10. 서브모듈과 서브트리

서브모듈 개념 및 설정

✅ 1. 서브모듈이란?

> **다른 Git 저장소를 현재 Git 프로젝트 안에 포함시키는 기능**
> 하나의 프로젝트(상위) 안에서 **다른 프로젝트(하위)를 Git으로 직접 관리**하고 싶은 경우에 사용

📦 예시

- `my-app/` ← 메인 프로젝트
  - `libs/some-library/` ← 외부 오픈소스 프로젝트를 가져와 직접 버전 고정
  - `firmware/` ← 다른 Git 저장소로 관리되는 펌웨어 코드

✅ 2. 언제 서브모듈을 사용하는가?

상황	설명
독립적인 저장소를 함께 관리	외부 라이브러리를 소스 채로 직접 가져와야 할 때
버전 고정을 원할 때	특정 커밋에 고정시켜 테스트 가능
프로젝트 내 여러 저장소를 구조화	Firmware + Backend + Frontend가 각각 Git인 경우

✅ 3. 기본 구조

```
my-repo/
├── .gitmodules
├── subproject/       ← 실제 폴더
└── (commit reference만 저장됨)
```

- `.gitmodules` : 서브모듈 메타데이터 파일
- `subproject/` : 실제 체크아웃된 하위 저장소
- **상위 저장소에는 커밋 해시만 저장되고, 실제 코드가 포함되진 않음**

✅ 4. 서브모듈 추가 방법

```
git submodule add <서브모듈 Git URL> <폴더경로>
```

예:

```
git submodule add https://github.com/some/library.git libs/library
```

결과:

- `.gitmodules` 생성
- `libs/library/` 폴더에 해당 저장소 clone
- 현재 커밋 기준으로 버전 고정

✅ 5. 서브모듈 커밋과 push

```
git add .gitmodules libs/library
git commit -m "Add submodule for library"
git push
```

서브모듈도 별도 저장소이므로,
해당 서브모듈 폴더 내에서도 독립적으로 커밋 & 푸시 해야 함:

```
cd libs/library
git checkout main
# 작업 후
git commit -am "Fix something"
git push
```

✅ 6. 클론 후 서브모듈 초기화

```
git clone <URL>
cd <project>
git submodule init
git submodule update
```

> 또는 단축 명령어:

```
git clone --recurse-submodules <URL>
```

✅ 7. 서브모듈 업데이트

서브모듈 내부에서 최신 버전 가져오기:

```
cd libs/library
git pull origin main
```

→ 상위 저장소에서 다시 커밋해줘야 적용됨:

```
cd ../
git add libs/library
git commit -m "Update submodule"
```

✅ 8. 서브모듈 관련 명령어 요약

명령어	설명
`git submodule add`	서브모듈 추가
`git submodule init`	초기화 (처음 클론 시)
`git submodule update`	서브모듈 다운로드
`git submodule update --remote`	원격의 최신 커밋으로 갱신
`git submodule foreach`	각 서브모듈에서 명령 반복 실행

✅ 9. 서브모듈 제거

```
git submodule deinit -f libs/library
rm -rf .git/modules/libs/library
git rm -f libs/library
```

→ `.gitmodules` 파일도 수동으로 편집해서 해당 항목 제거

✅ 10. 주의사항

이슈	설명
변경사항 반영 누락	상위 repo에 서브모듈 변경 커밋 필요
GitHub Actions 등에서 자동 처리 필요	CI에서 `--recurse-submodules` 명시
`.gitmodules`와 실제 상태 불일치	수동 정리 필요

🧠 요약

개념	내용
서브모듈	다른 Git 저장소를 현재 프로젝트에 포함시키는 방식
메타정보	`.gitmodules`, 고정된 커밋 해시
사용 목적	외부 의존성 포함, 다중 저장소 관리
주요 명령	`add`, `init`, `update`, `foreach`, `deinit`
push 주의	서브모듈도 독립적으로 `commit + push` 필요

- `git submodule add`, `init`, `update`

🔹 1. `git submodule add`

✅ 역할

현재 Git 프로젝트에 **새로운 서브모듈을 추가**할 때 사용하는 명령어.
즉, 외부 Git 저장소를 하위 디렉토리로 연결함.

✅ 기본 문법

```
git submodule add <git-url> [<path>]
```

- `git-url` : 서브모듈이 될 저장소의 Git 주소
- `path` : (선택) 현재 프로젝트 내에 clone할 디렉토리 경로

✅ 예시

```
git submodule add https://github.com/user/somelib.git libs/somelib
```

- `libs/somelib` 디렉토리에 외부 저장소가 clone됨
- `.gitmodules` 파일 생성됨
- 현재 커밋에 해당 서브모듈의 특정 커밋 해시가 기록됨

🔹 2. `git submodule init`

✅ 역할

`.gitmodules` 파일에 정의된 서브모듈을 **Git이 추적 대상으로 초기화**함.
즉, 서브모듈이 있는 프로젝트를 처음 clone한 이후, 로컬에서 사용 준비하는 단계.

✅ 사용 시점

- `git clone` 한 후 서브모듈이 빈 폴더일 경우
- `.gitmodules` 파일은 있지만 실제 디렉토리에 clone이 안 된 상태

✅ 예시

```
git submodule init
```

→ `.git/config` 에 서브모듈 정보가 등록됨
→ **이 명령어만으로는 실제 코드가 내려오진 않음!**

🔹 3. `git submodule update`

✅ 역할

서브모듈이 가리키는 커밋을 기준으로 **실제 코드 checkout**까지 수행.

✅ 사용 시점

- `init` 이후 실제 소스를 내려받을 때
- 서브모듈의 commit이 상위 repo에서 업데이트되었을 때

✅ 예시

```
git submodule update
```

- `init`된 서브모듈 디렉토리에 실제 파일을 내려받음
- 상위 repo에서 고정된 커밋으로 checkout됨

🧪 추가: 한 줄로 전체 처리

```
git clone --recurse-submodules <repo-url>
```

> 처음부터 `clone + init + update` 전부 한 번에 실행됨

🧠 요약 비교

명령어	설명	시점
`git submodule add`	새 서브모듈 추가 (최초 1회)	내 프로젝트에서 외부 저장소 포함할 때
`git submodule init`	`.gitmodules`에 등록된 서브모듈을 Git에 인식시킴	`clone` 이후 첫 사용 시
`git submodule update`	실제 파일 다운로드 및 커밋 checkout	`init` 후 or 커밋 이동 후

✅ 실전 예제 요약

```
# 서브모듈 추가
git submodule add https://github.com/user/lib.git external/lib

# 클론 받은 뒤 서브모듈 초기화 + 업데이트
git submodule init
git submodule update

# 또는 한 줄로:
git clone --recurse-submodules https://github.com/me/project.git
```

- # 의존성 관리

✅ 1. 의존성이란?

> 소프트웨어가 제대로 작동하기 위해 **외부에 의존하는 코드, 라이브러리, 도구**
> 대부분은 **오픈소스 패키지, 내부 공용 모듈, API** 등 형태로 존재

✅ 2. Git에서의 의존성 관리 방법 종류

방식	설명	사용 예
🔗 **Git Submodule**	외부 Git 저장소를 직접 하위에 포함	자체 버전 고정 필요 시
🎄 **Git Subtree**	외부 저장소를 통합시켜 병합	history를 함께 가져옴
📦 **패키지 매니저**	언어별 의존성 명세 관리	`npm`, `pip`, `gradle`, `go mod`
📝 **Lock 파일 사용**	버전 고정을 위한 snapshot 저장	`package-lock.json`, `Pipfile.lock`
🛠️ **CI/CD에서 자동 업데이트**	GitHub Actions 등으로 보안 패치	Dependabot
📁 **벤더링(Vendoring)**	직접 소스코드를 프로젝트에 포함	Go, C 계열에서 자주 사용

✅ 3. 언어별 의존성 관리 시스템

언어	도구	버전 고정 방식
JavaScript	`npm`, `yarn`	`package.json` + `package-lock.json`
Python	`pip`, `poetry`	`requirements.txt`, `Pipfile`
Java	`Maven`, `Gradle`	`pom.xml`, `build.gradle`
Go	`go mod`	`go.mod`, `go.sum`

언어	도구	버전 고정 방식
Rust	`cargo`	`Cargo.toml`, `Cargo.lock`
C/C++	`vcpkg`, `conan`, 직접 clone	`CMakeLists.txt`, `vendor/`
Ruby	`bundler`	`Gemfile`, `Gemfile.lock`

✅ 4. Git 기반 의존성 직접 선언

많은 패키지 매니저는 **Git URL을 직접 명시**해 외부 의존성을 설정할 수 있음.

📦 예시 (JavaScript)

```
"dependencies": {
  "my-lib": "git+https://github.com/user/my-lib.git#v1.2.0"
}
```

예시 (Python)

```
git+https://github.com/user/my-lib.git@v1.2.0#egg=my-lib
```

✅ 5. 의존성 보안 관리: GitHub Dependabot

> GitHub 저장소에 존재하는 의존성 파일을 스캔해
> 보안 취약점을 감지하고 자동으로 PR을 생성

설정 예시: `.github/dependabot.yml`

```
version: 2
updates:
  - package-ecosystem: "npm"
    directory: "/"
    schedule:
      interval: "daily"
```

✅ 6. 의존성 충돌 대응 전략

전략	설명
Semantic Versioning	MAJOR.MINOR.PATCH 기준 사용 (`^`, `~` 등)
Lock 파일 고정	CI 빌드 환경의 일관성 유지
Vendor 방식 사용	외부 코드 직접 포함 (소스 기반)
모듈화 분리	핵심 코드와 외부 코드의 의존 구간 분리

✅ 7. GitHub Actions와의 통합

- **빌드 & 테스트 시점에 의존성 자동 설치**
- **의존성 보안 점검**, 릴리즈 자동화
- `npm ci`, `pip install -r`, `go mod tidy` 등 명령어 자동화

🧠 요약

항목	설명
의존성 정의	패키지 매니저 or Git Submodule/Subtree
버전 고정	Lock 파일 or Git Tag
보안/업데이트	Dependabot, CI 자동 점검
전략 선택 기준	사용 언어, 외부 변경 빈도, 보안 민감도

서브트리 방식과 비교

✅ 1. 개념 정의

항목	Submodule	Subtree
정의	외부 저장소를 **포인터(커밋 해시)**로 연결하는 방식	외부 저장소를 **완전 병합(copy)**하여 현재 저장소 안에 포함
추적 방식	특정 커밋을 참조함 (링크 느낌)	전체 히스토리 포함한 코드 통합 (복사 느낌)
`.gitmodules` 필요 여부	필요함	필요 없음

✅ 2. 구조 및 커밋 관리 방식

🔸 Submodule 구조

```
📁 main-repo/
├── .gitmodules
├── libs/
|    └── sub-lib/   ← 서브모듈 폴더 (별도 Git)
```

- 메인 repo의 커밋에는 sub-lib의 **커밋 해시만 추적됨**
- 내부 내용은 별도 repo에서 관리됨

🔹 Subtree 구조

```
📁 main-repo/
 ├── libs/
 |     └── sub-lib/   ← 통합된 전체 코드 + Git 기록
```

- 서브 프로젝트가 **실제 코드와 커밋 히스토리 포함**된 상태로 통합됨

✅ 3. 커맨드 비교

작업	Submodule	Subtree
추가	`git submodule add <url>`	`git subtree add --prefix=path <url> <branch>`
업데이트	서브모듈 내부에서 pull 후 상위 repo에 커밋	`git subtree pull --prefix=path <url> <branch>`
푸시	서브모듈에서 별도로 commit & push 필요	`git subtree push` 명령 사용 가능
제거	`deinit`, `rm`, `.gitmodules` 수정	`rm -rf path` 및 커밋만 필요

✅ 4. 협업/CI 측면

항목	Submodule	Subtree
⚠️ CI 세팅	복잡함 (`--recurse-submodules` 필요)	단순함 (코드가 프로젝트에 포함됨)
GitHub Actions	별도 checkout 필요	없음
협업 시 git clone	`--recurse-submodules` 필수	일반 `git clone`으로 충분
접근성	`.gitmodules` 설정 없으면 코드 없음	누구나 clone하면 코드 있음

✅ 5. 장점과 단점 정리

🔸 Submodule

장점	단점
외부 저장소와의 분리 유지	clone/update 번거로움
독립 버전 관리 가능	서브모듈 직접 진입해서 commit 해야 함
가볍고 참조 기반	`.gitmodules`, `.git/config` 등 설정 복잡
배포 시 코드 무게 작음	서브모듈 commit 잊고 push 안 하면 꼬임 발생

🔹 Subtree

장점	단점
CI/CD에서 매우 단순	커밋 이력 통합 → 리포 크기 커짐
외부 repo 없이도 프로젝트 작동	서브 프로젝트를 개별적으로 독립 운영하기 어려움
PR 리뷰 시 서브 코드도 볼 수 있음	외부 저장소에서 history 추출해오기 번거로움
복잡한 설정 불필요	upstream 저장소가 자주 변경되면 병합 충돌 가능성 증가

✅ 6. 언제 Submodule vs Subtree?

상황	권장 방식	이유
외부 라이브러리/오픈소스 유지	**Submodule**	독립성, 외부 변경 수용 용이
사내 공통 모듈 공유	**Subtree**	코드 통합, 협업/배포 단순
한 저장소에서 모든 소스 관리	**Subtree**	CI/CD 및 배포 자동화 편리
오픈소스 외부에 pull 요청 예정	**Submodule**	원래 repo 분리 유지 필요

✅ 7. 실전 명령어 예시

🔹 Submodule

```
git submodule add https://github.com/example/lib.git libs/lib
git submodule init
git submodule update
```

🔹 Subtree

```
git subtree add --prefix=libs/lib https://github.com/example/lib.git main --squash
# 업데이트
git subtree pull --prefix=libs/lib https://github.com/example/lib.git main --squash
```

🧠 요약

항목	Submodule	Subtree
구조	포인터로 참조	전체 병합
push 대상	상위 + 하위 각각	하나로 통합
협업	복잡 (init/update 필요)	간단

항목	Submodule	Subtree
외부 저장소 추적	편함	어렵고 충돌 가능
버전 고정	쉬움 (해시 기반)	직접 관리 필요
의존성 분리	명확	상대적 불리

🔚 결론

- **깃 기반 라이브러리 재사용**이 목적이라면: `Submodule`
- **사내 모노레포 / 협업 배포 편의성**이 목적이라면: `Subtree`
- `git subtree add`, `pull`, `push`

✅ 1. `git subtree add`

📌 목적

외부 Git 저장소를 **현재 저장소의 하위 디렉토리로 통합**
(초기 1회 병합)

📜 문법

```
git subtree add --prefix=<디렉토리> <원격주소> <브랜치> [--squash]
```

- `--prefix`: 소스가 병합될 디렉토리 경로
- `--squash`: 커밋 히스토리를 하나로 압축해서 가져오기

💡 예시

```
git subtree add --prefix=libs/libA https://github.com/user/libA.git main --squash
```

- `libs/libA/` 경로에 외부 저장소 `libA`의 `main` 브랜치 코드가 들어옴
- `libA`의 전체 히스토리는 기본적으로 가져오며, `--squash` 옵션 시 하나의 커밋으로 압축됨

✅ 2. `git subtree pull`

📌 목적

이미 추가된 subtree 디렉토리의 **최신 커밋을 동기화**함

📜 문법

```
git subtree pull --prefix=<디렉토리> <원격주소> <브랜치> [--squash]
```

- 디렉토리를 기준으로 외부 repo의 변경분을 병합

💡 예시

```
git subtree pull --prefix=libs/libA https://github.com/user/libA.git main --squash
```

- `libA`의 최신 커밋을 가져와 `libs/libA/`에 병합함

✅ 3. `git subtree push`

📌 목적

하위 디렉토리에 있는 subtree를 **외부 저장소로 푸시**

> 사내 공용 라이브러리를 외부로 추출할 때 유용

📜 문법

```
git subtree push --prefix=<디렉토리> <원격주소> <브랜치>
```

💡 예시

```
git subtree push --prefix=libs/libA git@github.com:user/libA.git main
```

- 현재 저장소의 `libs/libA/` 하위 디렉토리 내용을
  외부 저장소의 `main` 브랜치에 푸시

🧠 주의사항 요약

항목	설명
`--prefix`	**반드시 디렉토리명 정확히 명시**
`--squash`	subtree 저장소의 커밋 히스토리를 하나로 병합
외부 저장소	`remote` 등록하지 않고도 URL 직접 사용 가능
push	remote에 **권한 필요** (SSH/HTTPS 등 인증됨)

🧪 실전 흐름 요약

```
# 1. 최초 추가
git subtree add --prefix=libs/libA https://github.com/user/libA.git main --squash

# 2. 이후 업데이트
git subtree pull --prefix=libs/libA https://github.com/user/libA.git main --squash

# 3. 수정된 코드를 외부 저장소에 푸시
git subtree push --prefix=libs/libA git@github.com:user/libA.git main
```

✅ Git remote 등록 방식으로도 사용 가능

```
git remote add libA https://github.com/user/libA.git
git subtree pull --prefix=libs/libA libA main --squash
```

🔚 정리

명령어	기능	사용 시기
`git subtree add`	외부 코드 최초 병합	초기 1회
`git subtree pull`	외부 저장소 최신 커밋 병합	업데이트
`git subtree push`	현재 디렉토리의 내용 외부 저장소로 푸시	라이브러리 분리 시