



서울대학교 컴퓨터공학부 특강

Rust 크로스 플랫폼 프로그래밍

Chris Ohk

utilForever@gmail.com

발표자 소개

서울대학교 컴퓨터공학부 특강
Rust 크로스 플랫폼 프로그래밍

- 옥찬호 (Chris Ohk)
 - (현) Momenti Engine Engineer
 - (전) Nexon Korea Game Programmer
 - Microsoft Developer Technologies MVP
 - C++ Korea Founder & Administrator
 - Reinforcement Learning KR Administrator
 - IT 전문서 집필 및 번역 다수
 - 게임샐러드로 코드 한 줄 없이 게임 만들기 (2013)
 - 유니티 Shader와 Effect 제작 (2014)
 - 2D 게임 프로그래밍 (2014), 러스트 핵심 노트 (2017)
 - 모던 C++ 입문 (2017), C++ 최적화 (2019)



utilForever



들어가며

서울대학교 컴퓨터공학부 특강
Rust 크로스 플랫폼 프로그래밍

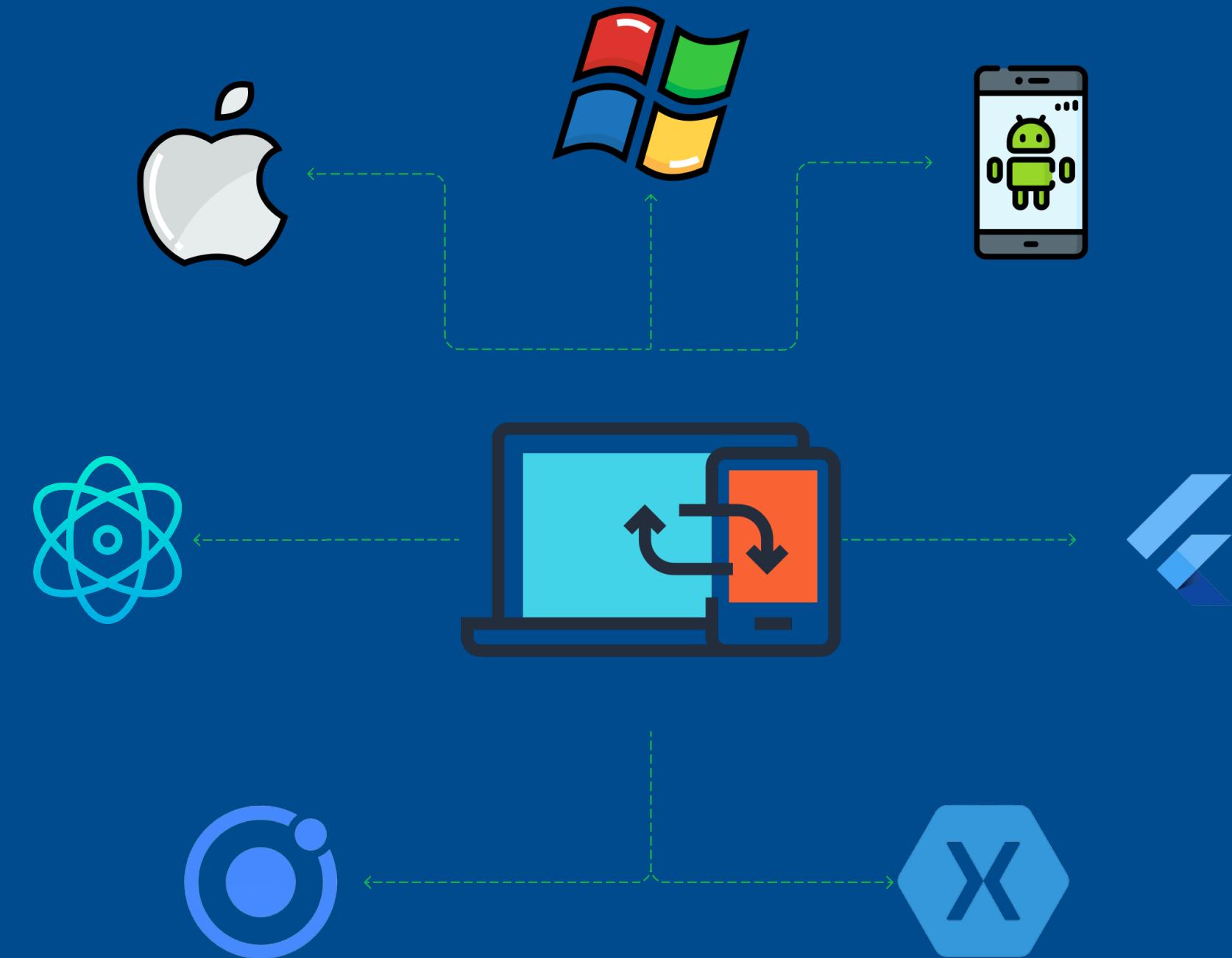
- Rust 언어의 문법을 설명하지 않습니다. 자세하게 공부하고 싶다면 공식 문서를 참고하세요.
<https://doc.rust-lang.org/book/>
- 오늘 발표에서 다뤘던 코드들은 다음 저장소에서 확인 가능합니다.
<https://github.com/utilForever/2023-SNU-Rust-CrossPlatform>

목차

서울대학교 컴퓨터공학부 특강
Rust 크로스 플랫폼 프로그래밍

- 크로스 플랫폼 프로그래밍을 하게 된 이유
- Rust 크로스 플랫폼 프로그래밍
 - Part 1 : Rust + Swift
 - Part 2 : Rust + WebAssembly
 - Part 3 : Refactoring
- 몇 가지 시행 착오와 팁

1



크로스 플랫폼 프로그래밍을 한 이유

와, 이직!

서울대학교 컴퓨터공학부 특강
Rust 크로스 플랫폼 프로그래밍



이직 후 첫번째 업무

서울대학교 컴퓨터공학부 특강
Rust 크로스 플랫폼 프로그래밍

- 여러 플랫폼에서 사용할 수 있는 코어 엔진을 만들어주세요.
 - iOS
 - Android
 - Backend
 - Web
 - ...

기존에는 어떻게 개발되고 있었는가

서울대학교 컴퓨터공학부 특강
Rust 크로스 플랫폼 프로그래밍

- 똑같은 기능을 하는 엔진 코드가 여러 플랫폼에 각각 구현되어 있었다.
 - iOS 앱 → C + Objective-C 기반
 - 프론트엔드 → TypeScript 기반
- 새로운 기능을 구현하거나 기존 기능을 수정해달라는 요청이 들어오면,
 - iOS 앱 → C 코드를 구현한 뒤, Objective-C 코드를 통해 앱에 적용한다.
 - 프론트엔드 → TypeScript 코드를 구현해 웹에 적용한다.

- 어떤 기능을 추가/삭제하거나 변경해야 할 때 지원하는 플랫폼마다 작업을 해줘야 한다.
 - 유지보수 측면에서 비효율적이다.
 - 지원하는 플랫폼이 늘어날수록 작업량이 늘어난다.
- 똑같은 기능을 수행했을 때 플랫폼마다 동작 결과가 달라질 수 있다.
 - 플랫폼마다 서로 다른 사람이 구현하기 때문에 발생하는 문제다.
 - 서로 다른 동작으로 인해 사용자에게 불편을 초래할 수 있다.

어떻게 개선할 것인가

서울대학교 컴퓨터공학부 특강
Rust 크로스 플랫폼 프로그래밍

- 플랫폼마다 로직을 구현하지 않고, 한 곳에서 작업하면 좋겠다.
- 다양한 플랫폼에 어떻게 대응해야 할까?
 - iOS 앱 → Swift API 제공
 - Android 앱 → Kotlin API 제공
 - 백엔드 → Elixir API 제공
 - 웹 사이트 → 웹어셈블리 바이너리 파일 제공

왜 Rust를 선택했는가

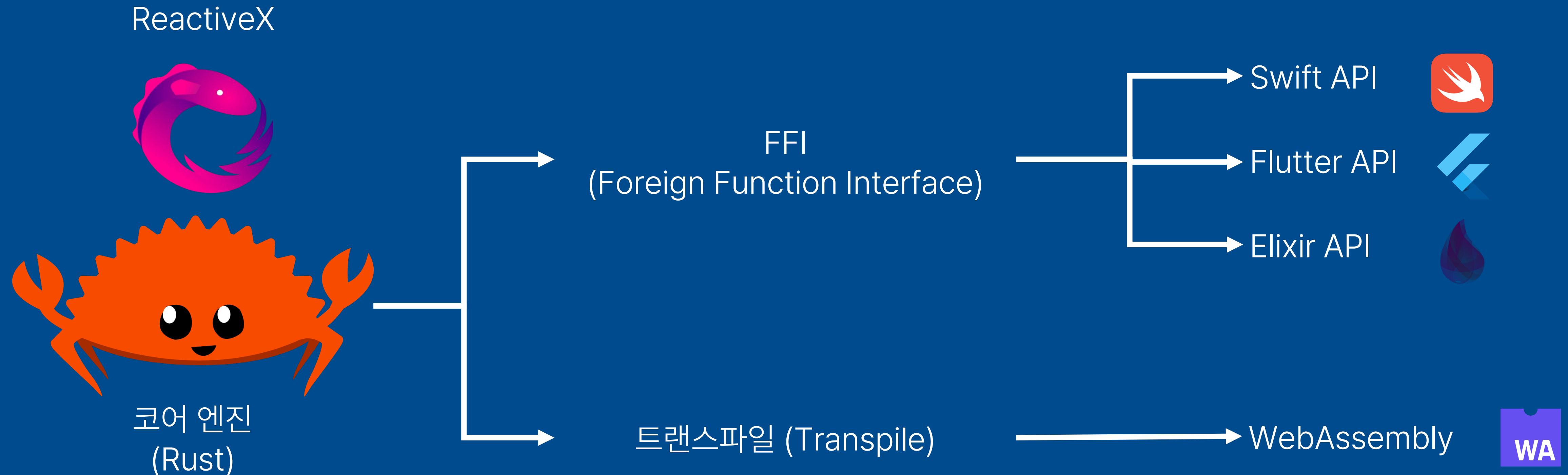
서울대학교 컴퓨터공학부 특강
Rust 크로스 플랫폼 프로그래밍

- 다양한 언어를 지원하기 위한 API와 웹 어셈블리 코드를 만들 수 있는 언어
 - C++, Rust, Java, Go, ...
- Rust를 선택한 이유
 - 타입 안전성
 - 메모리 안전성
 - 동시성 프로그래밍 안전성
 - FFI를 통해 API 지원을 쉽게 할 수 있음
 - WebAssembly 바이너리를 쉽게 만들 수 있음

프로젝트 구조

서울대학교 컴퓨터공학부 특강
Rust 크로스 플랫폼 프로그래밍

- 같은 동작을 하는 프로그램을 여러 플랫폼에서 사용할 수 있도록 만들고 있다.
 - iOS 앱 (Swift), 하이브리드 앱 (Flutter)
 - 프론트엔드 (TypeScript), 백엔드 (Elixir)
- 사용자 입력에 따라 로직을 처리하고 결과물을 보여주기 위한 코드를 개별적으로 작성한다.



2

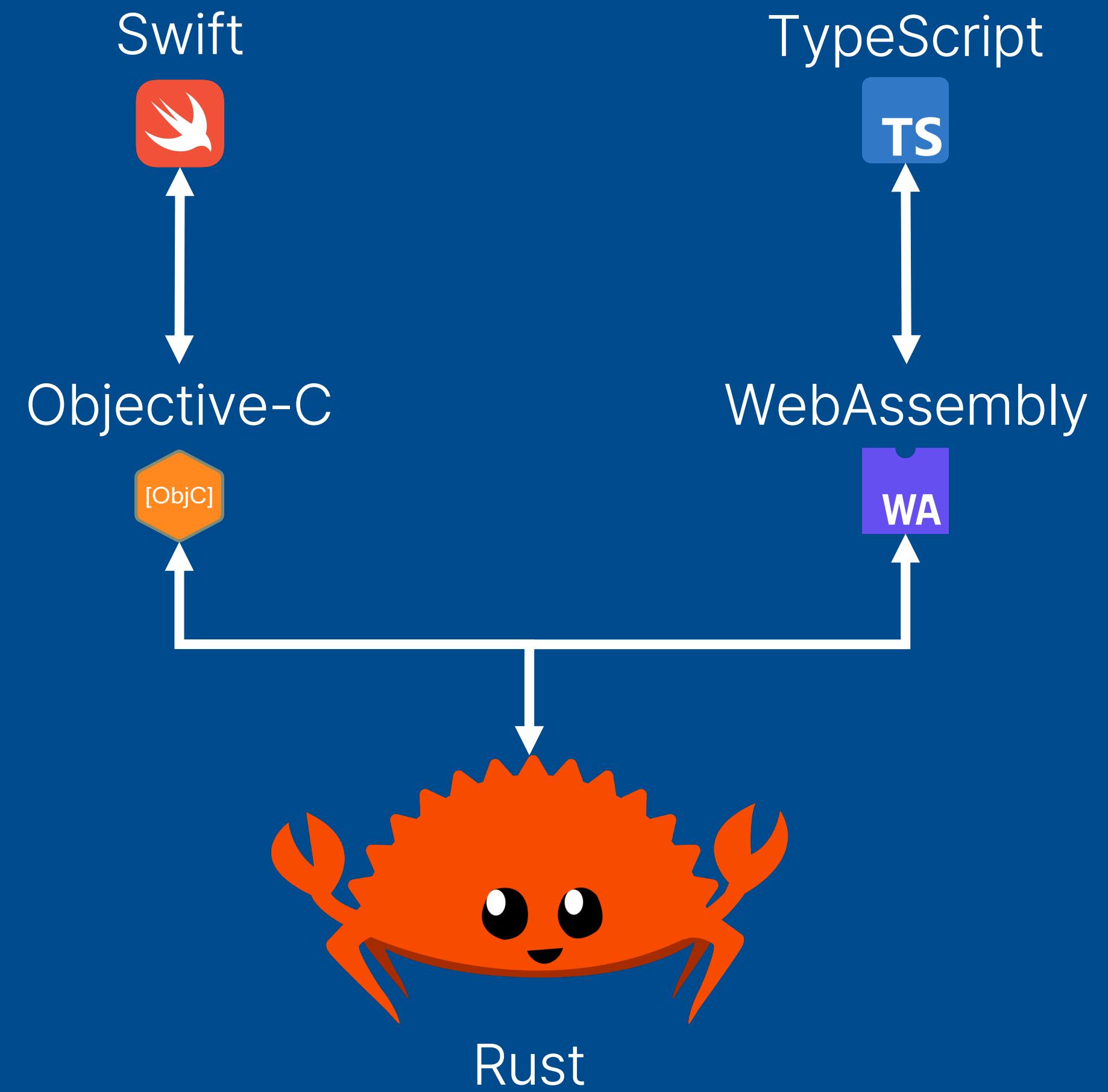


Rust 크로스 플랫폼 프로그래밍

무엇을 만들 것인가

서울대학교 컴퓨터공학부 특강
Rust 크로스 플랫폼 프로그래밍

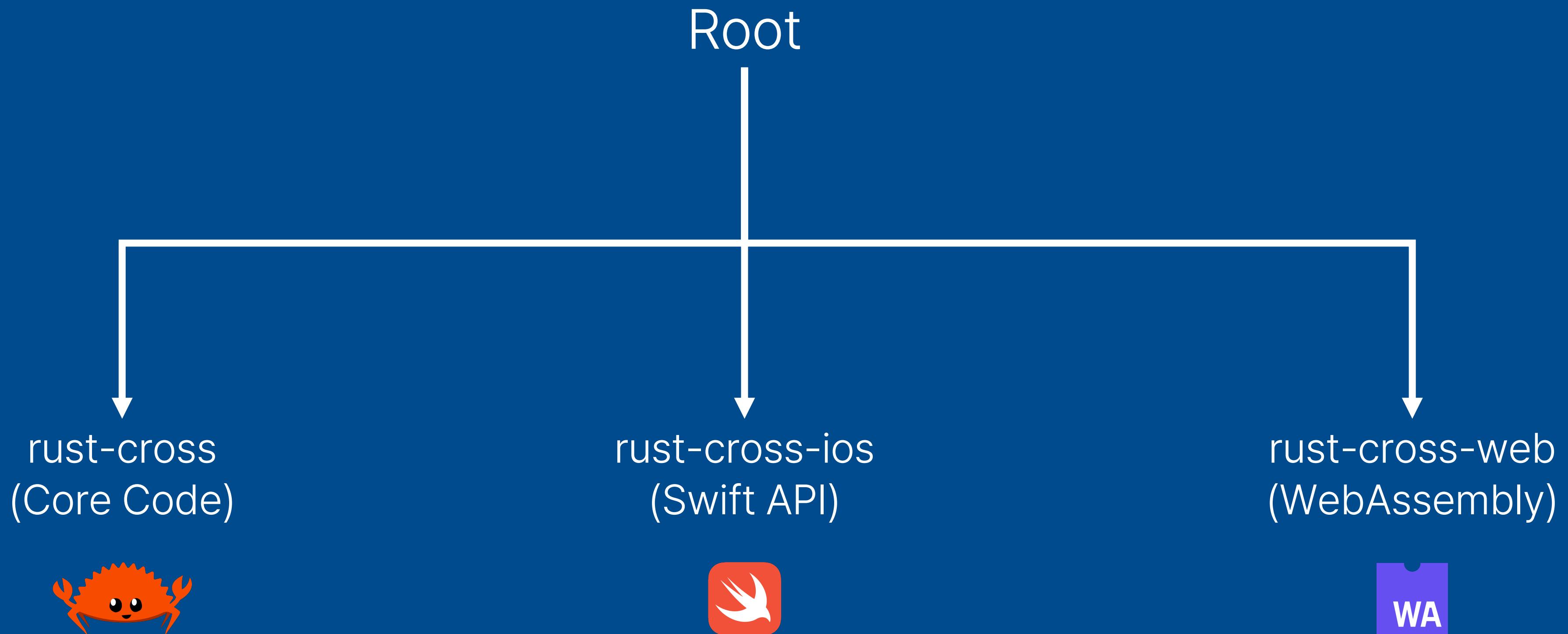
- 다양한 플랫폼에서 Rust 라이브러리에 있는 add 함수와 sub 함수를 호출해 결과를 받은 뒤 출력하는 예제를 만들어 보자.



프로젝트 구조

서울대학교 컴퓨터공학부 특강
Rust 크로스 플랫폼 프로그래밍

- rust-cross : 핵심 코드를 구현하는 크레이트. add 함수와 sub 함수를 구현한다.
- rust-cross-ios : Swift API를 만들기 위한 코드를 구현하는 크레이트.
- rust-cross-web : WebAssembly로 트랜스파일하기 위한 코드를 구현하는 크레이트.



프로젝트 구조 잡기

서울대학교 컴퓨터공학부 특강
Rust 크로스 플랫폼 프로그래밍

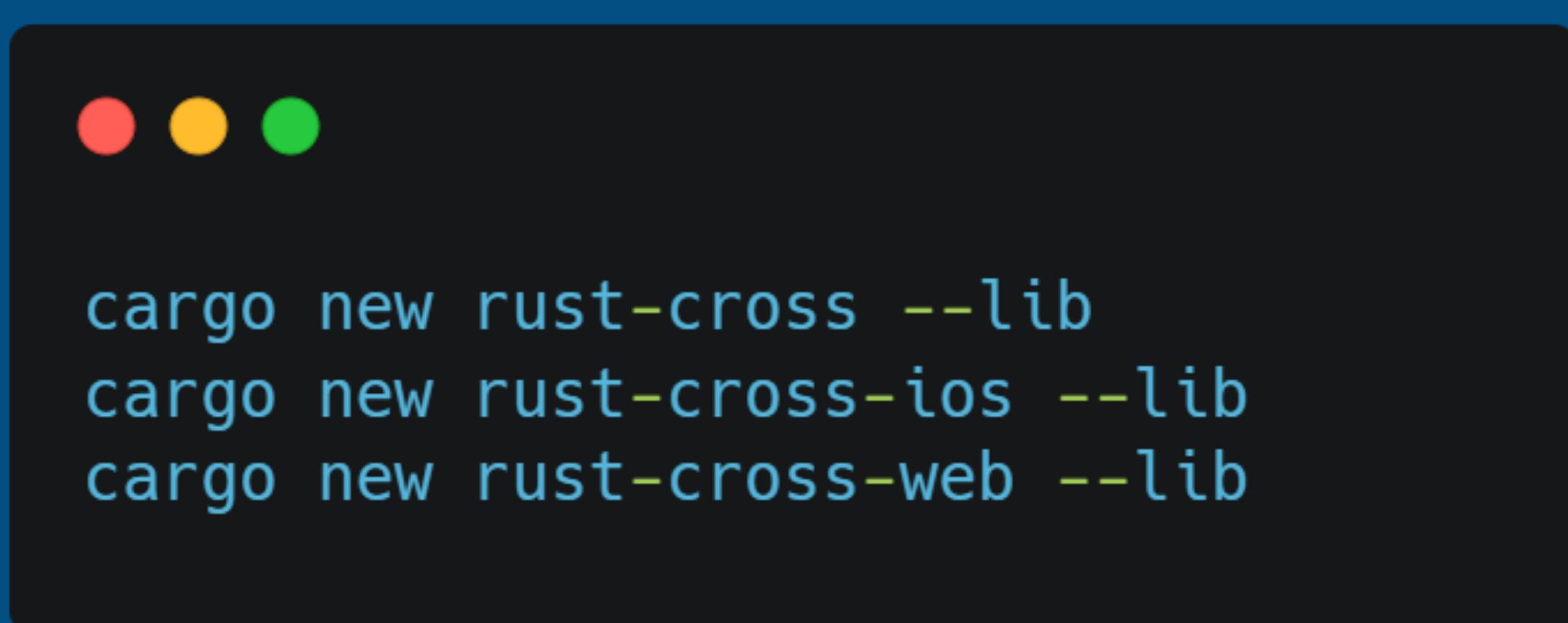
- 빈 디렉토리에 Cargo.toml 파일을 만든 뒤 다음과 같이 입력한다.

```
[workspace]
members = [
    "rust-cross",
    "rust-cross-ios",
    "rust-cross-web",
]
default-members = ["rust-cross"]
```

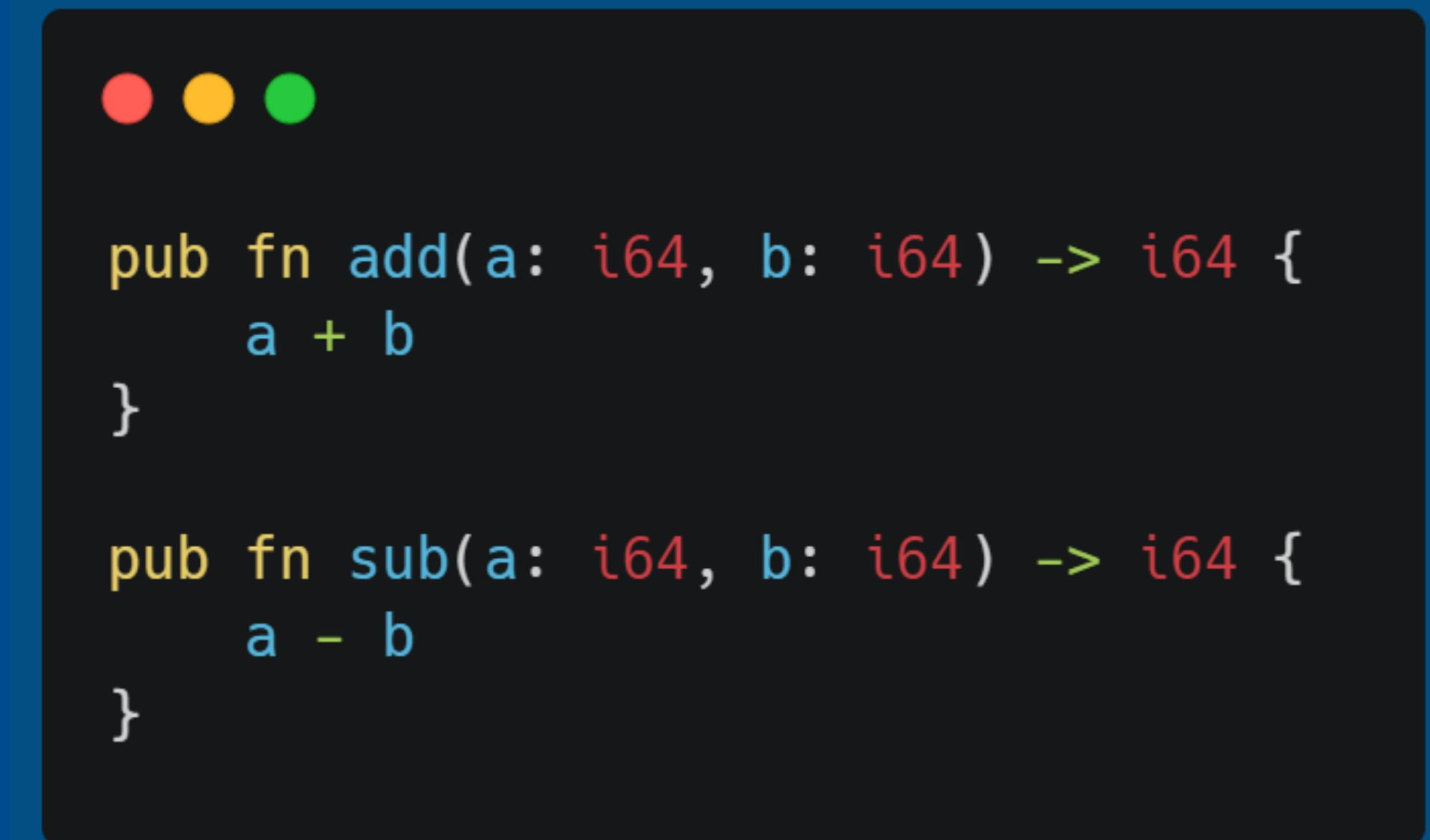
새 Rust 라이브러리 프로젝트 만들기

서울대학교 컴퓨터공학부 특강
Rust 크로스 플랫폼 프로그래밍

- cargo new 명령을 통해 만들 수 있다.
 - 라이브러리 프로젝트를 만들려면 --lib를 붙어야 한다.
 - 그렇지 않으면, 바이너리 파일을 생성하는 프로젝트가 만들어진다.



- add/sub 함수 구현하기
 - 외부에서 사용할 수 있도록 pub 키워드를 붙이자.
 - 구현을 완료했다면 cargo build --release 명령을 통해 빌드가 되는지 확인한다.



```
pub fn add(a: i64, b: i64) -> i64 {
    a + b
}

pub fn sub(a: i64, b: i64) -> i64 {
    a - b
}
```

- iOS를 위한 라이브러리 빌드 준비
 - Xcode를 설치한다.
 - App Store에서 설치할 수 있다.
 - Xcode 빌드 툴을 설치한다.
 - `xcode-select --install`
 - 그리고 크로스 컴파일이 가능하도록 iOS 아키텍처를 추가한다.
 - `rustup target add aarch64-apple-ios aarch64-apple-ios-sim x86_64-apple-ios`
 - 크로스 컴파일을 쉽게 할 수 있도록 도와주는 툴인 `cargo-lipo`를 설치한다.
 - `cargo install cargo-lipo`

- 프로젝트 설정

- Cargo.toml 파일을 연 뒤, 다음과 같이 수정한다.
 - 정적/동적 라이브러리를 만들기 위해 [lib]의 crate-type을 ["staticlib", "cdylib"]로 설정한다.
 - rust-cross 크레이트에 있는 함수들을 사용하므로 [dependencies]에 추가한다.

```
[package]
name = "rust-cross-ios"
version = "0.1.0"
edition = "2021"

[lib]
crate-type = ["staticlib", "cdylib"]

[dependencies]
rust-cross = { path = "../rust-cross" }
```

- C 브릿지에서 호출할 코드 작성
 - 함수의 이름이 맹글링되지 않도록 #[no_mangle]을 추가한다.
 - 맹글링(Mangling)이란 소스 코드에 선언된 함수/변수의 이름을 컴파일 단계에서 일정한 규칙을 갖고 변형하는 걸 말한다.

```
#[no_mangle]
pub extern "C" fn add(a: i64, b: i64) -> i64 {
    rust_cross::add(a, b)
}

#[no_mangle]
pub extern "C" fn sub(a: i64, b: i64) -> i64 {
    rust_cross::sub(a, b)
}
```

- WebAssembly를 위한 라이브러리 빌드 준비
 - Rust로 WebAssembly 바이너리 파일을 빌드해주는 툴인 wasm-pack을 설치한다.
 - curl <https://rustwasm.github.io/wasm-pack/installer/init.sh> -sSf | sh
 - JavaScript 번들러를 설치하고 실행할 수 있도록 패키지 매니저인 npm을 설치한다.
 - npm install npm@latest -g

- 프로젝트 설정

- Cargo.toml 파일을 연 뒤, 다음과 같이 수정한다.
 - 동적 시스템/러스트 라이브러리를 만들기 위해 [lib]의 crate-type을 ["cdylib", "rlib"]로 설정한다.
 - rust-cross 크레이트에 있는 함수들을 사용하므로 [dependencies]에 추가한다.
 - WebAssembly 빌드를 위해 wasm-bindgen을 [dependencies]에 추가한다.

```
● ● ●

[package]
name = "rust-cross-web"
version = "0.1.0"
edition = "2021"

[lib]
crate-type = ["cdylib", "rlib"]

[dependencies]
rust-cross = { path = "../rust-cross" }
wasm-bindgen = "0.2.48"
```

- WebAssembly로 빌드할 코드 작성
 - 필요할 경우 Wrapper로 처리할 수 있도록 #[wasm_bindgen]을 추가한다.
 - 구현을 완료했다면 wasm-pack build 명령을 통해 wasm 파일이 생성되는지 확인한다.

```
use wasm_bindgen::prelude::*;

#[wasm_bindgen]
pub fn add(a: i64, b: i64) -> i64 {
    rust_cross::add(a, b)
}

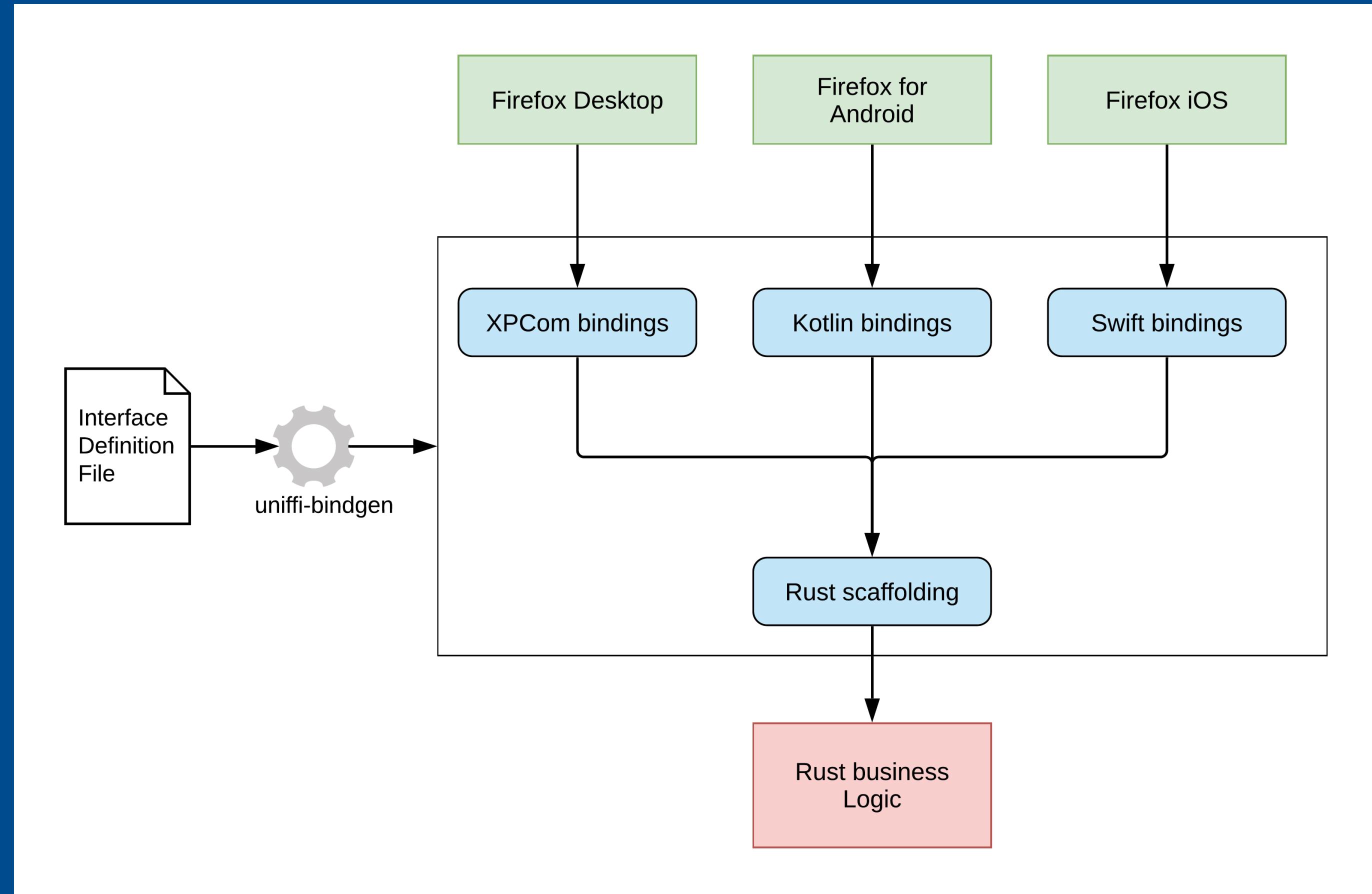
#[wasm_bindgen]
pub fn sub(a: i64, b: i64) -> i64 {
    rust_cross::sub(a, b)
}
```

- 이제 크로스 플랫폼 엔진을 만들 기반 코드를 갖췄다.
라이브러리가 필요한 각 플랫폼에 맞춰 빌드하면 된다. 하지만...
 - 버그를 수정하거나 새로운 기능을 추가하게 되면 지원 플랫폼의 수만큼 코드 작업을 해야 한다.
 - 코드를 한 벌로 만들기 위해 크로스 플랫폼 엔진을 개발했는데, 다시 여러 벌을 만들어야 하는 상황은 비효율적이다.
 - 어떻게 이 문제를 개선할 수 있을까?

리팩토링

서울대학교 컴퓨터공학부 특강
Rust 크로스 플랫폼 프로그래밍

- FFI도 한 벌로 만들어서 필요한 언어에 맞춰 라이브러리를 생성하면 어떨까?
→ uniffi-rs를 사용하면 된다. (<https://github.com/mozilla/uniffi-rs>)



리팩토링

서울대학교 컴퓨터공학부 특강
Rust 크로스 플랫폼 프로그래밍

- 라이브러리에 제공할 간단한 함수들을 만들어 보자.

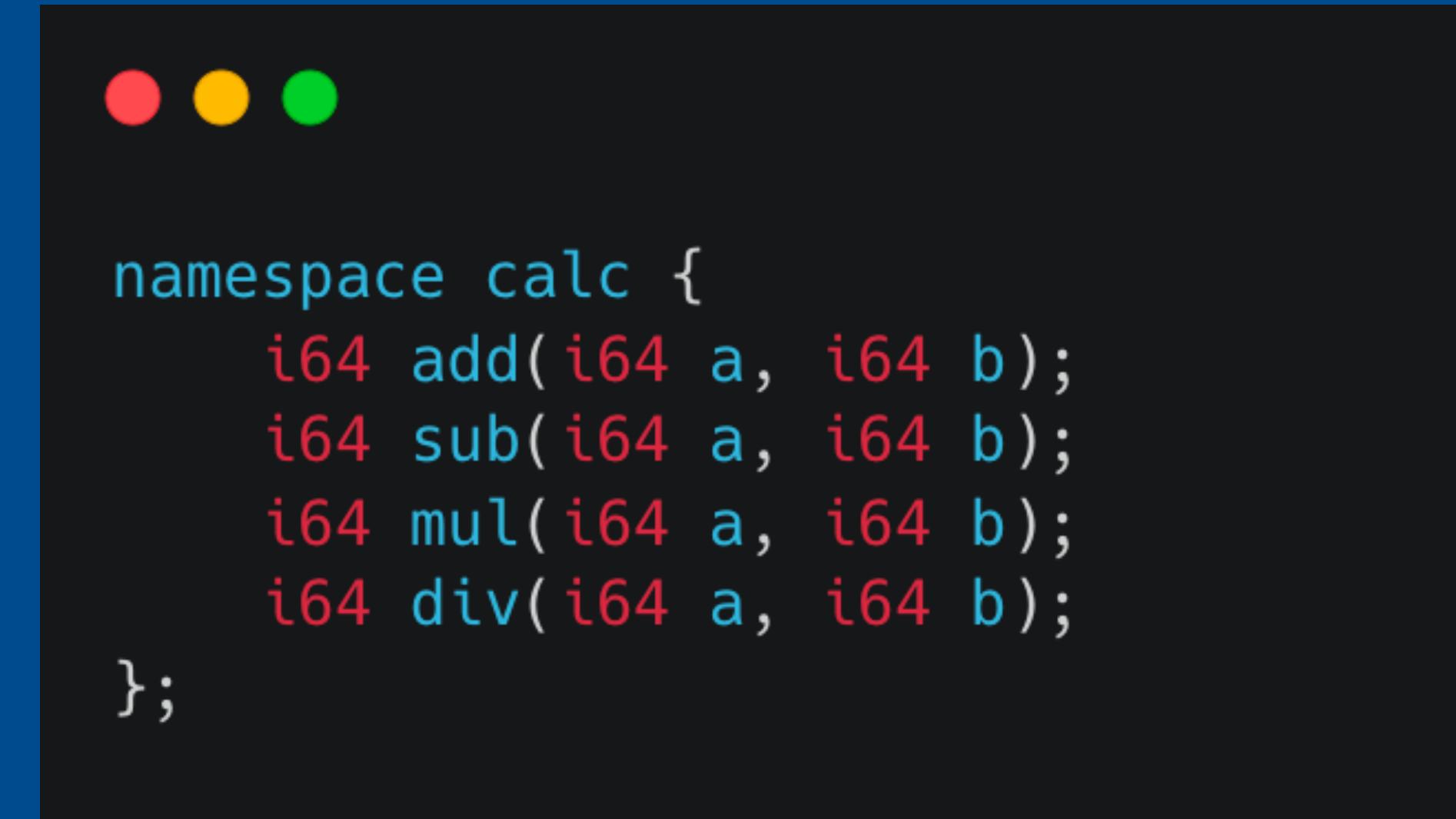
```
pub fn add(a: i64, b: i64) -> i64 {
    a + b
}

pub fn sub(a: i64, b: i64) -> i64 {
    a - b
}

pub fn mul(a: i64, b: i64) -> i64 {
    a * b
}

pub fn div(a: i64, b: i64) -> i64 {
    a / b
}
```

- 그리고 인터페이스 정의 언어(IDL; Interface Definition Language) 파일을 만든다.
이 파일을 통해 FFI 바인딩에 노출할 함수/메소드/타입 등을 정의한다.



```
namespace calc {
    i64 add(i64 a, i64 b);
    i64 sub(i64 a, i64 b);
    i64 mul(i64 a, i64 b);
    i64 div(i64 a, i64 b);
};
```

- 마지막으로 Rust 스캐폴딩 코드를 생성할 수 있도록 build.rs 파일을 만들고, lib.rs에 매크로를 추가해 생성된 스캐폴딩 코드를 포함하도록 한다.

build.rs

```
fn main() {
    uniffi_build::generate_scaffolding("./src/calc.udl").unwrap();
}
```

main.rs

```
uniffi_macros::include_scaffolding!("calc");
```

리팩토링

서울대학교 컴퓨터공학부 특강
Rust 크로스 플랫폼 프로그래밍

- 이제 Kotlin, Swift, Python, Ruby 중 원하는 언어로 내보내기가 가능하다.

```
// Kotlin
cargo run --bin uniffi-bindgen generate src/calc.udl --language kotlin

// Swift
cargo run --bin uniffi-bindgen generate src/calc.udl --language swift
```

3



몇 가지 시행 착오와 팁

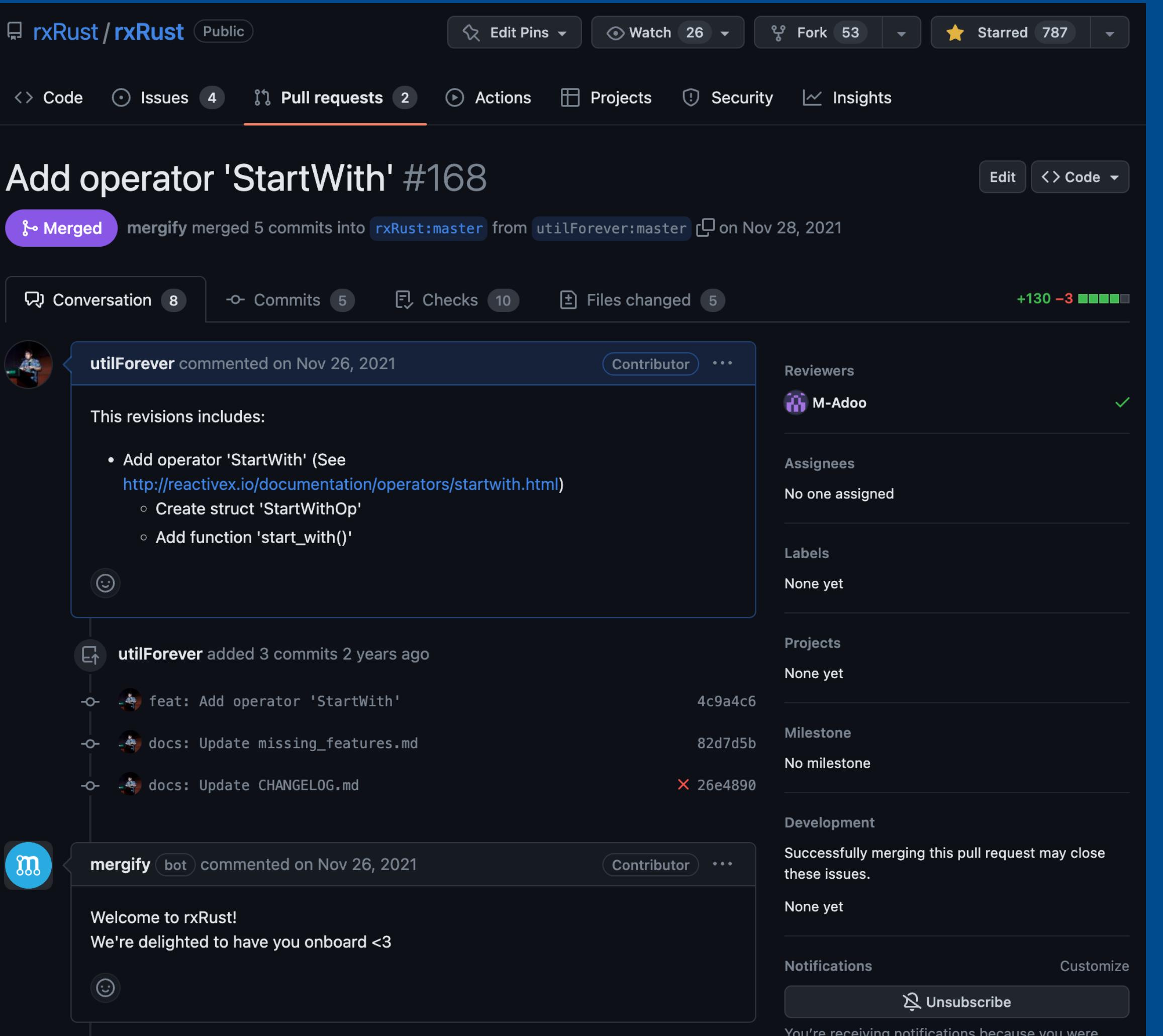
라이브러리 생태계 문제, Part 1

서울대학교 컴퓨터공학부 특강
Rust 크로스 플랫폼 프로그래밍

- Rust의 라이브러리 생태계는 계속 커지고 있지만, 아직은 부족한 부분들이 많다.
- 엔진에서 ReactiveX를 사용했었기에 Rust에 괜찮은 Rx 라이브러리가 있는지 조사했다.
하지만 원하는 기능을 모두 구현한 라이브러리는 존재하지 않았다.
- 방향성을 논의한 뒤, 최근까지 작업 기록이 있으면서 기능이 많이 구현된 rxRust(<https://github.com/rxRust/rxRust>)에 필요한 기능들을 구현하기로 결정했다.
- 경우에 따라 원하는 기능을 사용하기 위해 오픈 소스에 기여할 일이 생길 수 있다.

라이브러리 생태계 문제, Part 1

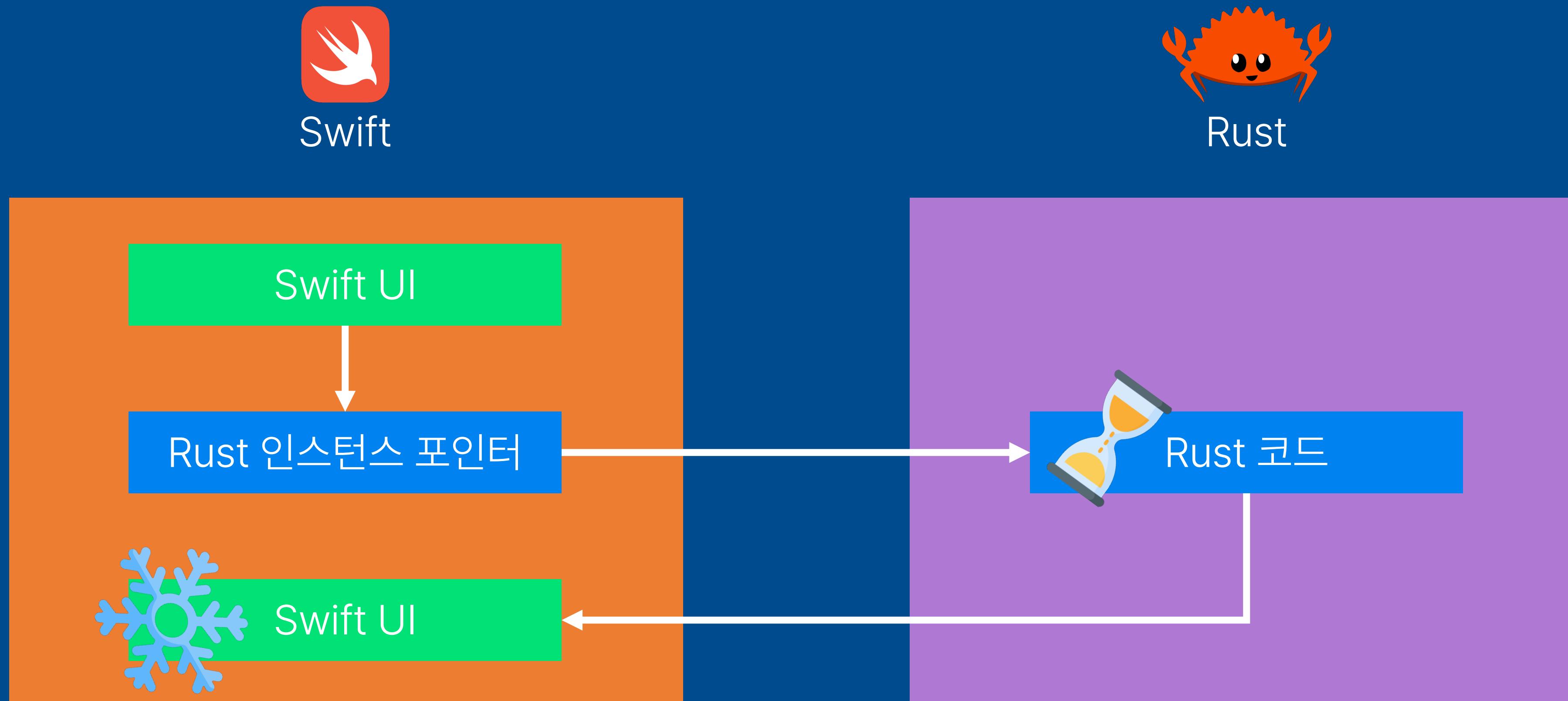
서울대학교 컴퓨터공학부 특강
Rust 크로스 플랫폼 프로그래밍



애플리케이션 프리징 문제

서울대학교 컴퓨터공학부 특강
Rust 크로스 플랫폼 프로그래밍

- Swift 앱에서 어떤 동작을 수행하기 위해 UI에서 버튼을 누르면, 엔진에서 처리되는 동안에 앱이 멈추는 문제가 발생한다.
- 원인은 Swift에서 필요한 함수들을 직접 호출하여, 모든 함수 호출은 동기적으로 동작하기 때문이다.



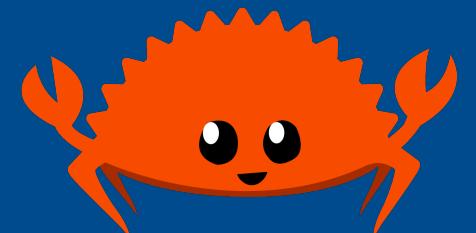
애플리케이션 프리징 문제

서울대학교 컴퓨터공학부 특강
Rust 크로스 플랫폼 프로그래밍

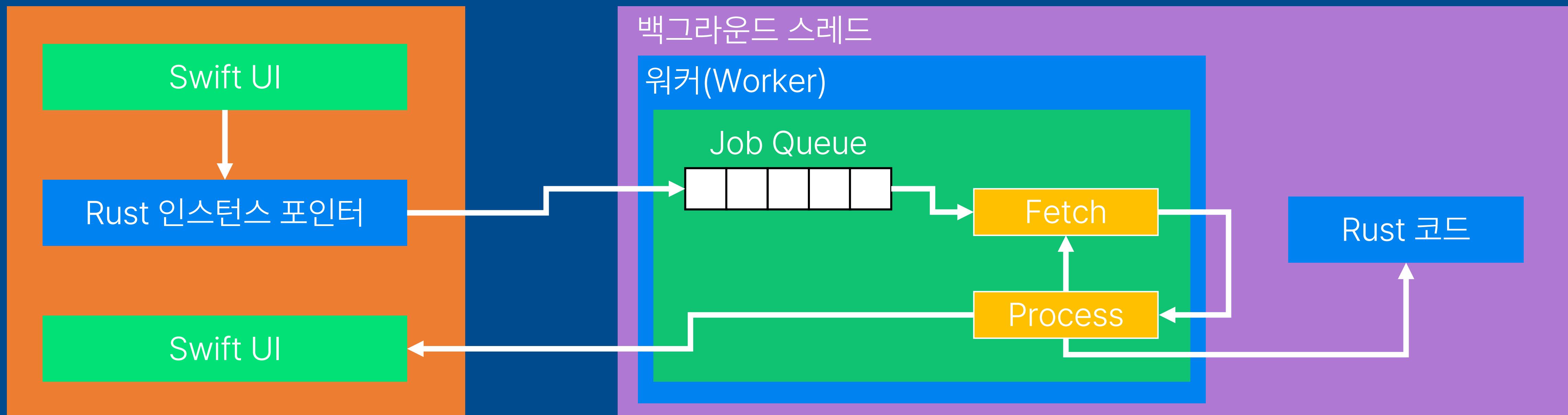
- 이를 해결하려면, 엔진에 대한 모든 요청을 처리하는 워커를 구현해 백그라운드 스레드에서 동작하게 만들어야 한다.
- 백그라운드 스레드에서 워커(Worker)를 통해 요청들을 Job Queue에 넣으면 작업을 비동기적으로 수행한다.



Swift



Rust



애플리케이션 프리징 문제

서울대학교 컴퓨터공학부 특강
Rust 크로스 플랫폼 프로그래밍

```
● ● ●

pub async fn main_loop(mut rx: mpsc::Receiver<Request>) {
    let context = Rc::new(Context::default());
    while let Some(request) = rx.recv().await {
        log::debug!("Get request (request: {request:?})");
        let cloned_context = Rc::clone(&context);
        spawn_local(async move {
            let result = process(cloned_context, request);
            basic_result_handle(result);
        });
    }
    log::info!("request channel is closed");
}
```

애플리케이션 프리징 문제

서울대학교 컴퓨터공학부 특강
Rust 크로스 플랫폼 프로그래밍

```
● ● ●

impl Default for Worker {
    fn default() -> Self {
        let (sender, receiver) = mpsc::channel(100);
        let handler = thread::Builder::new()
            .name("engine".into())
            .spawn(move || {
                log::debug!("Spawn thread");
                let rt = Builder::new_current_thread()
                    .build()
                    .expect("failed to create runtime");
                let local = LocalSet::new();
                local.block_on(&rt, main_loop(receiver));
            })
            .expect("failed to spawn background thread");
        Self {
            tx: sender,
            handler,
            rt: Runtime::new().expect("failed to create runtime"),
        }
    }
}
```

라이브러리 생태계 문제, Part 2

서울대학교 컴퓨터공학부 특강
Rust 크로스 플랫폼 프로그래밍

- iOS 라이브러리 지원 작업이 모두 끝난 뒤에, WebAssembly 지원 작업을 시작했다.
- 그런데 시작부터 큰 난관에 부딪혔다. rxRust가 WebAssembly 빌드를 지원하지 않는다!
- Rust 라이브러리 중에는 WebAssembly를 지원하지 않는 라이브러리들이 존재한다.
따라서 작업을 시작하기 전에 지원 여부를 꼭 확인해보기 바란다.
- 꼭 필요한 라이브러리라면 WebAssembly 빌드가 가능하게 작업을 해야 한다.

라이브러리 생태계 문제, Part 2

서울대학교 컴퓨터공학부 특강
Rust 크로스 플랫폼 프로그래밍

The screenshot shows a GitHub pull request page for the repository `rxRust/rxRust`. The pull request is titled "Support wasm build #187" and has been merged. The merge message indicates that 13 commits were merged from the `utilForever:master` branch into the `rxRust:master` branch on March 19, 2022.

The pull request details section shows the following information:

- Conversation:** 12 comments
- Commits:** 13
- Checks:** 11
- Files changed:** 49
- Code coverage:** +214 -93

The pull request body contains a comment from `utilForever` dated March 18, 2022, which includes a detailed list of changes:

- Support `wasm` build (Resolves [Add support for wasm #118](#), [rxrust = "0.15"](#) seems to be broken, and thus won't build #176 and [wasm support #177](#))
 - Add new feature 'wasm-scheduler'
 - I removed `SharedScheduler` when `feature` is `wasm-scheduler` and `target_arch` is `wasm32`
 - I tested it with simple example (<https://github.com/utilForever/rxrust-with-wasm>)

Below the comment, there is a reaction icon for thumbs up (1) and a link to the commit `feat: Support wasm build` with hash `bd49233`.

The right sidebar provides additional details about the pull request:

- Reviewers:** M-Adoo (checked)
- Assignees:** No one assigned
- Labels:** None yet
- Projects:** None yet
- Milestone:** No milestone
- Development:** Successfully merging this pull request may close these issues.
 - [Add support for wasm](#)
- Notifications:** Customize

WebAssembly 함수 맹글링 문제

서울대학교 컴퓨터공학부 특강

Rust 크로스 플랫폼 프로그래밍

- Rust + WebAssembly로 빌드하면 함수 이름들이 맹글링된다.
 - 맹글링된 함수 이름 때문에 콜 스택을 제대로 확인할 수 없는 문제가 있다.

✖▼panicked at 3:22 consoleObservable.ts:43

Stack:

Error

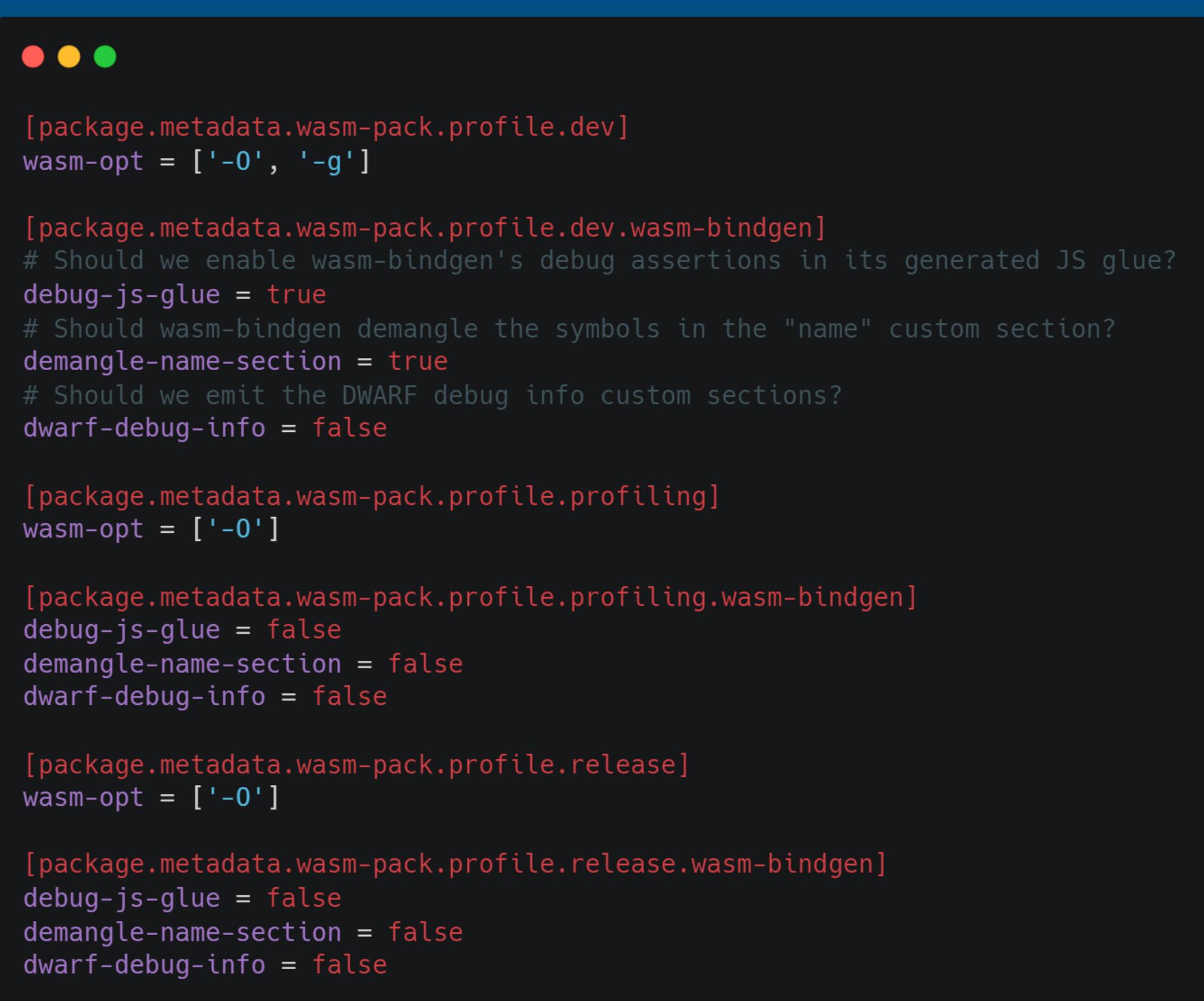
```
at [REDACTED]
at [REDACTED]
at [REDACTED]
at http://192.168.1.26:8085/c9d8136...module.wasm:wasm-function[7827]:0x15f69e
at http://192.168.1.26:8085/c9d8136...module.wasm:wasm-function[1448]:0xc0b5c
at http://192.168.1.26:8085/c9d8136...module.wasm:wasm-function[8213]:0x16308a
at http://192.168.1.26:8085/c9d8136...module.wasm:wasm-function[7479]:0x15b626
at http://192.168.1.26:8085/c9d8136...module.wasm:wasm-function[2927]:0x101871
at http://192.168.1.26:8085/c9d8136...module.wasm:wasm-function[4591]:0x12fd0a
at http://192.168.1.26:8085/c9d8136...module.wasm:wasm-function[8050]:0x161951
```

console.<computed> @ consoleObservable.ts:43
(anonymous) @
logError @
__wbg_error_09919627ac0992f5 @
\$func3992 @ c9d8136....odule
\$func1448 @ c9d8136...modul
\$func8213 @ c9d8136....odule
\$func7479 @ c9d8136....odule
\$func2927 @ c9d8136....odule
\$func4591 @ c9d8136....odule
\$func8050 @ c9d8136....odule
\$func8196 @ c9d8136....odule
\$func7466 @ c9d8136....odule
\$func8594 @ c9d8136....odule
\$func8672 @ c9d8136....odule

WebAssembly 함수 맹글링 문제

서울대학교 컴퓨터공학부 특강
Rust 크로스 플랫폼 프로그래밍

- 이 문제를 해결하려면 WebAssembly 코드를 배포하는 크레이트의 Cargo.toml에 각 빌드에 따른 설정을 해줘야 한다.
디버깅을 위해, dev 빌드에서 demangle-name-section을 true로 설정하고 wasm-opt에 플래그 -g를 추가한다.
- 이제 wasm 파일이 빌드되면 웹쪽 코드에 pkg 폴더를 붙여넣기하고 실행하면 된다.



```
[package.metadata.wasm-pack.profile.dev]
wasm-opt = ['-O', '-g']

[package.metadata.wasm-pack.profile.dev.wasm-bindgen]
# Should we enable wasm-bindgen's debug assertions in its generated JS glue?
debug-js-glue = true
# Should wasm-bindgen demangle the symbols in the "name" custom section?
demangle-name-section = true
# Should we emit the DWARF debug info custom sections?
dwarf-debug-info = false

[package.metadata.wasm-pack.profile.profiling]
wasm-opt = ['-O']

[package.metadata.wasm-pack.profile.profiling.wasm-bindgen]
debug-js-glue = false
demangle-name-section = false
dwarf-debug-info = false

[package.metadata.wasm-pack.profile.release]
wasm-opt = ['-O']

[package.metadata.wasm-pack.profile.release.wasm-bindgen]
debug-js-glue = false
demangle-name-section = false
dwarf-debug-info = false
```

WebAssembly 함수 맹글링 문제

서울대학교 컴퓨터공학부 특강
Rust 크로스 플랫폼 프로그래밍

- npm을 실행하면 Failed to decode custom "name" section @2024365; ignoring (Maximum call stack size exceeded) 오류가 발생할 때가 있다.
- 이 문제를 해결하려면 node를 실행할 때 --stack-size=10000을 추가해서 스택 크기를 늘려야 한다.
- 보안 이슈로 인해 NODE_OPTIONS를 사용할 수는 없고, package.json에서 npm start에 해당하는 부분에 추가하면 된다.
- 심볼 정보를 전부 포함한 채로 실행하기 때문에 불러오는데 꽤 오랜 시간이 걸릴 수 있다. (인내하고 기다리자!)

WebAssembly 함수 맹글링 문제

서울대학교 컴퓨터공학부 특강

Rust 크로스 플랫폼 프로그래밍

- 이제 잘 실행되고 디明朗된 함수 이름들을 볼 수 있다.

✖▼ panicked at 3:22 consoleObservable.ts:43

Stack:

```
  Error
    at [REDACTED]
    at [REDACTED]
    at [REDACTED]
    at console_error_panic_hook::Error::new::habaaa7a517e69b73 (http://192.168.1.26:8085/380a79c...module.wasm:wasm-function\[7827\]:0x15f78)
    at console_error_panic_hook::hook_impl::he97eccf8e518f1c3 (http://192.168.1.26:8085/380a79c...module.wasm:wasm-function\[1448\]:0xc0c07)
    at console_error_panic_hook::hook::h1ef0976bed306fb0 (http://192.168.1.26:8085/380a79c...module.wasm:wasm-function\[8213\]:0x163173)
    at core::ops::function::Fn::call::hfd142f2b7bbd24f1 (http://192.168.1.26:8085/380a79c...module.wasm:wasm-function\[7479\]:0x15b70f)
    at std::panicking::rust_panic_with_hook::h56d066e3564322fb (http://192.168.1.26:8085/380a79c...module.wasm:wasm-function\[2927\]:0x10192)
    at std::panicking::begin_panic_handler::{closure}::he926425bf39194d7 (http://192.168.1.26:8085/380a79c...module.wasm:wasm-function\[4591\]:0x12fdf1)
    at rust_begin_unwind (http://192.168.1.26:8085/380a79c...module.wasm:wasm-function\[8050\]:0x161a3a)  
  
console.<computed> @ consoleObservable.ts:43:1
  (anonymous) @ consoleObservable.ts:43:1
  logError @ consoleObservable.ts:43:1
  __wbg_error_09919627ac0992f5 @ consoleObservable.ts:43:1
  $console_error_panic_hook::error::h881975d7e773fe08 @ 380a79c...module.wasm:wasm-function\[7827\]:0x15f78
  $console_error_panic_hook::hook_impl::he97eccf8e518f1c3 @ 380a79c...module.wasm:wasm-function\[1448\]:0xc0c07
  $console_error_panic_hook::hook::h1ef0976bed306fb0 @ 380a79c...module.wasm:wasm-function\[8213\]:0x163173
  $core::ops::function::Fn::call::hfd142f2b7bbd24f1 @ 380a79c...module.wasm:wasm-function\[7479\]:0x15b70f
  $std::panicking::rust_panic_with_hook::h56d066e3564322fb @ 380a79c...module.wasm:wasm-function\[2927\]:0x10192
  $std::panicking::begin_panic_handler::{closure}::he926425bf39194d7 @ 380a79c...module.wasm:wasm-function\[4591\]:0x12fdf1
  $rust_begin_unwind @ 380a79c...module.wasm:wasm-function\[8050\]:0x161a3a
  $core::panicking::panic_fmt::h0a20ba97f16cb738 @ 380a79c...module.wasm:wasm-function\[7827\]:0x15f78
  $core::panicking::panic_display::h02eec950c6198561 @ 380a79c...module.wasm:wasm-function\[1448\]:0xc0c07
  $core::panicking::panic_str::haedd4881c3b18d7d @ 380a79c...module.wasm:wasm-function\[8213\]:0x163173
  $core::option::expect_failed::h7b17b0c26e58b8b9 @ 380a79c...module.wasm:wasm-function\[7479\]:0x15b70f
  $core::option::Option<T>::expect::h63b3ac7991ec1912 @ 380a79c...module.wasm:wasm-function\[2927\]:0x10192  
  
$<rxrust::observable::observable next::ObserverN<N, Item> as rxrust::observer::Observer>::next::h64e99edc0e0cd0af @ 380a79c...module.wasm:wasm-function\[7827\]:0x15f78
```

WebAssembly 프리징 문제

서울대학교 컴퓨터공학부 특강
Rust 크로스 플랫폼 프로그래밍

- 작업이 얼추 마무리되고 QA가 시작되었는데 rxRust의 특정 연산을 실행했을 때 주기적으로 프리징이 걸려 비정상적으로 동작하는 문제가 발생했다.
- 이 문제는 iOS에서는 발생하지 않았던 문제라서 원인을 추적하기 쉽지 않았다.
(실제로 문제의 원인을 정확히 파악하는 데 1달, 고치는 방법을 찾는데 1달 걸렸다.)

WebAssembly 프리징 문제

서울대학교 컴퓨터공학부 특강
Rust 크로스 플랫폼 프로그래밍

- 문제는 Timer에 있었다. Rust로 작성된 WebAssembly 타이머들을 살펴 보면 대부분 커스텀으로 구현된 로직을 볼 수 있는데 모두 프리징 문제가 발생했다.
Ex) <https://github.com/tomaka/wasm-timer/blob/master/src/timer/interval.rs>
- 이 문제를 해결하려면 JavaScript에 있는 Interval / TimeOut 함수들을 사용하는 타이머로 교체해야 한다. rustwasm에서 만든 gloo-timers가 이렇게 구현되어 있다.
Ex) <https://github.com/rustwasm/gloo/blob/master/crates/timers/src/future.rs>
Ex) <https://github.com/rustwasm/gloo/blob/master/crates/timers/src/callback.rs>

WebAssembly 프리징 문제

서울대학교 컴퓨터공학부 특강
Rust 크로스 플랫폼 프로그래밍

rxRust / rxRust Public Edit Pins Watch 26 Fork 53 Starred 787

Code Issues 4 Pull requests 2 Actions Projects Security Insights

Fix problem that operator 'interval()' is delayed in WebAssembly #198

Merged mergify merged 3 commits into rxRust:master from utilForever:master on Oct 28, 2022

Conversation 3 Commits 3 Checks 11 Files changed 4 +24 -10

utilForever commented on Oct 26, 2022

This revision includes:

- Fix problem that operator 'interval()' is delayed in WebAssembly
 - Add crate 'gloo-timers'
 - Replace code 'fluvio_wasm_timer::Interval' with 'IntervalStream'

utilForever added 2 commits 6 months ago

- fix: Add crate 'gloo-timers' for WebAssembly df5197b
- fix: Replace code 'fluvio_wasm_timer::Interval' with 'IntervalStream' 91aee48

mergify bot commented on Oct 26, 2022

Welcome to rxRust!
We're delighted to have you onboard <3

Reviewers M-Adoo ✓

Assignees No one assigned

Labels None yet

Projects None yet

Milestone No milestone

Development Successfully merging this pull request may close these issues.

None yet

Notifications Customize

- Rust 크로스 플랫폼 프로그래밍은 매력적인 선택지다.
- 아직은 트러블슈팅에 대한 정보가 많지 않아, 시행 착오가 꽤 많을 수 있다.
- 각 언어의 바인딩을 쉽게 할 수 있도록 도와주는 라이브러리들이 있으니 잘 사용하자.
ex) Rustler (Elixir), uniffi-rs (Kotlin, Swift, Python, Ruby), flutter_rust_bridge (Dart)
- 엔진의 동작으로 인해 애플리케이션이 프리징이 발생한다면,
각 작업을 큐에 넣고 비동기적으로 처리할 수 있도록 Worker를 만들자.
- WebAssembly를 작업하기 전에, 라이브러리들이 빌드를 지원하는지 먼저 확인하자.
- 라이브러리에 원하는 기능이 구현되어 있지 않은 경우, 오픈 소스에 기여할 기회가 생긴다.
- 여러 플랫폼에 같은 기능을 구현해야 되고, 성능도 고려하고 싶고, 웹 지원도 하고 싶다면?
지금 바로 Rust로 시작해보기 바란다. 꽤 재미있는 여정이 될 것이다.

Thank you!