4주차

session_4.pdf

Session 4

- 10.1 Plumbing and Porcelain
- 10.2 Git Objects
- 10.3 Git References
- 10.4 Packfiles
- 10.5 The Refspec
- 10.7 Maintenance and Data Recovery
- gc
- scalar
- maintenance

10.1

Plumbing

- 저수준 명령어
- 직접 커맨드라인에서 실행하기보다 새로운 도구를 만들거나 각자 필요한 스크립트를 작성할 때 사용

Porcelain

- 사용자에게 친숙한 사용자용 명령어
- `checkout`, `branch`, `remote` 등

10.2

Git 개체

- Git은 Key-Value(ex. 파일 이름과 파일 데이터) 데이터 저장소

  =어떤 형식의 데이터라도 집어넣을 수 있고 해당 Key로 언제든지 데이터를 다시 가져올 수 있음

- `git hash-object` : 주어지는 데이터를 저장, 이 데이터에 접근하기 위한 key를 반환(`-w` 옵션을 줘야 실제로 저장, `--stdin` 옵션을 주면 표준입력으로 입력되는 데이터를 읽음)
- `git cat-file` : 저장한 데이터를 불러오기(`-p` 옵션을 주면 파일 내용이 출력됨)

Tree 개체

- 디렉토리 구조를 저장하기 위한 개체
- Git은 일반적으로 Staging Area(Index)의 상태대로 Tree 개체를 만들고 기록함. 따라서, Tree 개체를 만들려면 우선 Staging Area에 파일을 추가해서 Index를 만들어야 함

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

- Staging Area를 Tree 개체로 저장할 때는 `git write-tree` 명령을 사용
- `git cat-file` 명령으로 이 개체가 Tree 개체라는 것을 확인

Commit 개체

- `commit-tree` 명령으로 생성. 이 명령에 커밋 개체에 대한 설명과 Tree 개체의 SHA-1 값 한 개를 넘김

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

10.3

Git Refs

- Refs(레퍼런스)는 커밋을 가리키는 포인터 역할을 함
- SHA-1 해시 값(커밋 ID)을 직접 기억하거나 사용하는 대신, 사용자가 이해하기 쉬운 이름으로 커밋을 참조할 수 있음
- `.git/refs` 디렉토리에 저장됨

HEAD

- HEAD 파일은 현 브랜치를 가리키는 간접(symbolic) Refs
- 이 Refs는 다른 Refs를 가리키는 것이라서 SHA-1 값이 없음

태그

- 누가, 언제 태그를 달았는지 태그 메시지는 무엇이고 어떤 커밋을 가리키는지에 대한 정보가 포함
- 브랜치처럼 커밋 개체를 가리키지만 옮길 수는 없음
- Lightweight 태그=단순히 커밋을 가리키는 Refs
- Annotated 태그=메타데이터(작성자, 메시지, 시간 등)를 포함하는 태그 개체

리모트

- 리모트 Refs=리모트를 추가하고 Push 하면 Git은 각 브랜치마다 Push 한 마지막 커밋이 무엇인지 `refs/remotes` 디렉토리에 저장

10.4

Packfile

- Packfile=여러 Git 개체(Blob, Tree, Commit, Tag 등)를 압축하여 하나의 파일로 저장하는 형식
- `git gc` : Packfile이 생성 명령

10.5

Refspec

- Refspec=Git에서 리모트와 로컬 간에 Refs를 매핑하는 규칙
- `$ git push origin master:refs/heads/qa/master`

  =`master` 브랜치를 리모트 저장소에 `qa/master` 로 Push
- Refspec으로 서버에 있는 Refs를 삭제 가능

  `$ git push origin :topic` =리모트의 `topic` 브랜치를 삭제

`$ git gc --auto` : Git이 Garbage를 Collect 할 지 말지 자동으로 판단해서 처리

10.7→(data recovery쪽 중요→reflog,fsck)

`master` 브랜치에서 강제로(Hard) Reset 한 경우

```
$ git reflog                  # HEAD의 변경 이력 확인
$ git branch recover-branch SHA값  # 복구 브랜치 생성
```

Reflog 삭제된 경우

```
$ git fsck --full             # Dangling 커밋 확인
$ git branch 복구브랜치 SHA값  # 복구 브랜치 생성
```

개체 삭제

누군가 매우 큰 바이너리 파일을 넣어버리면 Clone 할 때마다 그 파일을 내려받음→Clone 시 부담 발생

`git verify-pack` : 파일과 그 크기 정보를 수집

`git rev-list --objects` : 파일 이름과 SHA-1 확인

`filter-branch` : 히스토리에서 완전히 삭제

gc

`git gc` : 로컬 Git 저장소의 불필요한 파일을 정리하고 최적화하는 명령어

기능

- 파일 압축: 파일의 여러 버전을 압축하여 디스크 공간 절약.
- Unreachable Object 삭제: 사용되지 않는 개체 제거.
- Refs 압축: 참조를 `.git/packed-refs` 파일로 병합하여 효율성 증대.
- Reflog 및 기타 메타데이터 정리: 오래된 Reflog 항목 및 stale 상태의 작업 디렉토리 제거.
- Commit-Graph 업데이트: 저장소 성능을 높이는 부가적 인덱스 생성/갱신

scalar

`scalar` : 대규모 Git 저장소를 관리하기 위한 도구. Git의 성능을 향상시키고 유지보수를 단순화함.

주요 기능=Git 설정을 최적화, 백그라운드에서 저장소를 유지 관리, 네트워크 데이터 전송을 줄임

Clone

- 저장소를 클론

List

- 등록된 enlistment 목록을 표시

Register

- 지정된 enlistment를 Scalar에 등록하고 백그라운드 유지보수 시작

Unregister

- 등록된 enlistment를 Scalar에서 제거하고 백그라운드 유지보수를 중지

Run

- 특정 유지보수 작업(또는 전체 작업)을 실행. 일반적으로 자동으로 실행되므로 수동 실행은 드물게 필요함

Reconfigure

- Scalar를 업그레이드한 후, 또는 설정이 손상된 경우 enlistment를 재구성

Diagnose

- 문제 보고를 위해 현재 enlistment의 상태 및 로그를 포함한 진단 데이터를 수집

Delete

- enlistment를 파일 시스템에서 삭제하고 Scalar에서 등록 해제

maintenance

`git maintenance` : Git 저장소의 데이터를 최적화하여 Git 명령어 속도를 높이고 저장 공간 요구사항을 줄임

Run

- 지정된 최적화 작업을 실행

Start

- 현재 저장소에 대해 백그라운드 유지보수를 시작

Stop

- 백그라운드 유지보수 스케줄을 중단

Register

- 현재 저장소를 유지보수 대상으로 등록하고 추천 설정을 활성화