Git, GitHub

📚🔗 **Reference and Quick**

> - 참조 URL
> - GitHub 치트 시트 : https://education.github.com/git-cheat-sheet-education.pdf
> - Pro Git : https://git-scm.com/book/ko/v2
> - 터미널 명령 참조 : https://github.com/0nn0/terminal-mac-cheatsheet/blob/master/한국어/README.markdown

> - Conventional commits

분류	의미
feat	새로운 기능을 구현했을 때(서비스의 기능이 수정되는 것이라면 모두 포함, 문구 수정도 포함)
fix	기능에는 수정 사항이 없고 버그가 수정되었을 때
perf	서비스나 라이브러리의 성능을 개선했을때
refactor	기능 추가도 없고, 버그 수정도 없는 단순 리팩터링
test	테스트를 추가하거나 기존에 있는 테스트를 수정했을 때
build	빌드 시스템이나 npm 배포에 대한 수정
ci	CI 설정이 수정되었을 때(Jenkins, Travis 등)
chore	그 외 실제 코드에는 영향이 없는 단순 수정

> - global .gitignore 만들기 gitignore 생성 url : https://www.toptal.com/developers/gitignore/
> - `$ git config --global core.excludesfile ~/global_gitignore`

> - **태그 생성 및 푸시** GitHub의 [Tags] 탭에서 확인할 수 있고, [Release] 탭에서 다운받을 수 있다는 것!

```
$ git tag -a -m "첫 번째 태그 생성" v0.1  # 주석 있는 태그 생성
$ git push origin v0.1  # 태그 푸시
```

Table of Contents

Git, GitHub 시작하기

Chapter 0 : 빠른 실습으로 Git, GitHub 감 익히기

- 로컬저장소에서 커밋 관리하기

    1. 로컬 저장소 만들기

    ```
    $ git init
    ```

    Output

    ```
    Initialized empty Git repository in {path}
    ```

    2. 첫번째 커밋 만들기

        1. 정보 등록

        ```
        $ git config --global user.email "star2kis@nate.com"
        $ git config --global user.name "KangHwan-Cha"
        ```

        2. 파일 추가

        ```
        $ git add README.md
        또는
        $ git add .
        ```

        3. 커밋하기

        ```
        $ git commit -m "My first commit"
        ```

- 다른 커밋으로 시간여행하기

    1. `git log`: 커밋 확인
        - git log 명령은 최신 커밋부터 보여줌
    2. `git checkout {커밋 ID}`
    3. `git checkout -`: 최근에 있던 브랜치로 이동

> 최근 `switch` / `restore`명령어로 나누어짐

> 1. `switch`: 브랜치 간 이동
> 2. `restore`: 커밋에서 파일들을 복구

- GitHub 원격 저장소에 커밋 올리기

  1. 레포지토리: 원격 저장소

  2. 원격저장소 만들기

     > Git 레포지토리

  3. 원격 저장소 url: https://github.com/KangHwan-Cha/Study_Git.git

  4. **★ 원격 저장소에 커밋 올리기**

```
# 원격 저장소 주소 입력
$ git remote add origin https://github.com/KangHwan-
Cha/Study_Git.git
# 브랜치 만들기
$ git branch -M main
# 원격 저장소에 올리기
$ git push origin main
```

- GitHub 원격 저장소의 커밋을 로컬 저장소에 내려받기

  1. 클론clone: 코드와 버전 전체를 내려받기

     > [Download ZIP] 으로 받으면 원격 저장소와 버전정보가 제외되므로 `git clone`을 사용

```
# 주소 뒤에 한칸 띄고 마침표
# 마침표를 붙이면 현재 위치에 풀어서 clone
$ git clone {원격 저장소 주소} .
```

  2. 원격 저장소의 새로운 커밋을 로컬 저장소에 갱신하기

```
$ git pull origin main
```

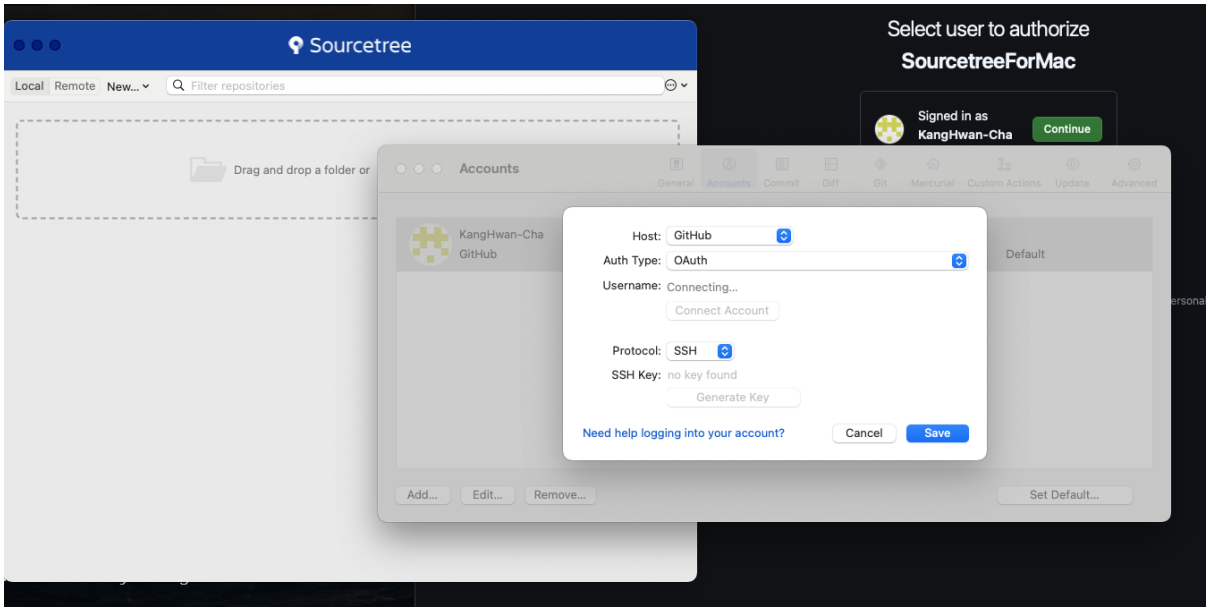
- 단어정리
  - **Git**: 분산 버전 관리 시스템으로, 파일 변경 이력 관리 및 협업을 위한 도구.
  - **GitHub**: Git을 기반으로 한 웹 서비스로, 소스 코드 호스팅 및 협업 기능을 제공.
  - **GUI** Graphical User Interface : 그래픽 사용자 인터페이스로, 명령어 대신 그래픽 요소로 시스템과 상호작용하는 방식.
  - **CLI** Command Line Interface: 명령어를 입력해 시스템과 상호작용하는 텍스트 기반 인터페이스.
  - **Git Bash**: Git을 명령어 기반으로 사용할 수 있게 해주는 터미널 프로그램. Git과 Bash 쉘을 지원.
  - **commit**: Git에서 파일의 변경 사항을 저장하는 단위. 각 커밋은 고유한 ID를 가짐.

- **log**: Git에서 커밋 이력을 확인할 수 있는 명령어. `git log` 명령어를 사용해 커밋 내역을 볼 수 있음.
- **checkout**: 특정 브랜치나 커밋으로 작업 공간을 변경하는 Git 명령어. `git checkout <브랜치명>`으로 사용.
- **워킹트리**working tree: 작업폴더 [.git]폴더를 뺀 나머지 부분이 워킹트리.
- **로컬 저장소**: 사용자의 컴퓨터에 저장된 Git 레포지토리로, 소스 코드와 이력을 포함. [.git]폴더가 로컬 저장소
- **원격 저장소**: GitHub, GitLab 등 외부 서버에 호스팅된 Git 레포지토리로, 협업을 위한 공유 공간.
- **repository**: 프로젝트의 소스 코드 및 변경 이력을 저장하는 Git 저장소. 로컬과 원격 저장소가 있음.
- **push**: 로컬 저장소의 변경 사항을 원격 저장소로 전송하는 Git 명령어. `git push`를 사용.
- **pull**: 원격 저장소에서 변경 사항을 로컬 저장소로 가져오는 Git 명령어. `git pull`을 사용.

Chapter 1 : GUI를 위한 버전 관리 환경 구축하기

- 소스트리 설치하기

1. Sourcetree(Click)



- 비주얼 스튜디오 코드 설치하기

> 이미 사용하고있으므로 Pass

- GitHub 둘러보기

> 책 참조

Chapter 2 ~ Chapter 3 : GIT GUI With. SourceTree

- 브랜치 : 줄기를 나누어 작업할 수 있는 기능
- HEAD : 브랜치 혹은 커밋을 가리키는 포인터
- 브랜치 작성 순서

1. 브랜치 생성
2. 생성된 브랜치 이동
3. 브랜치에서 커밋

4. 코딩이 완료되면 브랜치 병합
5. 개발 완료된 브랜치 삭제

> - 보통 하나의 개발 브랜치에는 **한 사람만 작업**해서 올리는 것이 바람직
> - 원격 저장소에는 미리 브랜치 규칙을 정하는 것이 일반적
>   - 규칙 예시
>   1. `main` 브랜치에는 직접 커밋을 올리지 않는다 (동시에 작업하다 꼬일 수 있으니).
>   2. 기능 개발을 하기 전에 `main` 브랜치를 기준으로 새로운 브랜치 를 만든다.
>   3. 이 브랜치 이름은 `feature/기능이름` 형식 으로 하고 한 명만 커밋을 올린다.
>   4. `feature/기능이름` 브랜치에서 기능 개발이 끝나면 `main` 브랜치에 이를 합친다.

- 병합 커밋 merge commit
- 빨리 감기 fast-forward
- 충돌 conflict
- 풀 리퀘스트pull request : 브랜치를 합치는 예의 바른 방법
  - 협력자에게 브랜치 병합을 요청하는 메시지를 보내는 것
  - 1. 수락 Accept
  - 2. 수정 요청 Request change
  - 3. 병합 Merge pull request
  - 패치(Fetch) : Git에서 새로운 이력을 업데이트

    > `Pull`: 실제 코드를 내려 받음 `Fetch`: 그래프만 업데이트
- 릴리즈 : 개발이 완료되었습니다. 출시하자!
  - 프로그램을 출시하는 것
  - 참고 - LTS(Long Time Support) 일반적인 버전보다 장기간에 걸쳐 지원하도록 특별히 만들어진 버전
  - **VERSION**
    - ver $1^{a)}$.$0^{b)}$.$0^{c)}$

      a) : Major - 사용자들이 크게 느낄 변화 b) : Minor - 작은 변화 등 c) : Maintenance - 버그나 유지보수 등 작은 수정
  - **태그**: 특정 커밋에 포스트잇 붙이기

Chapter 4 : 둘 이상의 원격 저장소로 협업하기

- **포크**fork : 다른 사람의 원격 저장소를 내 계정의 원격 저장소로 복사해 오는 것

명령	의의	편리한 점	불편한 점

명령	의의	편리한 점	불편한 점
브랜치	하나의 원본 저장소에서 분기를 나눈다.	하나의 원본 저장소에서 코드 커밋 이력을 편하게 볼 수 있다	다수의 사용자가 다수의 브랜치를 만들면 관리하기 힘들다.
포크	여러 원격 저장소를 만들어 분기를 나눈다.	원본 저장소에 영향을 미치지 않으므로 원격 저장소에서 마음껏 코드를 수정할 수 있다.	원본 저장소의 이력을 보려면 따로 주소를 추가해야 한다.

- **리베이스**rebase : 묵은 커밋을 새 커밋으로 이력 조작하기

  > - 커밋의 베이스를 똑 떼서 다른 곳으로 붙이는 것
  > - ★ 다른 개발자가 이 변경 사항을 사용하고 있지 않아야 함
  > - *히스토리를 강제로 조작하기 때문에 완전히 꼬일 수 있음*

  - **upstream** : 원본 저장소를 지칭하는 관용적 닉네임
  - **패치**fetch : 원본 저장소에 있는 커밋 히스토리를 받아오는 것

Chapter 5 : 실무 사례와 함께 Git 다루기

- **어멘드**amend : 수정 못한 파일이 있어요, 방금 만든 커밋에 추가하고 싶어요
  - `amend last commit`
- 원격 저장소의 마지막 커밋 수정하고 강제 푸시하기
  - 마지막 커밋 수정 후
  - 강제 푸시

```
$ git push origin main --force
```

- **체리 픽**cherry-pick : 저 커밋 하나만 떼서 지금 브랜치에 붙이고 싶어요

  > 문제가 발생한 커밋만 바로 수정

  - 브랜치 전략 예시

브랜치명	특징
feat/기능이름	- 각 개발자가 개발 중인 브랜치 - 직접 커밋을 올림
main	- [feat/기능이름] 브랜치에서 개발 완료된 코드가 합쳐진 브랜치 - 출시 전의 베타 버전 - 집적 커밋을 올리지 않음(병합을 통해서만 코드를 업데이트)
latest	- 실제 출시할 코드(대중에게 보여줄 완벽한 코드)를 올리는 브랜치 - [main] 브랜치에서 굵직한 개발이 끝나면 출시 시점에 [latest] 브랜치로 코드를 병합

- **리셋**reset : 옛날 커밋으로 브랜치를 되돌리고 싶어요

> 이전 커밋으로 상태 되돌리기

1. `Mixed` 모드 : 원하는 커밋으로 브랜치를 되돌리면서도 변경 사항은 커밋하기 전 상태
2. `Soft` 모드 : 변경 사항을 스테이지에 두어 **다시 당장 커밋이 가능**
3. `Hard` 모드 : 이력을 깔끔하게 과거로 되돌림

- ★ **리버트**revert : 이 커밋의 변경 사항을 되돌리고 싶어요

> 새로운 커밋을 추가해 커밋을 되돌리는 것

  - 잘못된 커밋이 있다면 언제든지 리버트 기능으로 돌리기!

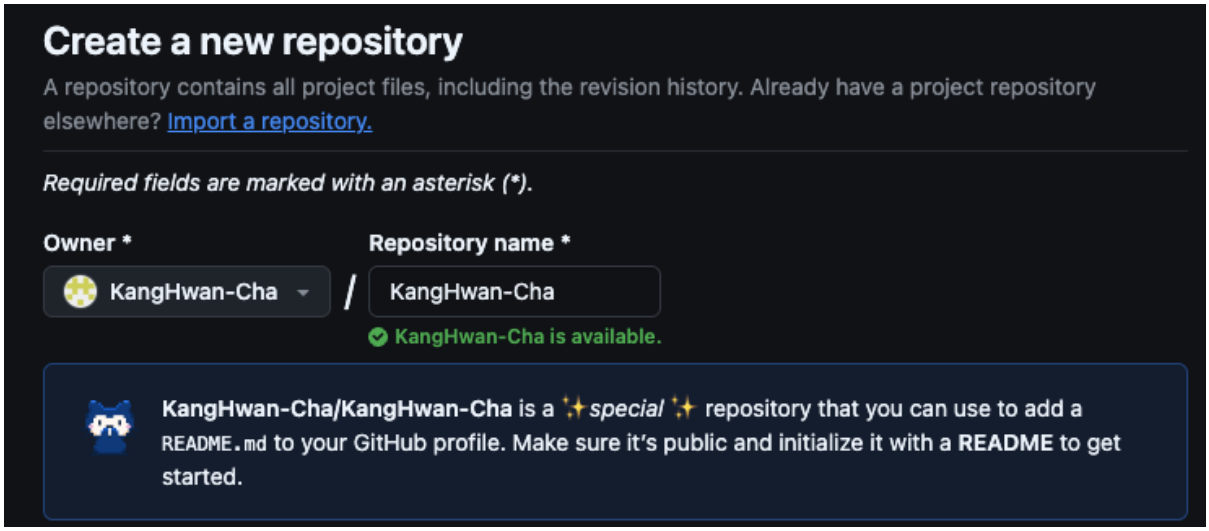
- ★ **스태시**stash : 변경 사항을 잠시 다른 곳에 저장하고 싶어요, 커밋은 안 만들래요

> - 커밋하기 전 파일을 임시로 저장
> - 스태시에는 tracked 상태(추적중 - 한 번이라도 Git에 올렸던 상태)인 파일만 가능

Chapter 6 : GitHub 100% 활용하기

GitHub의 계정 및 프로필 가이드

- REAME 꾸미기 기능
  - 깃 아이디로 레포지토리 생성



  - 나의 레포지토리의 readme에 마크다운 문법으로 업데이트
  - `your prifile`에 들어가면 업데이트를 확인할 수 있음
  - Curr Issue : 뱃지Badge 붙이기
    - Shields.io 활용

- 더 좋은 풀 리퀘스트 만들기
  - 1. 의미를 담은 제목 짓기

> Conventional commits 참조

분류	의미

분류	의미
feat	새로운 기능을 구현했을 때(서비스의 기능이 수정되는 것이라면 모두 포함, 문구 수정도 포함)
fix	기능에는 수정 사항이 없고 버그가 수정되었을 때
perf	서비스나 라이브러리의 성능을 개선했을때
refactor	기능 추가도 없고, 버그 수정도 없는 단순 리팩터링
test	테스트를 추가하거나 기존에 있는 테스트를 수정했을 때
build	빌드 시스템이나 npm 배포에 대한 수정
ci	CI 설정이 수정되었을 때(Jenkins, Travis 등)
chore	그 외 실제 코드에는 영향이 없는 단순 수정

    - 2. 풀 리퀘스트를 병합하는 세 가지 방법
        - 1. 병합 커밋 생성Create a merge commit
        - 2. 스쿼시해서 병합Squash and merge

            > 히스토리가 한줄로 남는다는 장점이 있음

        - 3. 리베이스해서 병합Rebase and merge

Chapter 7 : CLI 환경에서 Git 명령어 살펴보기

- 용어정리
    - **워킹트리** : 일반적인 작업이 일어나는 곳
    - **로컬 저장소** : .git 폴더, 커밋은 로컬 저장소에 저장
    - **원격 저장소** : GitHub 저장소, 로컬 저장소를 업로드하는 곳
    - **Git 저장소** : 엄밀하게는 로컬 저장소, 넓은 의미로 작업 폴더(워킹트리 + 로컬저장소)를 의미하기도 함
- Git 저장소 초기화하기

```
$ git init -b main # main 브랜치 초기화 및 git 저장소 생성
$ git config --global user.name <이름>
$ git config --global user.name  # 설정된 이름 확인
$ git config core.editor  # 설정된 기본 에디터 확인
$ git config --global user.email <이메일 주소>
$ git config --global color.ui auto  # Git Bash 창의 Git 컬러가 자동으로 설정
```

- Git 명령어 옵션 **우선순위**

> local지역 > global전역 > system시스템

- 좋은 커밋 메시지의 7가지 규칙
    - 제목과 본문을 빈 행으로 구분합니다.
    - 제목을 50글자 이내로 제한합니다.

  - 제목의 첫 글자는 대문자로 작성합니다.
  - 제목에는 마침표를 넣지 않습니다.
  - 제목은 명령문으로(영어로 쓸 경우 동사원형(현재형)으로 시작)
  - 본문의 각 행은 72글자 내로 제한합니다.
  - 어떻게보다는 무엇과 왜를 설명합니다.

- remote와 push: 원격 저장소 등록하고 커밋 업로드하기

```
$ git remote add origin <git url>  # 원격 저장소 등록, 통상 첫 번째 원격 저장소를 origin
$ git remote -v  # 원격 저장소 목록
$ git push -u origin main  # 푸시와 동시에 업스트림 지정
```

Chapter 8 : CLI 환경에서 브랜치 생성 및 조작하기

- 브랜치를 사용하는 다섯가지 경우

1. **새로운 기능 추가:** `feature/operation1`
   - 기능별로 브랜치를 생성하면 작업 내용을 메인 브랜치와 분리하여 독립적으로 개발할 수 있습니다.
   - 메인 브랜치의 안정성을 유지하면서 새로운 기능을 테스트하고 구현할 수 있습니다.
2. **버그 수정:** `hotFix` **또는** `bugFix`
   - 긴급한 문제를 해결하거나 버그를 수정할 때, 독립된 브랜치를 사용하면 메인 브랜치에 바로 영향을 주지 않고 안정적인 코드를 유지할 수 있습니다.
   - 수정 후 문제없이 동작하는지 확인한 뒤 병합합니다.
3. **병합과 리베이스 테스트**
   - 브랜치를 사용하면 병합(Merge)과 리베이스(Rebase) 과정을 안전하게 실험할 수 있습니다.
   - 충돌을 미리 확인하고 해결할 수 있어 병합 전략을 개선할 수 있습니다.
4. **이전 코드 개선**
   - 기존 코드를 리팩토링하거나 최적화 작업을 진행할 때, 별도의 브랜치에서 안전하게 작업할 수 있습니다.
   - 이렇게 하면 새로운 문제가 발생해도 원래 코드를 보호할 수 있습니다.
5. **특정 커밋으로 돌아가고 싶을 때**
   - 특정 시점의 코드 상태로 돌아가고 싶을 경우, 해당 커밋을 기반으로 브랜치를 생성하면 현재 작업에 영향을 주지 않고 원하는 코드 상태를 복구하거나 실험할 수 있습니다.

- **태그 생성 및 푸시**

> GitHub의 [Tags] 탭에서 확인할 수 있고, [Release] 탭에서 다운받을 수 있다는 것!

```
$ git tag -a -m "첫 번째 태그 생성" v0.1  # 주석 있는 태그 생성
$ git push origin v0.1  # 태그 푸시
```

- **트리에서 뻗어나온 가지 없애기**

Chapter 9 : Git 내부 동작 원리

```
my-project/           # Git 저장소의 루트 디렉토리 (프로젝트 폴더)
├── .git/             # Git 데이터를 저장하는 숨김 디렉토리
│   ├── HEAD          # 현재 체크아웃된 브랜치를 나타냄
│   ├── config        # 저장소의 Git 설정 정보
│   ├── description   # Bare 저장소용 설명 파일 (일반적으로 비어 있음)
│   ├── hooks/        # 커밋, 푸시 등의 이벤트에 실행되는 스크립트 디렉토리
│   ├── info/         # 추가 정보를 저장하는 디렉토리 (예: exclude 파일)
│   ├── objects/      # Git의 모든 데이터(커밋, 블롭 등)를 해시 값으로 저장
│   │   ├── info/     # 객체 데이터의 추가 정보
│   │   └── pack/     # 데이터 압축 및 패킹 파일
│   ├── refs/         # 브랜치와 태그 정보
│   │   ├── heads/    # 로컬 브랜치 정보
│   │   └── tags/     # 태그 정보
│   ├── logs/         # 브랜치 및 HEAD의 히스토리 로그
│   │   ├── HEAD      # HEAD 변경 기록
│   │   └── refs/     # 브랜치별 히스토리 로그
│   └── index         # 스테이징 영역(인덱스)의 상태 정보
├── 파일1             # 실제 프로젝트 파일
├── 디렉토리1/        # 실제 프로젝트 디렉토리
└── 파일2             # 실제 프로젝트 파일
```

- **구체적인 폴더 및 파일 설명**
  - `.git/` 디렉토리 Git이 모든 데이터(커밋, 브랜치, 태그 등)를 관리하는 핵심 디렉토리입니다. 이 디렉토리가 있는 폴더는 Git 저장소로 간주됩니다.
  - `.git/HEAD` 현재 체크아웃된 브랜치의 참조를 나타냅니다.
    예: `ref: refs/heads/main`
  - `.git/config` 해당 저장소의 Git 설정 파일입니다.
    예: 원격 저장소 정보, 사용자 설정 등이 포함됩니다.
  - `.git/hooks/` Git 이벤트 발생 시 자동으로 실행되는 스크립트를 저장하는 디렉토리입니다.
    예: `pre-commit`, `post-merge` 등의 훅(hook) 파일.
  - `.git/objects/` Git의 데이터 저장소로, 커밋, 블롭(blob), 트리(tree) 등을 해시로 관리합니다.
  - `.git/refs/` 브랜치와 태그의 참조 정보를 저장합니다.
  - **`heads/`**: 로컬 브랜치 정보
  - **`tags/`**: 태그 정보
  - `.git/logs/` 브랜치와 `HEAD`의 변경 내역을 기록합니다.

    - `.git/index` 현재 스테이징된 파일 상태를 저장하는 파일입니다.
    - `.git/info/` Git 저장소의 특정 설정을 저장합니다.
      예: `.gitignore` 외 추가적으로 제외할 파일을 지정할 수 있는 `exclude` 파일.

- git status로 clean한 상태
    - 워킹트리 = 스테이지 = HEAD
- staged 상태
    - 워킹트리 = 스테이지 != HEAD
- **Git 작동원리 핵심**

    1. `git add`
        - 워킹 트리(Working Tree)의 변경 내용을 **스테이징 영역(Staging Area)** 에 반영.
        - 커밋에 포함할 파일과 변경 사항을 선택하여 준비하는 단계.
    2. `git commit`
        - **스테이징 영역**의 내용을 기반으로 **트리(Tree) 객체**를 생성.
        - 생성된 트리 객체를 이용해 기존 **HEAD 커밋**을 부모로 하는 새로운 커밋을 만듦.
        - 커밋 메시지와 함께 변경 내역이 로컬 저장소 히스토리에 추가.
    3. **HEAD 업데이트**
        - 새롭게 생성된 커밋이 현재 **HEAD**가 가리키는 위치가 됨.
        - HEAD는 기본적으로 현재 브랜치의 최신 커밋을 참조.

Git CLI 중급

Chapter 1 : 중급 CLI 명령어 1

- 원격저장소 관련 명령어(원격 저장소 추가로 등록하기)

명령어
`git remote -v`
`git remote add <저장소이름> <url>`
`git remote rename <이전이름> <새이름>`
`git remote remove <저장소이름>`

- GitHub 저장소 백업하기

1. 원격저장소에 백업

```
$ git remote add backup <url>
$ git checkout main
$ git push backup --all  # main 브랜치만 push
$ git push backup --all  # feature1 브랜치도 push

$ git push backup --tags  # tag push

$ git bracn -rv  # 저장소 상태 확인
```

2. 다른 Git 호스팅 서비스에 백업

    1. AWS CodeCommit <-서울에 데이터 센터가 있어서 빠름
    2. BitBucket
    3. GitLab 등

- clean과 hard reset

    - git clean : 워킹트리 정리

명령어	설명
`git clean -f -d <파일 또는 폴더 이름>`	untracked 상태의 파일들을 삭제. 파일 또는 경로를 지정하지 않은 경우 모든 untracked 파일들을 삭제

> 주의 ! 삭제되면 복구가 불가능함

    - `reset --hard`

구분	soft	mixed(기본)	hard

구분	soft	mixed(기본)	hard
현재 브랜치(HEAD)	초기화	초기화	초기화
스테이지	남아있음	초기화	초기화
워킹트리의 변경사항	남아있음	남아있음	초기화

> **reset은 공통적으로 untracked 파일은 건드리지 않음**

- hard reset의 복구

```
$ git reset --hard <reset --hard 하기 전 메모해둔 체크섬을 이용>
```

> **로컬 저장소의 커밋은 없어지지 않음(사라진 것 처럼 안 보일 뿐)**

- **reflog로 사라진 커밋 되살리기**(reset --hard 실수 대처법)

> 현재까지의 모든 로컬저장소의 작업에 대해 커밋 체크섬을 볼 수 있음 git log 명령으로 확인 할 수 없는 커밋을 확인할 수 있음 **★ 단, reflog의 유효범위는 로컬저장소임!!!**

명령어	설명
`git reflog [-n숫자]`	reflog를 보여주는 명령 -n 숫자 옵션으로 객수를 제한해서 볼 수 있음

```
$ git reflog  # reflog 명령 수행
```

Output

```
efb2b3c (HEAD -> main, origin/main) HEAD@{0}: commit: Add Git command
summaries and branch naming guidelines
ebceb84 HEAD@{1}: checkout: moving from hotfix to main
ebceb84 HEAD@{2}: checkout: moving from main to hotfix
```

```
$ git reset --hard HEAD@{2}  # HEAD@{2}를 이용해서 hard reset
```

Chapter 2 : 중급 CLI 명령어 2

- 커밋 수정하기(commit --amend)
  - HEAD가 가르키는 커밋, 즉 현재 커밋을 수정하고 싶은 경우
  - 커밋 메시지를 바꾸고 싶은 경우
  - 이전 커밋의 파일 내용을 조금 수정하고 싶은 경우

- 깜빡 잊고 이전 커밋에 일부 중요 파일을 추가하지 않은 경우

1. commit 메시지 수정하기

```
$ git commit --amend -m "마지막 커밋 메시지 수정"
```

2. commit에 변경 내용 추가하기

```
$ git add .
$ git commit --amend  # 마지막 커밋 수정
```

> git push --force를 사용해야함 git commit --amend 는 **원격저장소에 이미 해당 커밋이 올라가 있는 경우**에는 **가급적이면 사용하지 않는 것이 좋음**

- **참고 : 위험한 명령어들**
  - rebase
  - git push --force
  - reset --hard
  - stash

- git diff로 변경사항 체크하기

명령어	설명
`git diff [a] [b]`	a 저장소를 토대로 b저장소에 추가된 내용의 차이를 보여줌 보통 a는 작업 전, b는 작업 후 저장소 옵션을 생략하면 아직 스테이지에 추가되지 않은 변경사항 단, untracked는 보여주지 않음

1. 스테이징 된 내용과 마지막 커밋과의 비교

```
$ git diff --cached
```

2. 두 커밋의 비교

```
$ git diff HEAD~ HEAD
```

3. git diff --check를 통한 공백 문자 확인

```
$ git diff --check
```

Output

```
master2.txt:2: trailing whitespace.
+space
```

- 참고 : Git이란 도구를 커밋을 중심으로 본다면...
    - 저장소 : 모든 커밋의 모음
    - 브랜치 : 특정 히스토리를 가지는 커밋의 모음
    - 커밋 : 파일의 묶음. Git의 기본 저장 단위
    - 스테이지 : 미래의 커밋
    - 작업 폴더(디렉토리) : 스테이지를 만들어 내기 위한 샌드박스sandbox로 이것저것 파일 내용을 작업하다가 맘에 드는 모양으로 완성되면 스테이지에 추가
    - stash : 임시 진열장

> 사용자 입장에서는 작업 폴더가 제일 중요하지만, **Git의 입장에서는 커밋이 제일 중요함!**

- -X 옵션으로 충돌 해결하기
    - merge에서 theirs / ours 옵션 사용하기

```
$ git merge feature1 -X theirs
$ git merge feature1 -X ours
```

    - rebase에서 theirs / ours 옵션 사용하기

- rebase -i로 커밋 정리하기

> rebase와 rebase -i는 전혀 다른 별개의 명령 **해당 커밋의 후손 커밋만 에디터에 표시** 흑역사 커밋 삭제하기!

    - 커밋들의 순서변경
    - 한 커밋을 다른 커밋과 합치기
    - 커밋 삭제
    - 커밋 메시지 수정
    - 오래된 커밋의 수정 ammend
    - **option**
        1. p pick : 해당 커밋을 변경없이 사용
        2. r reword : 해당 커밋을 사용, 커밋 메시지 편집 가능
        3. e edit : 해당 커밋을 사용, 커밋을 수정(amen)할 수 있도록 일시정지 상태가 됨
        4. s squash : 해당 커밋을 사용, 단 내용은 부모 커밋에 합쳐지고 커밋은 사라진 것 처럼 보임
        5. d drop : 해당 커밋을 제거

명령어	설명

명령어	설명
`git rebase -i <수정하려는 커밋들의 부모 커밋>`	HEAD와 지정한 커밋 사이의 커밋들의 히스토리를 수정 git log 명령과를 목록 순서가 반대

- rebase -i로 커밋 순서 바꾸기

```
$ git rebase -i HEAD~2
```

- 중간의 커밋 메시지 바꾸기

```
$ git rebase -i HEAD~2
# 에디터에서 pick을 reword로 변경
# 다시 에디터가 활성되면 원하는 commit 메시지로 수정
```

- 커밋을 이전 커밋이랑 합치기

```
$ git rebase -i HEAD~2
# 에디터에서 pick을 squash로 변경
# 다시 에디터가 활성되면 원하는 commit 메시지로 수정
```

> **squash 대상 커밋이 이전 커밋에 합쳐지는 것에 주의!**

- cherry-pick으로 커밋 골라먹기

> 다른 브랜치에 있는 특정 커밋을 골라 현재 브랜치에 합칠 때 주로 사용

```
$ git cherry-pick <커밋 체크섬>
```

Chapter 3 : 기타 CLI 명령어

- Git stash로 임시 저장하기

> 변경사항을 커밋하기도, 삭제하기도 모호한 상황에서 주로 사용
>
> - ex) 약간의 변경사항이 있는데 이를 남겨놓고 다른 브랜치를 체크아웃하려는 상황에서 사용

- stash 내용을 다른 브랜치에 적용

```
$ git stash -u
$ git switch -c stash-test
$ git stash pop 또는 apply
```

- 히스토리에서 파일 삭제하기

> 민감 정보가 이미 커밋이나 푸시되었을 때 삭제하는 방법

```
$ git filter-branch -f --tree-fileter 'rm -f file1.txt' HEAD  #
filter-granch 명령 수행
```

- 기타 팁
  - global .gitignore 만들기

> gitignore 생성 url : https://www.toptal.com/developers/gitignore/

```
$ git config --global core.excludesfile ~/global_gitignore
```

Git 명령어 요약

- GitHub 치트 시트 : https://education.github.com/git-cheat-sheet-education.pdf

설정 (Setup)

명령어	설명
`git config --global user.name "[이름]"`	모든 로컬 저장소에 사용할 사용자 이름 설정
`git config --global user.email "[이메일]"`	모든 로컬 저장소에 사용할 이메일 주소 설정
`git config --global color.ui auto`	Git 명령줄의 색상을 자동으로 설정하여 가독성 향상

설정 및 초기화 (Setup & Init)

명령어	설명
`git init`	현재 디렉토리를 새로운 Git 저장소로 초기화
`git clone [URL]`	원격 저장소를 복제하여 로컬에 새 디렉토리 생성

스테이징과 스냅샷 (Stage & Snapshot)

명령어	설명
`git status`	작업 디렉토리의 변경 사항 및 상태 확인
`git add [파일]`	지정한 파일을 스테이징 영역에 추가
`git reset [파일]`	스테이징 영역에서 지정한 파일 제거 (작업 디렉토리의 변경 사항은 유지)
`git reset --hard [파일]`	변경 내용을 완전히 삭제하기
`git diff`	스테이지에 없는 변경 사항 보기
`git diff --staged`	스테이지에 있지만, 커밋되지 않은 변경 사항 보기
`git commit -m "[메시지]"`	스테이징된 변경 사항을 커밋하며, 메시지 추가

브랜치와 병합 (Branch & Merge)

명령어	설명
`git branch`	브랜치 목록 표시 및 현재 브랜치 확인
`git branch [브랜치명]`	현재 커밋에서 새로운 브랜치 생성
`git switch [브랜치명]`	브랜치 변경
`git merge [브랜치명]`	지정한 브랜치의 이력을 현재 브랜치에 병합
`git log`	현재 브랜치의 커밋 이력 표시

이력 수정 (Rewrite History)

명령어	설명
`git rebase [브랜치명]`	현재 브랜치의 커밋을 지정한 브랜치 위로 재배치
`git reset --hard [커밋 해시]`	지정한 커밋으로 작업 디렉토리와 스테이징 영역을 초기화

공유 및 업데이트 (Share & Update)

명령어	설명
`git remote -v`	원격 저장소 목록
`git remote add [별칭] [URL]`	원격 저장소를 별칭으로 추가
`git remote rename <이전이름> <새이름>`	새로운 이름으로 변경
`git remote remove <저장소 이름>`	원격저장소 삭제
`git fetch [별칭]`	원격 저장소의 모든 브랜치 가져오기
`git merge [별칭]/[브랜치명]`	원격 브랜치의 변경 사항을 현재 브랜치에 병합
`git push [별칭] [브랜치명]`	로컬 브랜치의 커밋을 원격 저장소에 푸시
`git pull`	원격 저장소의 변경 사항을 가져와 현재 브랜치에 병합

임시 커밋 (Temporary Commits)

명령어	설명
`git stash <-u>`	수정된 추적 파일을 임시로 저장하여 작업 디렉토리를 깨끗하게 유지 기본적으로 untracked 상태의 파일은 저장하지 않지만 -u 옵션을 함께 사용할 경우 untracked 상태의 파일도 저장
`git stash list`	저장된 스태시 목록 표시
`git stash pop <stash 객체>`	가장 최근의 스태시를 적용하고, 스태시 목록에서 제거
`git stash apply <stash 객체>`	pop과 동일하지만, 스택에 있는 stash 객체를 제거하지 않음 pop보다는 약간 안전한 명령어
`git stash drop <stash 객체>`	가장 최근의 스태시를 스태시 목록에서 제거
`git stash clear`	스택의 모든 stash 객체를 삭제
`git stash branch <stash 객체>`	마지막에 체크아웃된 커밋에 stash 내용을 반영해서 새로운 브랜치를 만듭니다. 테스트를 할 때 의외로 유용하게 사용

비교 및 확인 (Inspect & Compare)

명령어	설명

명령어	설명
`git log`	현재 활성 브랜치의 커밋 이력 표시
`git log [브랜치B]..[브랜치A]`	브랜치B에는 없고 브랜치A에만 있는 커밋 표시
`git log --follow [파일]`	파일의 변경 이력을, 이름 변경 사항을 포함하여 표시
`git diff [브랜치B]...[브랜치A]`	브랜치A에만 있는 변경 사항의 차이점 표시
`git show [커밋 해시]`	지정한 커밋의 상세 정보 표시

경로 변경 추적 (Tracking Path Changes)

명령어	설명
`git rm [파일]`	파일을 삭제하고, 삭제한 사실을 스테이징 영역에 반영
`git mv [현재 경로] [새 경로]`	파일의 경로를 변경하고, 변경 사항을 스테이징 영역에 반영
`git log --stat -M`	파일 경로 변경을 포함한 모든 커밋 로그 표시

패턴 무시 (Ignoring Patterns)

명령어	설명
`git config --global core.excludesfile [파일]`	모든 로컬 저장소에 적용할 전역 .gitignore 파일 설정

브랜치명 작성 가이드

일반적인 브랜치명 규칙

1. **기능(feature) 추가**: `feature/<작업내용>`
2. **버그 수정**: `bugfix/<버그설명>`
3. **핫픽스(hotfix)**: `hotfix/<긴급수정내용>`
4. **실험용(experiment)**: `experiment/<실험내용>`
5. **릴리스 준비(release)**: `release/<버전명>`

브랜치명 예시

기능 개발

- `feature/login-page`
- `feature/add-payment-method`

버그 수정

- `bugfix/fix-login-error`
- `bugfix/correct-typo`

핫픽스

- `hotfix/security-patch`
- `hotfix/fix-critical-bug`

실험 브랜치

- `experiment/new-algorithm`
- `experiment/ui-redesign`

릴리스 브랜치

- `release/v1.0.0`
- `release/v2.1.3`

브랜치 작성 팁

- **소문자와 하이픈 사용**: 예) `feature/add-new-api`
- **팀원 간 합의된 네이밍 컨벤션 사용**: 모든 팀원이 동일한 규칙을 따르도록 합의합니다.
- **짧고 명확하게 작성**: 브랜치명이 지나치게 길어지지 않도록 작업의 핵심 내용을 담습니다.