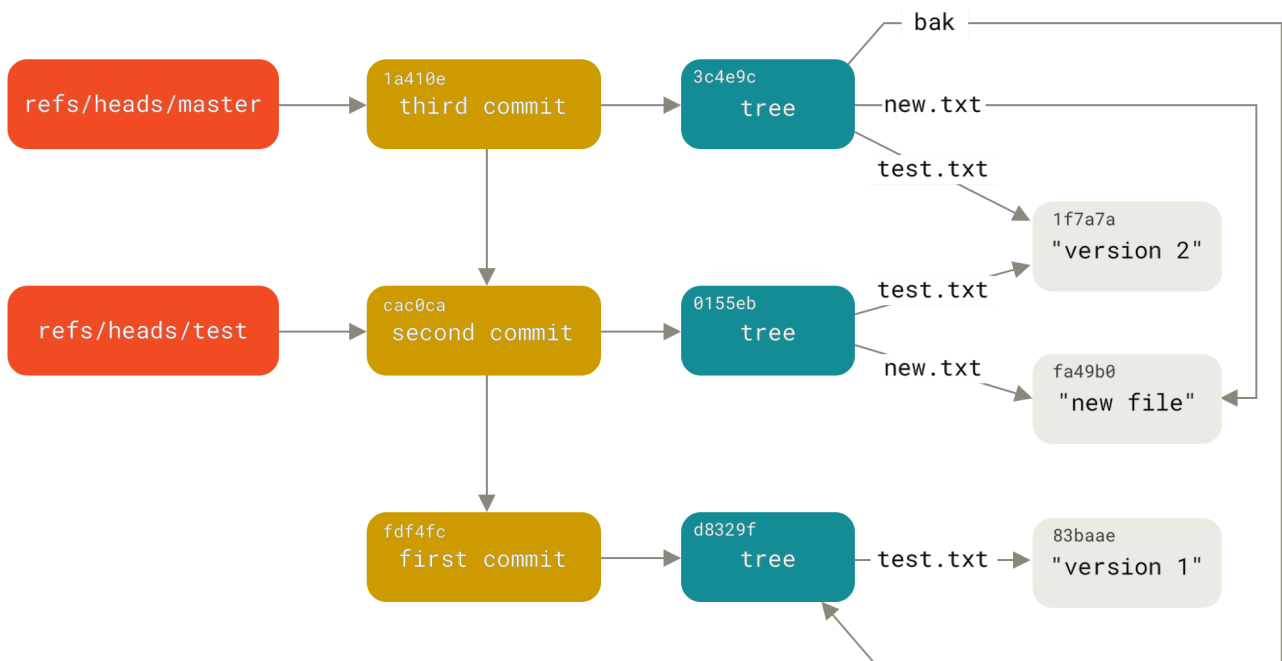
`git update-ref` 명령어를 이용해 원하는 ref를 직접 생성할 수 있다.

```
$ git update-ref refs/heads/master 1a410efbd13591db07496601ebc7a059dd55cfe
$ git update-ref refs/heads/test cac0ca
```

이제 `master` 나 `test` 를 이용해 원하는 커밋에 접근할 수 있다.

```
$ git log --pretty=oneline test
cac0cab538b970a37ea1e769cbbde608743bc96d Second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d First commit
```

현재 git 데이터베이스의 상태는 다음과 같다.



`git branch` 명령어나, `git checkout -b` 명령어를 사용하면 git은 `update-ref` 명령어를 이용해 ref를 추가한다.

The HEAD

`git branch <branch>` 명령어를 실행하면, 어떻게 git은 마지막 커밋의 SHA-1을 알 수 있을까? 바로 HEAD 파일을 이용하는 것이다. HEAD 파일은 다른 ref의 주소를 가르킨다.

`git commit` 명령어를 입력하면, `git commit-tree`가 실행될 때, HEAD의 ref를 참조해 새로 생기는 커밋의 부모 커밋을 결정하게 된다.

`git symbolic-ref` 명령어를 이용해 HEAD의 값을 변경할 수 있다.

Tags

마지막으로 살펴볼 object는 tag object이다. 커밋 object와 유사하게 생성 시각, 만든 사람, 메시지등을 저장하지만 ref처럼 커밋 object를 가르키고, 한 번 결정되면 수정되지 않는다.

이전에 배웠던 lightweight tag는 다음과 같이 생성할 수 있다.

```
$ git update-ref refs/tags/v1.0 cac0cab538b970a37ea1e769cbbde608743bc96d
```

만약 내용을 담은 annotated tag를 생성한다면

```
$ git tag -a v1.1 1a410efbd13591db07496601ebc7a059dd55cfe9 -m 'Test tag'
```

lightweight tag와 다르게 내용이 object가 된다.

```
$ cat .git/refs/tags/v1.1
9585191f37f7b0fb9444f35a9bf50de191beadc2
```

그리고 이 object의 내용이 tag의 내용이 된다. 신기하게도 tag인데 파일의 type이 커밋이다.

```
$ git cat-file -p 9585191f37f7b0fb9444f35a9bf50de191beadc2
object 1a410efbd13591db07496601ebc7a059dd55cfe9
type commit
tag v1.1
tagger Scott Chacon <schacon@gmail.com> Sat May 23 16:48:58 2009 -0700

Test tag
```

Remotes

원격 레포지토리에 대한 참조인 remote도 ref에서 관리한다. `git remote add` 명령어를 이용해 원격 레포지토리를 등록하고 `git fetch`를 이용해 원격 레포지토리의 정보들을 가져오게 되면 `refs/remotes` 디렉토리에 원격 레포지토리의 브랜치 정보들이 저장된다.

Tip

원격 레포지토리의 정보는 `.git/config` 파일에 저장된다. 뒤쪽의 Refspec에서 자세히 다룬다.

```
[remote "origin"]
  url = git@github.com:ksg1227/NetworkProgramming_teamProject.git
  fetch = +refs/heads/*:refs/remotes/origin/*
[branch "main"]
  remote = origin
  merge = refs/heads/main
```

예를 들어 `.git/refs/remotes/origin/master`의 값을 보게 되면, 마지막으로 원격 레포지토리와 연결했을 때 `master` 브랜치가 가르키는 커밋의 값을 가지고 있게 된다. `refs/remotes`에 있는 값들은 전부 read-only이다.

```
$ cat .git/refs/remotes/origin/master
ca82a6dff817ec66f44342007202690a93763949
```

10.4 Packfiles

git에서 파일들을 효율적으로 관리하기 위해 사용하는 방법이다. 원격 레포지토리에 푸시하거나, `git gc` 명령어를 이용하면 packfile이 생성된다. packfile이 생성되기 전에는 object들이 여러 포인터들에 의해 느슨하게 묶여 있다가 packfile이 생성되면 하나의 파일로 묶여진다.

`git gc` 명령어를 실행하고 나면, `.git/objects`에 남아있는 object는 어떤 커밋에도 포함되어 있지 않는 blob들 뿐이다.

```
$ find .git/objects -type f
.git/objects/bd/9dbf5aae1a3862dd1526723246b20206e5fc37
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
.git/objects/info/packs
.git/objects/pack/pack-978e03944f5c581011e6998cd0e9e30000905586.idx
.git/objects/pack/pack-978e03944f5c581011e6998cd0e9e30000905586.pack
```

`.pack` 파일은 `.git/objects`에서 삭제된 모든 파일들에 대한 내용을 나눠서 담고 있고, `idx` 파일은 특정 object를 각각의 packfile 내에서 빨리 찾기 위한 offset을 저장한다.

`git verify-pack` 명령어를 이용해 packfile 내부에 어떤 내용들이 담겨 있는지 확인할 수 있다.

특이한 점은 git에서 파일들의 변경사항을 저장할 때, 가장 마지막 버전을 온전한 상태로 packfile에 저장하고 파일의 이전 버전에 대한 내용을 차이점을 통해 저장한다는 것이다. 이 이유는 최신 버전일수록 접근을 자주 하기 때문에 접근 속도를 올리기 위해서이다.

10.5 The Refspec

`git remote add` 명령어를 이용해 원격 레포지토리를 로컬 레포지토리와 연결하게 되면 `.git/config` 파일 내에 값이 다음과 같이 기록된다.

```
[remote "origin"]
  url = https://github.com/schacon/simplegit-progit
  fetch = +refs/heads/*:refs/remotes/origin/*
```

refspec의 형식은 `[+]<src>:<dst>` 이다. `src`는 원격 레포지토리의 내용, `dst`는 로컬 레포지토리의 어디에서 추적할지가 저장된다. `+`가 붙어있으면 fast-forward가 아니라도 reference를 업데이트한다.

만약 `origin`의 모든 브랜치가 아니라 `master` 브랜치의 내용만 받아보고 싶다면

```
fetch = +refs/heads/master:refs/remotes/origin/master
```

이렇게 내용을 수정하면 된다.

콘솔에서 다음과 같이 여러 브랜치에 대해 각각의 refspec에서 값을 받아 올 수 있다.

```
$ git fetch origin master:refs/remotes/origin/mymaster \
    topic:refs/remotes/origin/topic
From git@github.com:schacon/simplegit
! [rejected]        master    -> origin/mymaster (non fast forward)
* [new branch]     topic      -> origin/topic
```

`push` 명령어의 경우에도 별도의 refspec을 가진다. `fetch`와 유사하게 `push = [+]<src>:<dst>` 형식이며 로컬의 레포지토리에서 원격 레포지토리로 정보를 전송하는 것이기 때문에 `src`가 로컬 ref가 된다.

예시로 다음과 같이 설정할 수 있다.

```
[remote "origin"]
  url = https://github.com/schacon/simplegit-progit
  fetch = +refs/heads/*:refs/remotes/origin/*
  push = refs/heads/master:refs/heads/qa/master
```

원격 레포지토리의 ref를 삭제할 때는

```
$ git push origin :topic
```

또는 더 직관적인 명령어인

```
$ git push origin --delete topic
```

를 사용하면 된다.

10.7 Maintenance and Data Recovery

Maintenance

이전에 나왔던 `git gc` 명령어가 하는 일중 하나는 `references` 들을 하나의 파일로 묶어준다는 것이다. 묶인 파일은 `.git/packed-refs` 파일에 저장된다.

```
$ cat .git/packed-refs
# pack-refs with: peeled fully-peeled
cac0cab538b970a37ea1e769cbbde608743bc96d refs/heads/experiment
ab1afef80fac8e34258ff41fc1b867c702daa24b refs/heads/master
cac0cab538b970a37ea1e769cbbde608743bc96d refs/tags/v1.0
9585191f37f7b0fb9444f35a9bf50de191beadc2 refs/tags/v1.1
^1a410efbd13591db07496601ebc7a059dd55cfe9
```

`ref`를 찾을 때는 우선 `.git/refs/heads`를 확인하고 fallback으로 `packed-refs`를 확인해서 찾는다.

Data Recovery

`git reset --hard` 또는 `git branch -D`로 내가 이제까지 열심히 짜놓은 코드가 다 날아갔을 때 어떻게 이 상황을 모면할 수 있을까?

설명을 돕기 위해 현재 `ref`에 5개의 커밋이 있는 상황에서

```
$ git log --pretty=oneline
ab1afef80fac8e34258ff41fc1b867c702daa24b Modify repo.rb a bit
484a59275031909e19aadb7c92262719cfcdf19a Create repo.rb
1a410efbd13591db07496601ebc7a059dd55cfe9 Third commit
cac0cab538b970a37ea1e769cbbde608743bc96d Second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d First commit
```

3번째 커밋으로 초기화를 해 2개의 커밋을 잃어버린 상황을 예시로 들어보자. 우리는 다시 5번째 커밋으로 `tree`를 복구하고 싶다.

```
$ git reset --hard 1a410efbd13591db07496601ebc7a059dd55cfe9
HEAD is now at 1a410ef Third commit
$ git log --pretty=oneline
1a410efbd13591db07496601ebc7a059dd55cfe9 Third commit
cac0cab538b970a37ea1e769cbbde608743bc96d Second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d First commit
```

만약 커밋의 해시값을 기억할 수 있다면 `.git/objects`에 들어있을테니 찾을 수 있겠지만 그걸 실제로 기억하기란 쉽지 않다.

`git reflog` 명령어는 이런 상황을 해결할 수 있는 가장 빠른 방법이다. `git`은 HEAD가 변경될 때마다 그 기록을 기록해두는데 `git reflog` 명령어를 통해 이 기록들을 볼 수 있다.

```
$ git reflog
1a410ef HEAD@{0}: reset: moving to 1a410ef
ab1afef HEAD@{1}: commit: Modify repo.rb a bit
484a592 HEAD@{2}: commit: Create repo.rb
```

`git log -g` 명령어를 입력하면 같은 결과를 더 읽기 쉽게 볼 수 있다.

```
$ git log -g
commit 1a410efbd13591db07496601ebc7a059dd55cfe9
Reflog: HEAD@{0} (Scott Chacon <schacon@gmail.com>)
Reflog message: updating HEAD
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri May 22 18:22:37 2009 -0700

    Third commit

commit ab1afef80fac8e34258ff41fc1b867c702daa24b
Reflog: HEAD@{1} (Scott Chacon <schacon@gmail.com>)
Reflog message: updating HEAD
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri May 22 18:15:24 2009 -0700

    Modify repo.rb a bit
```

이제 우리가 복구할 커밋을 찾았으니 해당 커밋을 이용해 브랜치를 만들면 복구에 성공할 수 있다.

```
$ git branch recover-branch ab1afef
$ git log --pretty=oneline recover-branch
ab1afef80fac8e34258ff41fc1b867c702daa24b Modify repo.rb a bit
484a59275031909e19adb7c92262719cfcdf19a Create repo.rb
1a410efbd13591db07496601ebc7a059dd55cfe9 Third commit
cac0cab538b970a37ea1e769cbbde608743bc96d Second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d First commit
```

그럼 다음으로 의도적으로 `recover-branch`를 삭제하고 `reflog`도 삭제해보자.

```
$ git branch -D recover-branch
$ rm -Rf .git/logs/
```

이때는 `git fsck` 명령어를 사용해 도움을 받을 수 있다. `git database`의 무결성을 검증하는 명령어인데 `--full` 옵션을 붙이면 다른 object와 연결되어 있지 않은 object들을 보여준다.

별도의 plumbing 명령어, 예를 들어 `git hash-object` 를 사용하지 않은 이상 다른 object와 연결이 없다는 얘기는 잃어버린 커밋과 관련된 object일 가능성이 높다.

```
$ git fsck --full
Checking object directories: 100% (256/256), done.
Checking objects: 100% (18/18), done.
dangling blob d670460b4b4aece5915caf5c68d12f560a9fe3e4
dangling commit ab1afe80fac8e34258ff41fc1b867c702daa24b
dangling tree aea790b9a58f6cf6f2804eeac9f0abbe9631e4c9
dangling blob 7108f7ecb345ee9d0084193f147cdad4d2998293
```

이제 여기서 `dangling commit` 인 `ab1afe` 를 이용해 이전과 같이 브랜치를 복구하면 된다.

Tip

책의 막바지에 `git filter-branch` 명령어를 이용해 과거 기록을 전부 수정하는 부분이 나오는데 이런 방법이 있다는 것만 알고 있으면 될 것 같다.

`git prune` 명령어를 이용해 접근이 불가능한 object들을 git database에서 모두 삭제할 수 있는데 이는 `git gc` 명령어에서 자동으로 수행하기 때문에 사용자가 호출할 일은 거의 없다.

scalar

용량이 큰 git 레포지토리를 관리할 때 사용할 수 있는 명령어이다. 작은 프로젝트를 진행할 때는 상관 없지만 당장 [react 레포지토리](#)만 봐도 용량이 너무 커서 clone하는데 한세월이다.

`scalar clone <url> [<enlistment>]` 의 형식으로 원하는 레포지토리를 복사할 수 있다. clone이 끝나면 `worktree`는 `<enlistment>/src` 에 위치한다.

이후에 `git sparse-checkout set <path>` 명령어를 이용해 레포지토리 내에 원하는 디렉토리의 내용만을 로컬로 내려받을 수 있다. [sparse-checkout](#)에 대한 자세한 내용은 [여기를](#) 참고하자.

maintenance

git 레포지토리를 최적화시켜주는 명령어이다. `git maintenance start` 명령어를 통해 레포지토리에 대한 관리를 시작할 수 있다.

하는 일은 다음과 같다.

commit-graph

[커밋 그래프](#) 파일을 점진적으로 업데이트하고 데이터가 정상적으로 포함됐는지 확인해준다.

prefetch

사용자가 실제로 `git fetch` 명령어를 실행하기 전에 미리 필요한 데이터들을 모든 원격 레포지토리로 부터 받아온다.

loose-objects & gc

두 작업 모두 packfile에 등록되어 있지 않은 파일들을 packfile로 묶어준다.

incremental-repack

합칠 수 있는 packfile들을 모아 더 큰 packfile로 만들어 저장공간을 최적화한다.

Summary

git을 사용하면서 많은 문제들이 발생했을 때 "어 이거 어디서 봤는데"라는 생각이 드는 스터디였으면 하는 바람으로 스터디를 마무리합니다. 이제까지 많은 내용을 다뤘는데 여러분들이 앞으로 git을 사용함에 있어서 조금이라도 도움이 되면 좋을 것 같아요.

이전부터 막연하게 git 스터디를 진행해보겠다고 마음은 먹었었는데 처음 진행하다 보니 진행이 원활하지 못했음에도 불구하고 모두 함께 공부하시느라 고생 많으셨습니다 :)