6. 원격 저장소(GitHub 등) 연동

`git remote` 명령어

> Git 원격 저장소 연결, 관리, 변경에 사용하는 핵심 명령어

✅ 1. `git remote` 개요

Git은 **로컬 저장소**와 **원격 저장소**(예: GitHub, GitLab 등)를 연결해 협업할 수 있다.
`git remote`는 이 **연결된 원격 저장소 목록을 관리하는 명령어**이다.

✅ 2. 기본 명령어 목록

명령어	설명
`git remote`	연결된 원격 저장소 이름 목록 출력
`git remote -v`	원격 저장소의 URL까지 자세히 출력
`git remote add <이름> <URL>`	원격 저장소 추가
`git remote remove <이름>`	원격 저장소 제거
`git remote rename <기존> <새이름>`	원격 저장소 이름 변경
`git remote set-url <이름> <URL>`	원격 저장소 URL 변경
`git remote show <이름>`	해당 원격 저장소 상세 정보 출력

✅ 3. 실전 예제

📌 현재 연결된 원격 저장소 보기

```
git remote
```

```
origin
```

📌 URL까지 출력 (보통 이거 씀)

```
git remote -v
```

```
origin  https://github.com/yourname/project.git (fetch)
origin  https://github.com/yourname/project.git (push)
```

✅ 4. 원격 저장소 추가

```
git remote add origin https://github.com/yourname/project.git
```

> ✅ 일반적으로 `origin`이라는 이름을 많이 씀

✅ 5. 원격 저장소 제거

```
git remote remove origin
```

> 💡 또는 `git remote rm origin` (약식도 됨)

✅ 6. 원격 저장소 이름 변경

```
git remote rename origin upstream
```

- 협업 시 `upstream`은 보통 **원본 저장소**,
  `origin`은 **내 포크(fork)**에 사용함

✅ 7. 원격 저장소 URL 변경

```
git remote set-url origin git@github.com:newuser/newrepo.git
```

> HTTPS → SSH 방식으로 바꿀 때도 이 명령어 사용함

✅ 8. 원격 저장소 정보 보기

```
git remote show origin
```

예시 출력:

```
* remote origin
  Fetch URL: https://github.com/yourname/project.git
  Push  URL: https://github.com/yourname/project.git
  HEAD branch: main
  Remote branches:
    main tracked
  Local branch configured for 'git pull':
    main merges with remote main
```

✅ 9. 자주 묻는 실무 질문

질문	답변
`origin`은 무슨 뜻이야?	최초로 추가한 원격 저장소의 기본 이름
`origin` 외에 이름 지어도 돼?	가능 (예: `upstream`, `github`, `live`)
같은 저장소에 원격 두 개 연결할 수 있어?	가능 (예: `origin`, `backup`)
`-v`는 왜 붙여?	URL까지 보여주기 위해 (verbose)

✅ 10. 사용 예: GitHub 협업 구조

```
# 내 포크 저장소 (origin)
git remote add origin https://github.com/myname/project.git

# 원본 저장소 (upstream)
git remote add upstream https://github.com/original/project.git

# 원본에서 변경사항 가져오기
git fetch upstream
```

✅ 마무리 요약

명령어	설명
`git remote -v`	연결된 원격 저장소 목록 + URL
`git remote add`	새 원격 저장소 연결
`git remote remove`	원격 저장소 제거
`git remote rename`	이름 변경
`git remote set-url`	URL 변경
`git remote show`	상세 정보 확인

`git push`, `git fetch`, `git pull`

> 원격 저장소와의 **전송, 수신, 통합**을 담당하는 핵심 명령어

✅ 개념 먼저 간단 정리

명령어	방향	설명
`git push`	**로컬 → 원격**	내 커밋을 원격 저장소로 전송
`git fetch`	**원격 → 로컬**	원격 변경사항을 받아오되 병합하지 않음
`git pull`	**원격 → 로컬 + 병합**	fetch + 병합까지 자동 수행

✅ 1. `git push`: 원격 저장소에 업로드

```
git push origin main
```

- 로컬의 `main` 브랜치 커밋을 원격(origin)의 `main` 브랜치로 **보냄**
- 기본적으로 **Fast-forward** 방식만 허용
- 강제로 덮어쓰려면: `git push --force-with-lease`

📌 **실무 꿀팁**

- 최초 `push` 전에는 `git push -u origin main` 으로 기본 remote/tracking 설정
- `-u`는 이후 `git push`만 입력해도 되게 함

✅ 2. `git fetch`: 원격 변경사항만 가져오기

```
git fetch origin
```

- **병합은 하지 않고** 원격의 커밋을 로컬 저장소로 다운로드
- 로컬 브랜치는 그대로이고, `origin/main`이라는 **원격 추적 브랜치**만 업데이트됨

예시 흐름

```
git fetch origin
git diff origin/main
git merge origin/main
```

📌 **실무 꿀팁**

- 안전하게 변화 확인 후 병합하고 싶을 때 사용
- `fetch`는 항상 **병합 수동 제어** 가능해서 문제 생길 확률 적음

✅ 3. `git pull`: 가져오고 자동 병합까지

```
git pull origin main
```

- `fetch`와 `merge`를 한 번에 수행
- 자동 병합되며, **충돌이 날 수 있음**

흐름:

```
git pull = git fetch + git merge
```

📌 실무 주의점

- 충돌 발생 시 원인 파악 어려움
- 보통 `pull`보다는 `fetch + merge/rebase`를 권장

✅ 4. `pull` 시 `rebase` 사용하기

```
git pull --rebase
```

- fetch 후 merge가 아니라 **rebase** 방식으로 깔끔한 히스토리 유지
- 협업 시 불필요한 merge 커밋 없이 유지 가능

✅ 5. 시각화 비교

🎯 현재 상황

```
origin/main: A---B---C
local/main : A---B---X---Y
```

🟦 `git push`

```
git push origin main
```

- `X, Y` 커밋을 원격 `main`에 업로드

🟦 `git fetch`

```
git fetch origin
```

- 원격 `C`는 받아오지만 `local/main`에는 영향 없음
- `origin/main`으로만 저장됨

🟦 `git pull`

```
git pull origin main
```

- `origin/main`의 `C`를 `local/main`에 병합
- `A---B---X---Y---M` (M: merge 커밋)

✅ 6. 주요 옵션 정리

명령어	주요 옵션	설명
`push`	`-u`	트래킹 브랜치 설정 (`--set-upstream`)
`push`	`--force-with-lease`	안전한 강제 푸시
`fetch`	`--all`	모든 원격 저장소의 브랜치 가져오기
`pull`	`--rebase`	머지가 아닌 rebase 방식 병합
`pull`	`--ff-only`	fast-forward만 허용 (병합 커밋 방지)

✅ 7. push/pull 충돌 발생 시

- 원격에서 먼저 커밋이 올라왔는데 내가 `push`하려고 하면:

```
! [rejected] main -> main (fetch first)
```

→ 해결법:

```
git pull --rebase
git push
```

✅ 마무리 요약

작업	명령어	특징
원격으로 커밋 보내기	`git push origin main`	로컬 → 원격
원격 커밋 확인만	`git fetch origin`	원격 → 로컬 추적 브랜치
원격 커밋 병합까지	`git pull origin main`	자동 merge 수행
병합 대신 재배치	`git pull --rebase`	깔끔한 히스토리 유지

원격 브랜치 추적

✅ 1. 개념부터 시작

🧠 기본 용어 정의

용어	설명
원격 저장소 (Remote)	GitHub, GitLab 등 서버 저장소 (예: `origin`)
원격 브랜치 (Remote Branch)	원격 저장소에 있는 브랜치 (예: `origin/main`)
추적 브랜치 (Tracking Branch)	**로컬 브랜치가 연결된 원격 브랜치**

> 예: 로컬 `main` 브랜치가 `origin/main`을 추적하면,
> `git pull`, `git push` 명령 시 따로 지정 안 해도 자동 동작함.

✅ 2. 추적 브랜치 설정 방법

📌 `-u` 옵션 사용

```
git push -u origin main
```

- `main` 브랜치를 `origin/main`과 연결해 추적 설정함
- 이후에는 그냥 `git push`, `git pull`만 입력해도 동작

✅ 3. 추적 상태 확인

```
git branch -vv
```

출력 예:

```
* main  a1b2c3d [origin/main] 최신 커밋 메시지
  dev   e4f5g6h [origin/dev]   작업 중
```

- 대괄호 안에 `[origin/브랜치명]`이 추적 중인 원격 브랜치
- `*`는 현재 브랜치

✅ 4. 추적 브랜치 자동 설정

📥 `git clone` 시 기본 설정

```
git clone https://github.com/user/repo.git
```

- `main` 또는 `master` 브랜치가 자동으로 `origin/main`을 추적함

✅ 5. 수동으로 설정 또는 변경하기

🔧 이미 있는 브랜치 추적 설정

```
git branch --set-upstream-to=origin/dev dev
```

- 로컬 `dev` 브랜치가 `origin/dev`을 추적하게 설정

🔧 새 브랜치 생성하며 추적 설정

```
git checkout -b feature-x origin/feature-x
```

- `origin/feature-x`에서 로컬 브랜치 `feature-x` 생성하며 자동 추적

✅ 6. 실전 흐름 예제

🧪 예: 팀원이 만든 `origin/feature-login` 브랜치 가져오기

```
git fetch
git checkout -b feature-login origin/feature-login
```

- `feature-login`이라는 로컬 브랜치가 원격 브랜치를 **추적**
- 이후에는 아래 명령어가 생략 가능:

```
git pull     # 자동으로 origin/feature-login과 통신
git push     # 자동으로 origin/feature-login에 반영
```

✅ 7. 추적 브랜치 삭제/변경 시 주의

🗑️ 원격 브랜치 삭제 후 생기는 현상

```
git fetch -p   # --prune 옵션: 추적 중인 삭제된 원격 브랜치 제거
```

👄 추적 브랜치 변경

```
git branch --unset-upstream
```

→ 현재 브랜치의 추적 대상 해제

✅ 8. Git 설정에서 확인하기

```
git config -l | grep branch
```

출력 예:

```
branch.main.remote=origin
branch.main.merge=refs/heads/main
```

- 현재 브랜치 `main`이 `origin/main`을 추적 중이라는 의미

✅ 마무리 요약

명령어/옵션	설명
`git push -u origin main`	추적 브랜치 설정하며 푸시
`git branch -vv`	추적 상태 확인
`git branch --set-upstream-to=origin/브랜치 브랜치`	수동 설정
`git fetch -p`	삭제된 원격 브랜치 정리
`git checkout -b 로컬 origin/원격`	추적 브랜치 생성

브랜치 푸시/삭제 (`git push origin branch-name`, `--delete`)

> `git push origin <branch>`, `--delete` 명령어 완전 이해

✅ 1. 로컬 브랜치를 원격에 푸시

```
git push origin 브랜치이름
```

📌 예시

```
git push origin feature/login-ui
```

- 로컬의 `feature/login-ui` 브랜치를 원격 저장소(origin)에 푸시함
- 원격에 동일 이름의 브랜치가 생성됨
- **협업 시** 다른 팀원이 `git fetch` 또는 `git checkout`으로 사용 가능해짐

✅ 2. 푸시하면서 추적 브랜치 자동 설정 (`-u`)

```
git push -u origin 브랜치이름
```

- 이후엔 `git push`, `git pull`만 입력해도 자동 연결

예:

```
git push -u origin feature/login-ui
```

✅ 3. 원격 브랜치 삭제

```
git push origin --delete 브랜치이름
```

📌 예시

```
git push origin --delete feature/login-ui
```

- 원격 저장소의 `feature/login-ui` 브랜치를 삭제함
- 로컬 브랜치는 **그대로 존재**

⚠️ **주의:** 삭제는 즉시 반영되며 복구가 어렵다 (`git reflog`도 불가)

✅ 4. 원격 브랜치 삭제: 옛 방식 (권장 안함)

```
git push origin :브랜치이름
```

예시:

```
git push origin :feature/login-ui
```

- `:`는 "해당 브랜치로 아무것도 푸시하지 않겠다" → 삭제 의미
- 옛날 방식이고, `--delete`가 더 명확해서 권장됨

✅ 5. 실전 예시 요약

작업	명령어
로컬 브랜치 푸시	`git push origin my-feature`
푸시 + 추적 설정	`git push -u origin my-feature`
원격 브랜치 삭제	`git push origin --delete my-feature`
옛 방식 삭제	`git push origin :my-feature`

✅ 6. 원격 브랜치 목록 확인

```
git branch -r
```

```
origin/main
origin/dev
origin/feature/login-ui
```

✅ 7. 삭제된 원격 브랜치 로컬에서 정리

```
git fetch -p
```

> `-p` = `--prune`: 원격에서 삭제된 브랜치를 로컬에서 정리
> `origin/삭제된브랜치` 가 잔존하는 걸 제거함

✅ 8. GUI 툴에서 푸시/삭제 (선택사항)

- SourceTree, GitKraken 등에서는 브랜치 우클릭 → “Push” 또는 “Delete remote branch” 선택 가능
- 내부적으로 위 명령어와 동일하게 동작함

✅ 마무리 요약표

목적	명령어
로컬 브랜치 원격 푸시	`git push origin <branch>`
추적 설정 포함 푸시	`git push -u origin <branch>`
원격 브랜치 삭제	`git push origin --delete <branch>`
로컬에서 삭제 반영	`git fetch -p`
원격 브랜치 목록 확인	`git branch -r`

origin 외 다른 remote 설정 (upstream 등)

✅ 1. 기본 개념

용어	의미
`origin`	보통 내가 클론하거나 푸시할 기본 원격 저장소 이름
`upstream`	**원본 프로젝트 저장소** (fork해서 작업할 때 주로 사용)
`backup`, `devserver` 등	필요에 따라 추가 가능한 다른 원격 저장소 이름

Git은 **여러 개의 원격 저장소를 동시에 연결**할 수 있고, 각각 이름으로 구분해 사용함.

✅ 2. `git remote add` 로 추가

```
git remote add 이름 URL
```

예: upstream 추가

```
git remote add upstream https://github.com/original-author/project.git
```

이제 `origin` 외에 `upstream`이라는 remote 이름으로도 fetch/pull/push 가능해짐.

✅ 3. 연결된 remote 목록 보기

```
git remote -v
```

```
origin   https://github.com/myname/project.git (fetch)
origin   https://github.com/myname/project.git (push)
upstream https://github.com/original-author/project.git (fetch)
upstream https://github.com/original-author/project.git (push)
```

✅ 4. 원격 저장소에서 가져오기 (`fetch`, `pull`)

✅ `fetch` (병합 없이 가져오기)

```
git fetch upstream
```

→ `upstream/main` 등의 원격 브랜치가 업데이트됨.

✅ `pull` (병합까지 수행)

```
git pull upstream main
```

→ `upstream/main`의 내용을 로컬 `main`에 병합함.

✅ 5. push는 origin으로, pull은 upstream에서

```
git push origin 내브랜치
git fetch upstream
git merge upstream/main
```

> 보통 **내 저장소(origin)**로 푸시하고,
> **원본 저장소(upstream)**로부터 최신 변경사항을 받아와 병합하는 구조로 작업함.

✅ 6. remote 이름 변경/삭제

✅ 이름 변경

```
git remote rename 기존이름 새이름
```

예:

```
git remote rename upstream mainserver
```

✅ 삭제

```
git remote remove upstream
```

또는:

```
git remote rm upstream
```

✅ 7. 실전: 오픈소스 fork 구조

1. GitHub에서 원본 저장소를 **Fork**
2. 내 저장소를 `git clone` → 자동으로 `origin` 설정됨
3. 원본 저장소를 `upstream`으로 추가

```
git remote add upstream https://github.com/original/project.git
```

1. `upstream`에서 주기적으로 최신 소스 받아오기

```
git fetch upstream
git rebase upstream/main
```

1. 변경 후 내 저장소(origin)에 푸시하고 PR 생성

✅ 8. 참고 명령어 요약

작업	명령어
원격 저장소 추가	`git remote add upstream <url>`
목록 확인	`git remote -v`
원격 브랜치 fetch	`git fetch upstream`
병합 pull	`git pull upstream main`
push to origin	`git push origin <branch>`
이름 변경	`git remote rename upstream backup`
삭제	`git remote remove upstream`