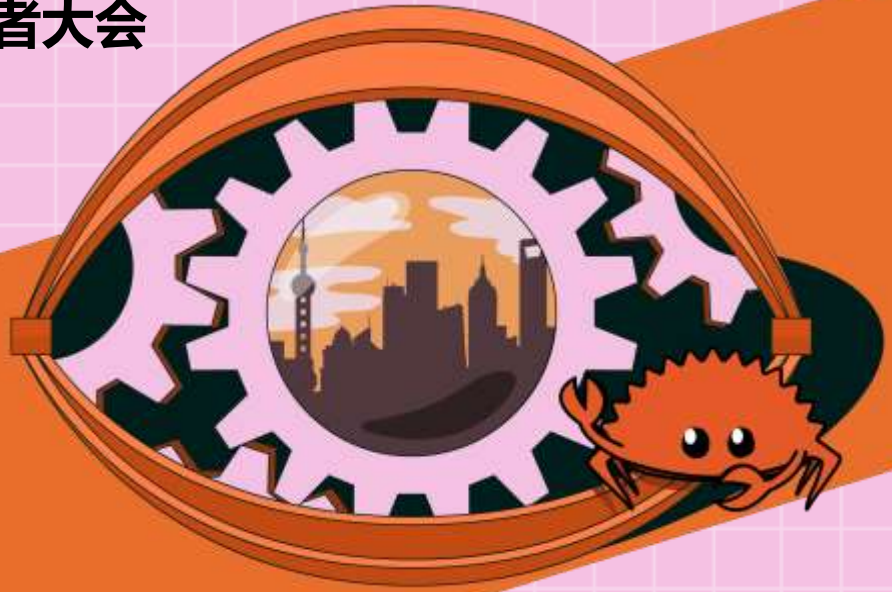
RUST CHINA CONF 2023

第三届中国Rust开发者大会



6.17-6.18 @Shanghai

WebAssembly 介绍

WebAssembly 简介

WebAssembly(简称 Wasm)是一种新的编译目标,帮助在 web 中运行高性能应用。它是一种低级语言,设计为编译器目标，以在 web 浏览器中高效运行。

WebAssembly 简介

高性能

WebAssembly 代码可以以接近原生的速度运行,且具有很小的二进制大小和快速加载速度。

标准稳定

WebAssembly 最初由 Mozilla、Google、Microsoft 等主要浏览器供应商共同设计。它现已在所有主流浏览器中实现,包括 Chrome、 Firefox、 Safari 和 Edge。

多语言支持

现在多种语言都有编译器支持 WebAssembly,如 C/C++、 Rust、 Go、 Zig 等。

安全性

WebAssembly 设计为安全地嵌入到网页中。它提供一种沙箱环境,禁止直接访问浏览器功能或用户数据。而是需要通过 host function 来访问宿主环境。

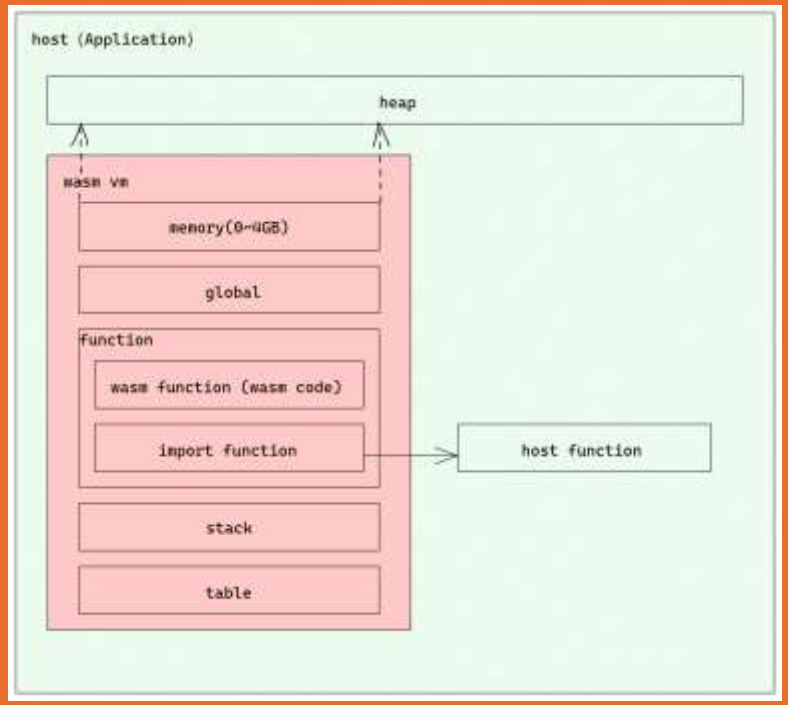
WebAssembly 机制

memory

Wasm 的 memory 是 host 内存中的一部分。
对于 Wasm 而言，这块内存是从 0 开始的，而不是 host 所看到的地址。

function

Wasm 编写的 function 可以通过 “export” 导出给 host 调用。
Host 可以把自己的 function 通过 “import” 提供给 wasm 调用。



WebAssembly 简介

Wasm 实例

c > C main.c > ...

```c
int hello() __attribute__((
    __import_module__("env"),
    __import_name__("hello")
));

int main(){
  return hello();
}
```

c > WA main.wat

```
(module
 (type $none_=>_i32 (func (result i32)))
 (type $i32_=>_none (func (param i32)))
 (type $none_=>_none (func))
 (import "env" "hello" (func $hello (result i32)))
 (import "wasi_snapshot_preview1" "proc_exit" (func $wasi.proc_exit (param i32)))
 (global $_stack_pointer (mut i32) (i32.const 66560))
 (memory $0 2)
 (export "memory" (memory $0))
 (export "_start" (func $_start))
 (func $_start
  (local $0 i32)
  (local $1 i32)
  (global.set $_stack_pointer
  )
  (i32.store offset=12
  )
  (local.set $1
   (call $hello)
  )
  (global.set $_stack_pointer
  )
  (if
   (local.get $1)
   (block
    (call $wasi.proc_exit
     (local.get $1)
    )
    (unreachable)
   )
  )
 )
 ;; custom section "producers", size 108
)
```

WebAssembly 介绍

WASI 简介

WebAssembly System Interface （简称 WASI）,它定义了一组 WASM 模块可以调用的系统调用接口。WASI 的目的是让 WASM 模块可以访问底层系统的功能，比如文件系统、网络等。这使得 WASM 可以作为一个更广泛的运行时,不仅仅局限于浏览器环境。WASI 当前定义了一组 POSIX 兼容的系统调用,让 WASM 模块可以访问文件系统。未来 WASI 还会加入更多系统接口,为 WASM 提供更广泛的系统访问能力。

WASI 实例

WASI 的本质就是一套 host 提供的 function。
与开发者自行提供的 host function 相比，
WASI 在 Rust 被内置在 std 中。

```rust
fn main() {
    println!("Hello, WASM!");
    std::fs::write(path: "./data", contents: "Hello, WASI").unwrap();
}
```

src/main.rs

```
(type $i64_i32_i32_=>_i32 (func (param i64 i32 i32) (result i32)))
(import "wasi_snapshot_preview1" "fd_write" (func $wasi::lib_generat
(import "wasi_snapshot_preview1" "path_open" (func $wasi::lib_genera
(import "wasi_snapshot_preview1" "environ_get" (func $__imported_was
(import "wasi_snapshot_preview1" "environ_sizes_get" (func $__import
(import "wasi_snapshot_preview1" "fd_close" (func $__imported_wasi_s
(import "wasi_snapshot_preview1" "fd_prestat_get" (func $__imported_
(import "wasi_snapshot_preview1" "fd_prestat_dir_name" (func $__impo
(import "wasi_snapshot_preview1" "proc_exit" (func $__imported_wasi_
(global $global$0 (mut i32) (i32.const 1048576))
(memory $0 17)
```

WASM 的应用场景

由于 WASI ， WASM 不仅可以在浏览器中运行，其作为一种通用二进制格式，也适用于浏览器外的许多场景:

1. 物联网设备: WASM 体积小、加载快,很适合运行在物联网设备上。使用 WASM 可以让这些设备运行更复杂的逻辑,实现设备间的互操作性。

2. 云计算: WASM 模块可以部署在云端运行,为用户提供服务。因为 WASM 是sandbox的,所以可以保证代码的安全性。WASM 的模块化也让云端应用更易于构建和部署。

3. 用户定义函数(UDF)： WASM UDF 安全性更高。WASM 运行在沙箱中,访问受限,可以防止恶意 UDF 对数据和系统产生破坏。与解释执行的 UDF 相比,WASM 作为二进制格式可以获得更高的运行性能。

WASM 使用场景和问题

WASM 中 IO 阻塞问题

在 WASI 和 一些用户自定义的 Host function 中，难免存在一些如网络服务的阻塞行为。当在 tokio 之类的 async runtime 中执行一些特别的 WASM 时就会遇到 WASM 阻塞 tokio 最终导致服务不可用的情况。

WASM 使用场景和问题

阻塞示例

```rust
#[tokio::main(flavor = "current_thread")]
▶ Run | Debug
async fn main() {
    env_logger::init();
    async fn async_ticker() {
        let mut i: i32 = 0;
        loop {
            println!("[host] tick {i}");
            tokio::time::sleep(std::time::Duration::from_secs(1)).await;
            i += 1;
        }
    }

    let _ = tokio::spawn(async_ticker());
    sync_run_demo();
}
```

```rust
fn main() {
    println!("[wasm] wait 3s...");
    std::thread::sleep(std::time::Duration::from_secs(3));
    println!("[wasm] exit");
}
```

```
[wasm] wait 3s...
[wasm] exit
[run_demo] Ok([])
➜  wasm_timeout git:(main) ✗
```

Async Wasm 解决方案

利用语言本身 Async 机制

因为 Rust 的 async 机制是无栈协程，会将 async 部分在编译时隐式转换成一个 Future。
所以我们可以利用这一点来实现一个 Async 的 Wasm。

利用本身 Async 机制

自行实现 Async Runtime

- 在 wasm 中把 future 存入固定内存处。
- 导出 poll 函数给 host 调用。
- 把 host function 包装成自定义 Future。

利用本身 Async 机制

优点

- 实现简单

缺点

- 方案不通用（wasm 局限于某一种语言）
- 无法与现有生态配合

Async Wasm 解决方案

基于 fiber / ucontext

wasmtime-fiber

wasmtime-fiber 是一个通过内联汇编，保存当前寄存器和栈数据来实现有栈协程的 rust 库。

ucontext

Ucontext 和 fiber 功能相同，但是 linux 的系统库。

Async Wasm 解决方案

执行流程

基于 fiber / ucontext

优点

- 与 WASM 的语言无关。
- 可以复用 WASM 编写语言本身的生态。
- 不会对 WASM 执行产生性能损失

缺点

- 实现困难，涉及到汇编。容易出错。
- 需要极其注意内存安全。

Async Wasm 解决方案

效果示例

```rust
#[tokio::main(flavor = "current_thread")]
▶ Run | Debug
async fn main() {
    env_logger::init();
    async fn async_ticker() {
        let mut i: i32 = 0;
        loop {
            println!("[host] tick {i}");
            tokio::time::sleep(std::time::Duration::from_secs(1)).await;
            i += 1;
        }
    }

    let _ = tokio::spawn(async_ticker());
    run_demo().await;
}
```

```rust
fn main() {
    println!("[wasm] wait 3s...");
    std::thread::sleep(std::time::Duration::from_secs(3
    println!("[wasm] exit");
}
```

```
[wasm] wait 3s...
[host] tick 0
[host] tick 1
[host] tick 2
[wasm] exit
[run_demo] Ok([])
→  wasm_timeout git:(main) ×
```

Async Wasm 解决方案

基于 Asyncify(Binaryen)

Binaryen 是一个编译器基础架构库，提供了一套用于处理 WebAssembly 的工具。
其中一个功能是 asyncify，它允许将同步的 WebAssembly 代码转换为异步代码。
就像 rust 对 async function 做的事情一样。

Async Wasm 解决方案

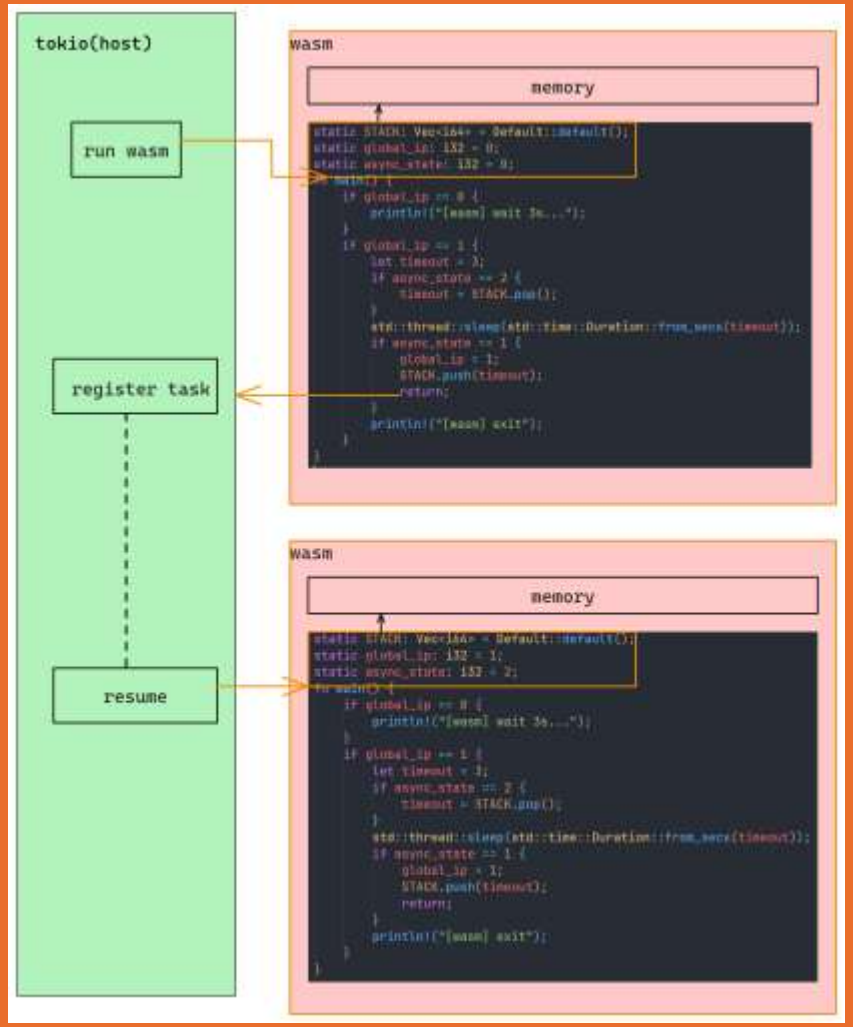
Asyncify 原理示意

```rust
fn main() {
    println!("[wasm] wait 3s...");
    std::thread::sleep(std::time::Duration::from_secs(3));
    println!("[wasm] exit");
}
```

```rust
static STACK: Vec<i64> = Default::default();
static global_ip: i32 = 0;
static async_state: i32 = 0;
fn main() {
    if global_ip == 0 {
        println!("[wasm] wait 3s...");
    }
    if global_ip == 1 {
        let timeout = 3;
        if async_state == 2 {
            timeout = STACK.pop();
        }
        std::thread::sleep(std::time::Duration::from_secs(timeout));
        if async_state == 1 {
            global_ip = 1;
            STACK.push(timeout);
            return;
        }
        println!("[wasm] exit");
    }
}
```

Async Wasm 解决方案

执行流程



Async Wasm 解决方案

Asyncify(Binaryen)

优点

- 与 WASM 的语言无关。
- 与 CPU 汇编指令无关。
- **可以跨机器调度。**

缺点

- 运行效率有所下降。

Async Wasm 解决方案

Asyncify 跨机器调度

```rust
#[tokio::main]
async fn main() {
    simple_log::quick!("warn");
    // read wasm
    let wasm: Vec<u8> = load_wasm_bytes("wasm/demo.wasm");

    // pass async module
    let async_wasm: Cow<'_, [u8]> = pass_async_module(&wasm).unwrap();

    let config: Option<Config> = Config::create();
    let mut snapshot: Option<InstanceSnapshot> = None;
    let mut wasi_snapshot: Option<SerialWasiCtx> = None;

    if let Ok(f: File) = std::fs::File::open("./snapshot") {
        let Task { task: (SerialInstanceSnapshot, _), .. } = rmp_serde::from_read(f).unw
        let (s: SerialInstanceSnapshot, wasi: SerialWasiCtx) = task;

        snapshot = Some(s.into());
        wasi_snapshot = Some(wasi);
    }

    match run_wasm(
        &config,
        &async_wasm,
        (snapshot.take(), wasi_snapshot.take()),
    )
```

```rust
match run_wasm(
    &config,
    &async_wasm,
    (snapshot.take(), wasi_snapshot.take()),
)
.await
{
    Ok(_r: Vec<WasmVal>) => {
        log::warn!("wasm exit");
    }
    Err((s: InstanceSnapshot, wasi: SerialWasiCtx)) => {
        let ddl: SystemTime = if let IoState::Sleep { ddl: &SystemTime } = &wasi.i
        } else {
        };
        let task: Task = Task {
        };
        let data: Vec<u8> = rmp_serde::to_vec_named(&task).unwrap();
        std::fs::write("./snapshot", data).unwrap();
        log::warn!("save snapshot");
    }
}
```

Asyncify 跨机器调度

```rust
fn main() {
    println!("[wasm] wait 3s...");
    std::thread::sleep(std::time::Duration::from_secs(3));
    println!("[wasm] exit");
}
```

```
➜  wasmedge_asyncify git:(std-asyncify) ✗ cargo run --package yield_sleep
   Compiling yield_sleep v0.1.0 (/home/csh/workspace/wasmedge_asyncify/examples/yield_sleep)
    Finished dev [unoptimized + debuginfo] target(s) in 36.23s
     Running `target/debug/yield_sleep`
[wasm] wait 3s...
2023-06-15 19:01:26.011767522 [WARN] <yield_sleep:169>:save snapshot
➜  wasmedge_asyncify git:(std-asyncify) ✗ cargo run --package yield_sleep
    Finished dev [unoptimized + debuginfo] target(s) in 0.84s
     Running `target/debug/yield_sleep`
[wasm] exit
2023-06-15 19:01:35.107572269 [WARN] <yield_sleep:154>:wasm exit
➜  wasmedge_asyncify git:(std-asyncify) ✗ ls -lh snapshot
-rw-r--r-- 1 csh csh 1.1M Jun 15 19:01 snapshot
➜  wasmedge_asyncify git:(std-asyncify) ✗
```

Thank you!