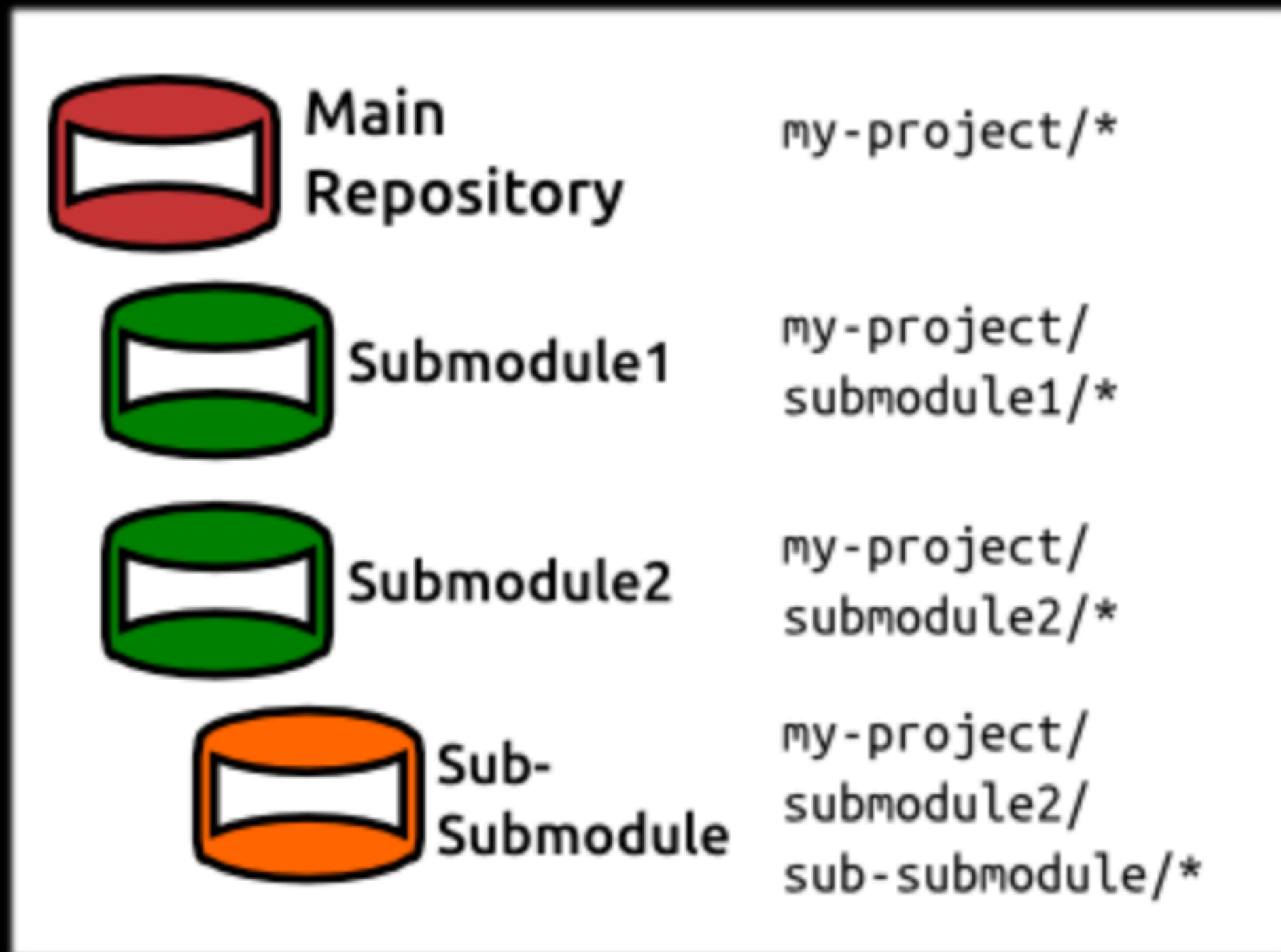


Git Submodule

2024.01.23.화 FE Tech Talk 박소영

1. Git Submodule이란?



<https://git-scm.com/book/en/v2/Git-Tools-Submodules>

- 중복 코드를 관리하고, 여러 프로젝트 간에 코드를 공유하면서도 각 프로젝트를 독립적으로 유지할 수 있는 방법 중 하나
- 다른 git 레파지토리의 특정 버전을 현재 프로젝트에 포함시킬 수 있고, 이를 통해 코드의 중복을 피하면서 각 프로젝트가 필요한 외부 코드 활용 가능함
- Git Docs
“서브 모듈을 사용하면 git 레파지토리를 다른 git 레파지토리의 하위 디렉터로 유지할 수 있습니다. 이를 통해 다른 레파지토리를 프로젝트에 복제하고 커밋을 분리하여 유지할 수 있습니다.”

1. Git Submodule이란?

git / Documentation / RelNotes / 1.5.3.txt

 npitre and gitster Documentation: move RelNotes into a directory of their own

761e742 · 14년 전 History

Code Blame 366 lines (262 loc) · 13.7 KB

Raw Copy Download Edit View

```
1  GIT v1.5.3 Release Notes
2  =====
3
4  Updates since v1.5.2
5  -----
6
7  * The commit walkers other than http are officially deprecated,
8    but still supported for now.
9
10 * The submodule support has Porcelain layer.
11
12 Note that the current submodule support is minimal and this is
13 deliberately so. A design decision we made is that operations
14 at the supermodule level do not recurse into submodules by
15 default. The expectation is that later we would add a
16 mechanism to tell git which submodules the user is interested
17 in, and this information might be used to determine the
18 recursive behaviour of certain commands (e.g. "git checkout"
19 and "git diff"), but currently we haven't agreed on what that
20 mechanism should look like. Therefore, if you use submodules,
21 you would probably need "git submodule update" on the
22 submodules you care about after running a "git checkout" at
23 the supermodule level.
24
```

- git submodule 도입 전 초기에는 서드파티 도구나 스크립트를 사용해 비슷한 기능 구현함
- git 1.5.3 버전에서 서브 모듈이라는 개념 처음 소개됨 (2007년 6월 29일 릴리즈됨)
- 이후 지속적으로 기능 추가되며 개발됨

2. Git Submodule을 도입하게 된 계기

- 비슷한 어드민형 웹사이트를 여러 프로젝트로 반복 구축하게 됨
- 초기엔 작은 프로젝트 사이즈로 시작해서 굳이 모듈별로 나눌 필요성을 느끼지 못했는데, 프로젝트가 점점 커지고, 여러 프로젝트에서 같은 모듈들로 사용하다보니 컴포넌트 및 프로젝트에 종속되지 않은 코드들을 분리해 관리해야겠다 필요성 느낌
- 공통으로 사용 가능한 코드는 모듈로 만들자
- 중복 코드를 관리하고,
여러 프로젝트 간 코드를 공유하면서,
각 프로젝트를 독립적으로 유지할 수 있는 방법들에는 무엇이 있을까

2. Git Submodule을 도입하게 된 계기

보일러플레이트와 컴포넌트/라이브러리는 운영 방법이 좀 차이가 있어요 :)
관련 부분을 경력 중에 5년이상은 고민했던 부분이라 정리해봤습니다 :)

[1. 보일러플레이트]

[1-1. 보일러플레이트 운영]

보일러플레이트의 운영 방법은

웹 사이트를 만들기 위해 필요한 **일반적인 요소들**만 모아서 **저장소에 운영**합니다.

특정 웹 사이트를 위한 기능은 존재하면 안되고,

회사에서 **공통된 사이트의 기능이 있다면 보일러플레이트**에 넣기도 합니다.

[1-2. 보일러플레이트 사용과 한계]

보일러플레이트를 사용할 때는 **보일러플레이트 저장소**를 **fork** 해서 웹 사이트를 만듭니다.

보일러플레이트가 변경이 될 때, 사용측도 변경이 필요한 경우가 있습니다.

하지만 폴더 구조와 사용측에서 변경된 코드에 의존성이 생겨서 명령어로 자동화하기 힘든 한계가 있어요 πππ

[2. 컴포넌트/라이브러리]

컴포넌트/라이브러리도 보일러플레이트와 마찬가지로 해당 요소만 저장소에 운영합니다.

하지만 회사 내부 사정에 따라 다르게 사용하게 되서, 적합한 방법을 사용하시는 것을 권장합니다 :)

1) NPM

- 방법: 외부 NPM에 올려서 관리.

- 장점: 가장 익숙한 방법

- 단점: 외부에 올리면 회사 코드가 외부에 노출되는 문제가 있어요.

2) private npm

- 방법: 사내 NPM 저장소를 구축하고, 사내 NPM에 올려서 관리.

- 장점: 가장 익숙한 방법. 코드가 외부에 노출되지 않음.

- 단점: 사내 NPM 저장소를 구축하는 노력이 들어요.

- 참고: <https://verdaccio.org/> 로 구현해서 사용하곤 합니다.

3) git submodule

- 방법: git 기능 중 submodule를 활용. 다른 저장소를 submodule로 연결할 수 있음.

- 장점: git 기능을 활용해서 다른 패키지 매니저에 영향이 없어요.

- 단점: git submodule의 형상관리가 초반에는 익숙하지 않을 수 있어요.

4) mono repo

- 방법: 하나의 저장소에서 여러개 패키지를 구성하는 방법

- 장점: 하나의 저장소에서 여러개 사이트를 만들고, 중복된 코드를 관리할 수 있음.

- 단점: mono repo에 대한 러닝 커브.

- 참고: <https://lerna.js.org/> 를 사용률이 높습니다.

[3. 의견]

- 보일러플레이트는 문서화 기반으로 운영하는 게 가장 좋을 것 같아요.

- 기반은 create-react-app으로 하고, 필요할 라이브러리들은 명령어를 문서화 해두는 방법입니다.

- 그럼 버전 업그레이드 할 때, 명령어의 도움을 받을 수 있을 것 같아요. - <https://create-react-app.dev/docs/updating-to-new-releases>

- 보일러플레이트 구축 문서화 명령어 예시)

\$ npx create-react-app my-app

\$ cd my-app

\$ npm install mobx mobx-react moment

- 코드를 작성하는 방법(Store / Template / Container 등)도 문서화해두고,

해당 문서 기반으로 파일/폴더를 생성하도록 하는 것이 좋아요.

추후에 동료가 생기면 동일한 룰로 프로젝트를 진행하기 유용합니다.

- 중복된 코드는 가장 쉬운 방법으로 git submodule로 관리할 수 있을 것 같아요 :)

3. Git Submodule의 사용

<서브모듈 추가>

\$ git submodule add [remote repository] [디렉토리명(option)]

작업할 메인 프로젝트에서 위 명령어를 사용해 git 저장소를 서브 모듈로 추가. 기본적으로 서브모듈은 프로젝트의 저장소명으로 디렉토리를 만들게 되는데, 다른 이름으로 만들고 싶으면 명령 마지막에 원하는 이름을 넣으면 됨.

.gitmodule 파일 :

서브 디렉토리와 하위 프로젝트 url의 매핑 정보를 담은 설정 파일.
위 파일을 보고 어떤 서브모듈 프로젝트가 있는지 확인 가능.
해당 파일도 버전을 관리해야하므로 다른 파일처럼 git에 올려야 함

부모와 서브모듈 사이의 유일한 연결 고리 =

부모 커밋에 저장된 서브모듈 체크아웃 SHA의 기록된 값.

이 기록된 SHA의 변경 사항은 서브모듈에 자동으로 반영되지 X

git diff --cached 명령어를 실행하면 신기한 점 발견할 수 있음
서브모듈로 추가한 디렉토리들을 서브모듈로 취급하기 때문에
해당 디렉토리 아래의 파일 수정사항을 직접 추적하지 않음
대신 서브모듈 디렉토리를 통째로 특별한 커밋으로 취급하게 됨
git diff --cached --submodule

```
diff --git a/test.js b/test.js
```

```
new file mode 100644
```

```
index 0000000..e69de29
```

```
(END)
```

```
diff --git a/dayjs b/dayjs
```

```
new file mode 160000
```

```
index 0000000..bdcc336
```

```
--- /dev/null
```

```
+++ b/dayjs
```

```
@@ -0,0 +1 @@
```

```
+Subproject commit bdcc336613c9fa466b385574e88d0b43629475bc
```

| | |
|-----------------------|------------------------------------|
| → dayjs @ bdcc336 | added dayjs module |
| → submodule @ b5b3b56 | add submodule submodule |
| → .gitignore | add submodule submodule |
| → .gitmodules | add submodule submodule |

3. Git Submodule의 사용

<서브모듈 추가>

하위 프로젝트를 포함한 커밋을 생성하면 그림과 같은 결과를 확인할 수 있는데,
dayjs와 react-hook-form의 디렉터리 모드는 160000으로 노출됨
(하위 디렉토리나 파일이 아닌 디렉토리 항목으로 커밋을 기록한다는 의미)

160000 모드 = 일반적인 파일이나 디렉토리가 아닌 git link를 의미
100644 = 일반 파일

이후 push를 하면 git submodule 추가가 완료됨

```
[master (최상위 -커밋) 59b4a0a] added dayjs, react-hook-form module
4 files changed, 9 insertions(+)

create mode 100644 .gitmodules

create mode 160000 dayjs

create mode 160000 react-hook-form

create mode 100644 test.js
```

3. Git Submodule의 사용

<서브모듈을 포함하는 프로젝트 Clone>

서브모듈을 포함하는 프로젝트를 clone하면 기본적으로 서브모듈 디렉토리가 비어있는 상태로 나오게 됨.
실제 서브모듈의 내용을 가져오기 위해선 로컬의 submodule 정보를 바탕으로 submodule의 remote repository 데이터를 끌어와야 함.

\$ git submodule init
서브모듈 정보를 기반으로 로컬 환경설정 파일 구성

\$ git submodule update
서브모듈 remote 저장소에서 데이터를 가져오고,
메인 프로젝트가 현재 저장하고 있는 submodule의 commit 정보를 바탕으로 checkout함

\$ git clone [url] --recurse-submodule
위의 init + update를 자동으로 실행해 서브모듈을 초기화하고 업데이트 함

\$ git submodule foreach git checkout main
각각의 서브모듈에 대한 명령어 일괄실행

3. Git Submodule의 사용

<메인 프로젝트에서 서브모듈 업데이트>

가장 단순한 서브 모듈 사용 방법은 서브 모듈을 수정하지 않고, 참조만 하면서 서브 모듈을 최신 버전으로 업데이트 하는 방식

서브 모듈 프로젝트를 최신으로 업데이트 하려면
서브 모듈 디렉토리로 이동 후 git fetch + git merge를 해줘야 함

\$ git submodule update --remote [서브모듈명] : git fetch + git merge
\$ git submodule update --remote : 모든 서브모듈 업데이트

기본적으로 서브모듈 저장소의 master 브랜치를 checkout하고 업데이트를 수행하는데 브랜치 변경도 가능함
.gitsubmodule 파일에 설정 또는 개인 설정 파일인 .git/config 파일에 설정 가능
\$ git config -f .gitmodules submodule.submodule.branch stable

\$ git submodule status : 항상 현재 상태. 현재 체크아웃된 커밋
\$ git ls-tree HEAD : 대상 상태

3. Git Submodule의 사용

<메인 프로젝트에서 서브모듈 수정 & 업데이트>

서브모듈을 수정하고, 그 내용을 담은 커밋을 유지한 채로 메인 프로젝트와 서브모듈을 함께 관리하는 방법.

서브모듈 저장소에서 git submodule update 명령어를 실행하면 git은 서브모듈의 변경사항을 업데이트 함

하지만 서브모듈 로컬 저장소는 'Detached Head' 상태로 남음

(=변경 내용을 추적하는 로컬 브랜치가 없다는 의미)

변경 내용을 추적하는 브랜치 없이 서브모듈에서 수정 작업을 하게 되면 이후 git submodule update 시 수정한 내용 잃어버릴 수 있음

따라 서브모듈 안에서 수정사항을 추적하려면 추가 작업이 필요함

3. Git Submodule의 사용

<메인 프로젝트에서 서브모듈 수정 & 업데이트>

1. 서브모듈 코드 수정

- 메인 프로젝트의 디렉토리에서 하위 서브모듈 디렉토리로 이동 : `$ cd <서브모듈-디렉토리>`
- 추적할 브랜치를 checkout 해야 함 : `$ git checkout stable`
- 여기서 서브모듈의 코드를 수정하고 변경 commit

2. 메인 프로젝트에 변경사항 반영


- 다시 메인 프로젝트로 이동 : `$ cd ../`
- 서브모듈의 변경사항 반영 : `$ git submodule update --remote --merge`
(서브모듈에서 변경된 내용을 가져와 메인 프로젝트에 반영함)

3. 메인 프로젝트에 변경사항 commit

- 메인 프로젝트에서는 서브모듈의 변경사항을 반영한 후, 메인 프로젝트 자체의 변경사항과 함께 커밋함


3. Git Submodule의 사용

bash

 Copy code

```
$ git diff
Submodule DbConnector c87d55d..82d2ad3:
  > Merge from origin/stable
  > updated setup script
  > unicode support
  > remove unnecessary method
  > add new option for conn pooling
```

bash

 Copy code

```
$ git push --recurse-submodules=check
The following submodule paths contain changes that can
not be found on any remote:
  DbConnector
```

Please try

```
git push --recurse-submodules=on-demand
```

or **cd** to the path and use

```
git push
```

to push them to a remote.

1. 서브모듈 수정사항 공유하기


현재 상황 = 서브모듈이 변경된 내용을 포함하고 있음
(일부는 Upstream 저장소에서 가져오고, 일부는 로컬에서 직접 수정함)

2. 서브모듈 push 상태 확인

메인 프로젝트에서 메인 프로젝트의 변경사항을 Push하기 전,
서브모듈을 모두 Push 했는지 검사하는 것 중요

3. Git Submodule의 사용

bash

 Copy code

```
$ git push --recurse-submodules=on-demand  
Pushing submodule 'DbConnector'  
Counting objects: 9, done.  
...
```

3. 서브모듈 Push 또는 on-demand 설정

서브모듈의 변경사항을 Push하지 않은 상태로
메인 프로젝트를 Push하면 문제가 발생할 수 있음

4. 영구적인 설정

서브모듈의 변경사항이 항상 확인되도록 하려면,
git config push.recurseSubmodules on-demand 명령을
사용해 설정.

3. Git Submodule의 사용

<서브모듈 Merge>

여러 명의 개발자가 동시 작업하는 경우 발생할 수 있는 서브모듈 간의 충돌 해결

1. 서브모듈 상태 확인

다른 개발자가 서브모듈을 수정하고 Push 한 경우,

메인 프로젝트에서 서브모듈을 Pull 하는 과정에서 충돌이 발생할 수 있음

Git은 이 때 "merge following commits not found"라는 메시지를 통해 서브모듈의 브랜치가 갈라져서 Merge가 필요하다는 것을 알려줌

2. 서브모듈의 두 커밋 비교

git diff 명령을 사용하여 서브모듈의 두 브랜치 간의 변경사항을 확인함

두 브랜치의 SHA 해시값을 확인하고, 어떤 부분이 충돌이 나는지를 파악함

3. 서브모듈의 브랜치 생성 및 Merge

충돌이 발생한 서브모듈 디렉토리로 이동하여 충돌이 난 부분을 해결하고, 변경사항을 커밋.

새로운 브랜치를 생성하고, 이 브랜치를 사용하여 메인 프로젝트에서 서브모듈을 Pull.

메인 프로젝트에서 서브모듈을 Merge 하여 충돌을 해결함

(git rev-parse : git에서 참조값 출력 명령어)

4. 서브모듈 업데이트 및 메인 프로젝트 업데이트

서브모듈 디렉토리에서 업데이트를 수행하고, 변경된 내용을 메인 프로젝트에 반영함

충돌이 발생한 서브모듈의 변경사항을 확인하고, 필요한 작업을 수행함

```
$ git pull
remote: Counting objects: 2, done.
remote: Compressing objects: 100% (1/1), done.
remote: Total 2 (delta 1), reused 2 (delta 1)
Unpacking objects: 100% (2/2), done.
From https://github.com/chaconinc/MainProject
  9a377d1..eb974f8  master    -> origin/master
Fetching submodule DbConnector
warning: Failed to merge submodule DbConnector (merge following commits not found)
Auto-merging DbConnector
CONFLICT (submodule): Merge conflict in DbConnector
Automatic merge failed; fix conflicts and then commit the result.
```

```
$ cd DbConnector

$ git rev-parse HEAD
eb41d764bccf88be77aced643c13a7fa86714135

$ git branch try-merge c771610
(DbConnector) $ git merge try-merge
Auto-merging src/main.c
CONFLICT (content): Merge conflict in src/main.c
Recorded preimage for 'src/main.c'
Automatic merge failed; fix conflicts and then commit the result.
```

```
$ git diff
diff --cc DbConnector
index eb41d76,c771610..0000000
--- a/DbConnector
+++ b/DbConnector
```


4. Git Submodule의 문제점

- 직관적이지 않은 git submodule :
git submodule을 사용하는 경우 로컬에서 프로젝트를 실행할 때
원격 디렉터리에서 submodule을 다운로드하고 종속성을 설치해야 함
`$ git submodule update -init && npm install`
- 사용중인 버전 추적 어려움 :
다른 개발자가 어떤 라이브러리 버전을 사용 중인지, 어떤 버전을 키고 있는지 확인할 수 없음.
`$ Subproject commit 33d6a12560f348c5c663c004ac13c7a124dd982c`
- 업데이트 & 동기화의 어려움 (수동 업데이트) :
서브모듈의 기록된 버전을 업데이트하고 상위 레파지토리에서 최신 변경사항을 가져와도,
서브모듈 레파지토리는 여전히 이전 버전의 서브모듈을 가리킴 (업데이트하려면 수동으로 명령 실행)
협업자들이 명시적으로 업데이트 명령을 실행해야 하므로, 업데이트를 놓치거나 동기화 문제 발생 가능
- 변경사항 기여 어려움 :
서브모듈로 연결된 저장소에 변경사항 기여 복잡함.
일반적으로 서브모듈은 커밋에 고정되어있어, 서브모듈의 변경사항을 만들고 이를 상위 저장소에 통합하는 과정 번거로움
- 서브 모듈이 모듈 간의 종속성 관계를 관리하지 않음
- 충돌을 해결할 때 하위 모듈에 대한 포인터 저장하지 않음 :
포인터를 수동으로 업데이트 하지 않으면 변경사항 병합 시 해결된 충돌 손실됨
- 다른 의존성 관리 도구의 존재 :
현재는 git submodule보다 훨씬 강력하고 사용하기 쉬운 의존성 관리 도구들 많아졌으므로 굳이 git submodule을 사용할 이유 없음
submodule 기능에 대한 추가 자동화를 제공하는 다양한 도구가 있지만 (git-subtree, gitslave, braid, giternal 등),
현재로서 프로젝트와 레포지토리 전반에서 모듈의 소스코드 변경사항과 종속성을 모두 관리하는 도구는 JS의 Bit 뿐.

5. Git Submodule의 사용 예시

타 프로젝트에서의 git submodule 활용 사례

- Git : <https://github.com/git/git/blob/master/.gitmodules>
- Rust : <https://github.com/rust-lang/rust/blob/867d39cdf625e4db4b381faff993346582e598b4/.gitmodules>
- Ace : <https://github.com/ajaxorg/ace/blob/64101ccf7e60d8f5bab26ca0108111a6f4690db2/.gitmodules#L4>
- OpenStack : <https://github.com/openstack/openstack/blob/master/.gitmodules>

6. Git submodule과 다른 선택지들 비교

- subtree : git의 기능 중 하나, 코드를 여러 저장소 간에 공유하고 관리하는데 사용되는 도구.
submodule과 비슷하지만, 외부 저장소의 코드를 현재 저장소의 특정 디렉토리에 그대로 가져와 관리 가능
- Sub module vs Subtree
 - 둘 다 git에서 서브 프로젝트나 외부 레파지토리를 메인 프로젝트에 통합하는 다른 접근 방식
 - Git submodule :
 - 독립성 : git submodule을 사용하면, 서브모듈은 완전히 독립된 git 레파지토리로서 유지됨
 - Commit : 메인 프로젝트에서 서브모듈의 특정 커밋을 가리키며, 서브모듈의 변경사항은 해당 서브모듈 레파지토리에서 관리됨
 - 분리된 작업 : 서브모듈에 대한 작업을 별도로 수행해야 하며, 메인 프로젝트와 서브모듈 간의 통합을 위해 추가적인 명령어 필요
 - Git Subtree :
 - 통합된 레파지토리 : git subtree 사용 시, 외부 레파지토리의 일부를 메인 레파지토리에 병합 가능
(마치 외부 레파지토리가 메인 레파지토리의 일부인 것처럼 동작함)
 - 커밋 히스토리 통합 : 변경 내용이 서브트리에 커밋되면, 메인 프로젝트의 히스토리에도 커밋이 기록됨
 - 메인 프로젝트와 함께 변경 : 서브트리의 코드는 메인 프로젝트 디렉토리 안에 있으므로 메인 프로젝트에서 직접 수정 가능함
- Shared Libraries : 코드의 일부를 라이브러리로 분리하고, 해당 라이브러리를 정적/동적으로 프로젝트에 통합함
라이브러리를 사용하기 위해 해당 라이브러리의 헤더 파일을 포함하고, 컴파일 시에 라이브러리를 링크해야 함
정적 링크(static linking) : 라이브러리 코드가 애플리케이션의 실행파일에 포함됨
동적 링크(dynamic linking) : 라이브러리는 실행 파일과 별도로 존재하며, 실행 시에 필요한 라이브러리를 로드함

6. Git submodule과 다른 선택지들 비교

| | Git Submodule | Git Subtree | Package Managers | Shared Libraries | Monorepo |
|-----------------|---------------------------|---------------------------------------|---------------------------------|------------------------------|----------------------------------|
| 코드 공유 방식 | 외부 레파지토리를 포함 | 외부 레포지토리 일부 병합 | 패키지 매니저 사용 | 라이브러리로 공유 | 하나의 대형 레포지토리에 모든 프로젝트 포함 |
| 유지보수 및 업데이트 용이성 | 중복 코드 수정 및 업데이트 어려움 | 중복 코드 수정 및 업데이트 어려움 | 패키지 버전 업데이트 용이, 패키지 자체 수정 어려움 | 라이브러리 내에서 자유롭게 수정 가능 | 일관된 업데이트 및 유지보수 용이 |
| 사용자 정의 및 수정 용이성 | 제한적 (서브 모듈의 변경이 복잡함) | 어려움 (서브트리 변경이 복잡함) | 패키지 버전의 수정 용이, 패키지 자체 수정 어려움 | 라이브러리 내에서 자유롭게 수정 가능 | 프로젝트 간에 코드를 자유롭게 조작 가능 |
| 프로젝트 간 독립성 | 유지됨 | 유지됨 | 유지됨 | 유지됨 | 유지됨 |
| 변경사항 통합 용이성 | 어려움 (서브 모듈의 변경이 복잡함) | Submodule에 비해 비교적 간단함 | 패키지 버전 업데이트는 쉽지만 패키지 자체 수정이 어려움 | 변경 및 업데이트 어려움 | 일관된 업데이트 및 충돌 해결이 어려울 수 있음 |
| 커스터마이징 용이성 | 어려움 (외부 레파지토리 변경이 제한적) | 어려움 (외부 레파지토리 변경이 제한적) | 패키지 버전 변경이 어려움. 패키지 내부 수정 어려움 | 라이브러리 내에서 자유롭게 수정 가능 | 프로젝트 내에서 코드 수정이 자유롭지만 충돌 가능성 존재함 |
| 코드 배포 용이성 | 복잡함 (외부 레파지토리의 배포와 연계) | 비교적 간단함 (외부 레파지토리의 특정 부분만 가져와 사용함) | 패키지 버전을 통한 배포가 쉬움 | 라이브러리를 배포하려면 빌드 및 배포 과정이 필요함 | 여러 프로젝트를 포함한 대형 레포지토리를 관리하기 어려움 |

7. 관련 토픽 주제

- Mono Repo vs Multi Repo vs Monolith Repo(django 같은)
- 라이브러리 또는 새로운 프레임워크를 선택할 때 우선순위 기준과 경험 또는 선택 과정
- 브랜치 전략, 배포 주기 등 CI/CD