

WebAssembly 与 Rust 实践

简介

Rust 编程语言和 WebAssembly (Wasm) 都是由 Mozilla 推动的前沿技术，并且它们在许多方面相互补充。**Rust 正致力于成为 WebAssembly 的首选编程语言。**

Rust 语言的最大特色之一是在保证系统性能的同时提供内存安全性。通过其所有权系统、借用检查器和生命周期规则，Rust 可以在编译时捕获内存安全错误，避免了许多常见的内存错误，如空指针引用、数据竞争等。这使得 Rust 成为开发高性能、安全和可靠系统级软件的理想选择。

WebAssembly 是一种可移植的低级字节码格式，可以在 Web 环境中运行。它的目标是在不同平台和语言之间提供统一的执行环境，并以接近原生的性能运行代码。WebAssembly 允许以高效且安全的方式在 Web 浏览器中执行性能敏感的任务，同时保持与底层平台的兼容性。

Rust 语言在与 WebAssembly 的结合方面有很多优势。Rust 从诞生之初就考虑到与 C 语言的兼容性，使得使用 Rust 编写的代码可以直接与 C 语言的二进制接口 (ABI) 进行交互。这使得 Rust 成为编写 WebAssembly 模块的理想语言之一。Rust 的 Cargo 包管理工具也为构建、打包和发布 WebAssembly 模块提供了便利。通过使用 Rust 和 WebAssembly，开发人员可以借助 Rust 语言的安全性和性能优势，同时利用 WebAssembly 的跨平台和高性能特性，构建出功能强大、高效且安全的 Web 应用程序、浏览器扩展、游戏和其他 Web 平台的应用。

Rust 环境安装

安装 Rust 开发环境

访问 Rust 语言官方网站的安装地址 <https://www.rust-lang.org/tools/install>

先下载 rustup 安装程序再安装，安装的所有工具在 `~/.cargo/bin` 目录下，我们需要将该目录添加到 PATH 环境变量。

whl963854

Rust Visual C++ prerequisites

Rust requires a linker and Windows API libraries but they don't seem to be available.

These components can be acquired through a Visual Studio installer.

- 1) Quick install via the Visual Studio Community installer (free for individuals, academic uses, and open source).
- 2) Manually install the prerequisites (for enterprise and advanced users).
- 3) Don't install the prerequisites (if you're targeting the GNU ABI).

打开安装程序后，我们会发现需要依赖的 windowsAPI 库，我们采用下载 vs installer 的方式获取对应库。

Current installation options:

```
default host triple: x86_64-pc-windows-msvc
default toolchain: stable (default)
profile: default
modify PATH variable: yes
```

- 1) Proceed with installation (default)
 - 2) Customize installation
 - 3) Cancel installation
- >1

安装 vs install 以后重新打开 rustup 安装程序，选择默认安装。

安装完成后，我们需要将 `~/.cargo/bin` 目录添加到 PATH 环境变量。

我们可以通过以下方式查看 rustup、rustc 和 cargo 工具的版本信息：

```
C:\Users\wps>rustup --version
rustup 1.26.0 (5af9b9484 2023-04-05)
info: This is the version for the rustup toolchain manager, not the rustc compiler.
info: The currently active `rustc` version is `rustc 1.69.0 (84c898d65 2023-04-16)`

C:\Users\wps>rustc --version
rustc 1.69.0 (84c898d65 2023-04-16)

C:\Users\wps>cargo --version
cargo 1.69.0 (6e9a83356 2023-04-12)
```

其中，rustup 是 Rust 工具的管理工具，rustc 是 Rust 程序编译器，cargo 是 Rust 工程的管理工具

Rust 程序测试

我们从“你好，世界”这个例子开始测试。



The screenshot shows a code editor with a file named `test.rs` and an executable named `test.exe`. The code in `test.rs` is as follows:

```
1 fn main() {  
2     println!("你好，世界");  
3 }
```

Below the code editor, the terminal output is shown. It indicates that the command `cd "d:\workplace\rust\" && rustc test.rs && "d:\workplace\rust\test` was executed, and the output is "你好，世界".

其中，fn 是关键字，表示定义一个函数，定义的函数的名字是 main。main() 函数的参数在小括弧中列出（这里的 main() 函数没有参数），函数体位于大括弧内。这里 main() 函数体中只有一个语句，就是用 println! 宏输出一个字符串并换行。

Cargo 管理工程

Rust 编程语言引以为傲的一个重要特性是其工程管理工具 Cargo。Cargo 是 Rust 官方提供的构建系统和包管理器，被广泛认可为行业标杆。

以下是 Cargo 的一些主要特性和优势：

- 依赖管理：Cargo 提供了强大的依赖管理功能。通过 Cargo.toml 文件，可以定义项目的依赖项和版本约束，能够轻松地引入、更新和管理外部库。
- 构建系统：Cargo 提供了一个集成的构建系统，使得构建、编译和测试 Rust 项目变得简单而高效。它自动处理依赖关系、编译顺序和编译标志等，开发者只需要专注于编写代码而不必手动管理构建过程。
- 项目管理：Cargo 提供了一组命令行工具，用于创建、初始化和管理工作 Rust 项目。通过简单的命令，您可以创建新项目、生成文档、运行测试、发布软件包等，提高开发者的效率和 workflows。

- 社区集成：Cargo 和 Rust 社区紧密结合，通过 Cargo 可以方便地共享、发布和发现 Rust 库和工具。Cargo 提供了 Cargo Crates 网站 <https://crates.io> Rust 生态系统中集中管理和发布库的中央仓库。
- 多工作区支持：Cargo 允许在单个项目中管理多个工作区，每个工作区可以拥有自己的依赖关系和构建配置。这对于大型项目或拥有多个模块的项目特别有用，可以更好地组织和管理代码。

工程一般以目录的方式组织，因此我们先创建一个空的目录（目录的名字自由选择），其中包含一个 Cargo.toml 文件。

Plain Text

```
1 [package]
2 name = "hello"
3 version = "0.1.0"
```

Cargo.toml 是一种 TOML 格式的工程文件（TOML 格式和 ini 格式类似，但是其功能更加强大）。其中，[package] 部分包含工程的基本信息：name 字段表示工程的名字，version 字段表示工程的版本。

然后，创建一个 src 目录，在目录中创建一个 main.rs 文件：

Rust

```
1 fn main() {
2     println!("你好，世界");
3 }
```

Cargo 工具默认以 src/main.rs 为程序的入口文件，因此只需要输入 cargo run 就可以编译并运行程序了

```
PS D:\workplace\rust\hello> cargo run
Compiling hello v0.1.0 (D:\workplace\rust\hello)
Finished dev [unoptimized + debuginfo] target(s) in 0.53s
Running `target\debug\hello.exe`
你好，世界
PS D:\workplace\rust\hello> |
```

whl963854

Cargo 底层依然是调用 rustc 编译器工具。但是，Cargo 不仅可以管理可执行程序 and 库，还可以对其他第三方库的依赖进行管理，同时支持自定义的构建脚本。

`cargo-generate` 是一个开发人员工具，通过利用预先存在的 git 存储库作为模板，帮助开发者快速启动和运行新的 Rust 项目。

Shell

```
1 cargo install cargo-generate
```

本地文档

Rust 还提供了本地文档用于帮助学习和了解 Rust。

我们可以在终端运行：

Shell

```
1 rustup doc
```

打开文档。

whl963854

Rust Documentation

Welcome to an overview of the documentation provided by the [Rust project](#)[🔗]. This page contains links to various helpful references, most of which are available offline (if opened with `rustup doc`). Many of these resources take the form of “books”; we collectively call these “The Rust Bookshelf.” Some are large, some are small.

All of these books are managed by the Rust Organization, but other unofficial documentation resources are included here as well!

If you’re just looking for the standard library reference, here it is: [Rust API documentation](#)

Learning Rust

If you’d like to learn Rust, this is the section for you! All of these resources assume that you have programmed before, but not in any specific language:

The Rust Programming Language

Affectionately nicknamed “the book,” [The Rust Programming Language](#) will give you an overview of the language from first principles. You’ll build a few projects along the way, and by the end, you’ll have a solid grasp of how to use the language.

Rust By Example

If reading multiple hundreds of pages about a language isn’t your style, then [Rust By Example](#) has you covered. RBE shows off a bunch of code without using a lot of words. It also includes exercises!

安装 WebAssembly 开发环境

Rust 是目前 WebAssembly 生态中支持力度最强的一种语言。开源社区中不仅有 Rust 语言开发的 WebAssembly 虚拟机，还有基于 WebAssembly 模块的管理工具。

Rust 语言默认安装的是生成本地应用的开发环境，因此 WebAssembly 开发环境需要单独安装。

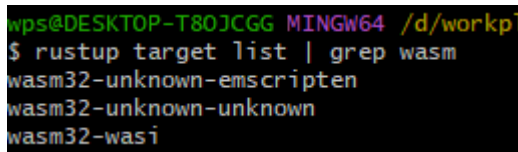
首先查看有哪些环境可以安装：

whl963854

Shell

```
1 rustup target list | grep wasm
```

运行结果如下所示:



```
wps@DESKTOP-T80JCGG MINGW64 /d/workp
$ rustup target list | grep wasm
wasm32-unknown-emscrip
wasm32-unknown-unknown
wasm32-wasi
```

- `rustup target add wasm32-wasi`:
 - 这个指令将把 `wasm32-wasi` 目标加入到 Rust 工具链中。`wasm32-wasi` 是用于 WebAssembly 系统接口(WASI)的目标, WASI 是一种与操作系统无关的 WebAssembly 运行环境, 允许在不同平台上运行 WebAssembly 模块。
- `rustup target add wasm32-unknown-emscrip`:
 - 为了将 `wasm32-unknown-emscrip` 目标添加到 Rust 工具链, 需要运行以下命令。`wasm32-unknown-emscrip` 是 Emscripten 工具链提供的一个目标, 它能够将 Rust 代码编译为适用于浏览器环境的 WebAssembly 模块。
- `rustup target add wasm32-unknown-unknown`:
 - 此命令将在 Rust 工具链中添加 `wasm32-unknown-unknown` 目标。`wasm32-unknown-unknown` 目标是一个通用的 WebAssembly 目标, 适用于在不特定于操作系统或运行时环境的情况下构建 WebAssembly 模块。

了解环境之后, 我们可以通过以下命令安装:

Shell

```
1 rustup target add wasm32-wasi
2 rustup target add wasm32-unknown-emscrip
3 rustup target add wasm32-unknown-unknown
```

为了方便测试和运行 WebAssembly 模块, 还需要安装 wasmer 虚拟机环境。使用 Scoop:

whl963854

Shell

```
1 scoop install wasmer
```

安装成功之后，输入以下命令查看 wasmer 版本信息：

Shell

```
1 wasmer -h
```

将 Rust 编译为 wasm

我们仍然使用刚才的入门测试程序，打印你好，世界！

然后，在编译时指定目标为 wasm32-wasi：

Shell

```
1 cargo build --target=wasm32-wasi
```

在 src 同级目录下面，target/wasm32-wasi/debug/ 内已经生成了对应的 hello.wasm 文件。

最后，通过 wasmer 工具运行该程序文件

Shell

```
1 wasmer run target/wasm32-wasi/debug/hello.wasm
```

可以看到程序正常输出

```
PS D:\workplace\rust\hello> wasmer run target/wasm32-wasi/debug/hello.wasm  
你好，世界
```

导入和导出函数

whl963854

导出 main() 函数

当在 Rust 中构建 WebAssembly 模块时，导入和导出函数是实现与宿主环境进行交互的关键部分。通过导入函数，可以在 Rust 模块中调用宿主环境中提供的功能，而通过导出函数，可以将 Rust 函数暴露给宿主环境使用。

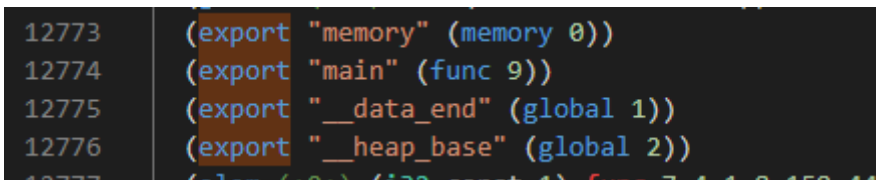
上面测试时编译目标为 `wasm32-wasi`，因此程序只能在宿主支持的 WASI 虚拟机环境运行。现在，我们设置编译目标为 `wasm32-unknown-unknown`（即纯 WebAssembly 环境），然后查看其导出的函数。

输入以下命令重新构建：

Shell

```
1 cargo build --target wasm32-unknown-unknown
```

默认生成 `./target/wasm32-unknown-unknown/debug/hello.wasm`

A terminal window with a dark background showing the output of a command. The output lists several exported functions and globals from a WebAssembly module. The first four lines are highlighted with a light blue background. The lines are: 12773 (export "memory" (memory 0)), 12774 (export "main" (func 9)), 12775 (export "__data_end" (global 1)), and 12776 (export "__heap_base" (global 2)). The fifth line is partially visible: 12777 (export "..." (func 3)).

```
12773 (export "memory" (memory 0))
12774 (export "main" (func 9))
12775 (export "__data_end" (global 1))
12776 (export "__heap_base" (global 2))
12777 (export "..." (func 3))
```

可以看到我们已经导出了 `main` 函数。

我们还可以在 Node.js 环境，通过 `console.dir()` 调试函数查看导出的内容，在 `wasm` 对应目录下创建 `hello.js` 文件：

JavaScript

```
1 const fs = require('fs');
2 //readFileSync文件读取会阻塞代码的执行，直到读取完成
3 const buf = fs.readFileSync('./hello.wasm');
4 WebAssembly.instantiate(new Uint8Array(buf)).then(function(result) {
5     console.dir(result.instance.exports);
6 });
```

运行代码，得到以下输出：

wh1963854

```
PS D:\workplace\rust\hello\target\wasm32-unknown-unknown\debug> node hello.js
[Object: null prototype] {
  memory: Memory [WebAssembly.Memory] {},
  main: [Function: 9],
  __data_end: Global [WebAssembly.Global] {},
  __heap_base: Global [WebAssembly.Global] {}
}
```

这说明普通的应用程序默认导出了 main() 函数。

导入宿主打印函数

WebAssembly 是一个与宿主环境隔离的虚拟机，因此 WebAssembly 模块不能直接访问宿主环境的控制台或文件系统等资源。为了在 WebAssembly 模块中输出信息，需要通过和宿主环境的交互来完成输出操作。

我们假定 Node.js 宿主环境提供了一个 console_log() 函数，用于输出一个整数值：

JavaScript

```
1 function console_log(x) {
2     console.log(x);
3 }
```

这样就可以通过 console_log() 函数输出信息了。src/main.rs 内容修改如下：

Rust

```
1 extern "C" {
2     fn console_log(a: i32);
3 }
4 fn main() {
5     unsafe {
6         console_log(42);
7     }
8 }
```

其中，紧跟 extern "C" 后的大括弧内是宿主导入函数声明，它们都是以 C 语言 ABI 规范导入。由于外部函数的实现是由宿主环境提供的，而不是由 Rust 代码控制的，因此我们无法保证这些

函数的安全性和正确性。为此，我们需要将调用外部函数的代码放在 `unsafe` 语句块中，这样编译器就会知道该代码可能存在不安全操作，并需要我们自行确保其正确性和安全性。

对应 Node.js 环境的启动代码如下

JavaScript

```
1 const fs = require('fs');
2 const buf = fs.readFileSync('./hello.wasm');
3
4 function console_log(x) { console.log(x); }
5 WebAssembly.instantiate(new Uint8Array(buf), {
6   env: { "console_log": console_log }
7 }).then(function(result) {
8   result.instance.exports.main();
9 });
```

在 `WebAssembly.instantiate` 实例化模块对象的时候，通过第二个参数传入了宿主导入对象，将宿主环境提供的 `console_log()` 函数作为导入对象传递给 `WebAssembly` 实例化函数。在导入对象的 `env` 属性中，包含的是对应 Rust 语言中 `extern "C"` 表示的函数列表，这里只有一个 `console_log()` 函数。在实例化完成之后，通过 `result.instance.exports.main()` 手工执行 `main()` 函数。

现在运行 `main()` 函数就可以得到正确的结果：

```
PS D:\workplace\rust\hello\target\wasm32-unknown-unknown\debug> node hello.js
42
PS D:\workplace\rust\hello\target\wasm32-unknown-unknown\debug> node hello.js
42
```

表明我们已经成功地将外部宿主导入的函数引入到 Rust 环境了。

导出自定义函数

导入宿主函数的真正目的是实现更复杂的功能。比如上面的例子中，`main()` 函数通过导入的 `console_log()` 函数实现了输出功能。对于可执行程序，作为入口的 `main()` 函数是默认导出的，那么如何导出一个自定义的函数呢？

首先，新建一个名为 `hello-double` 的 Cargo 工程，其 `src/main.rs` 内容如下

whl963854

Rust

```
1 fn main() {}  
2  
3 #[no_mangle]  
4 pub fn double(n: i32) -> i32 {  
5     n * 2  
6 }
```

由于不再需要 `main()` 函数作为默认入口函数，因此 `main()` 函数中没有任何代码（但是，这里 `main()` 函数本身依然需要保留）。本例的目的是导出一个名为 `double` 的函数，该函数的功能是将输入的整数翻倍后再返回。对于要导出的函数需要用 `pub` 关键字修饰。`#[no_mangle]` 注解告诉编译器不要修改导出函数名，使其导出后的函数名与 Rust 中的函数名保持一致。

`pub` 关键字用于将函数标记为公开可见，使其可以在其他模块或外部引用。

然后，在 Node.js 环境导出 `double()` 函数：

JavaScript

```
1 const fs = require('fs');  
2 const buf = fs.readFileSync('./hello-double.wasm');  
3 WebAssembly.instantiate(new Uint8Array(buf)).then(function(result) {  
4     const double = (i) => {  
5         return result.instance.exports.double(i);  
6     }  
7     for (let i = 0; i < 5; i++) {  
8         console.log(double(i));  
9     }  
10 });
```

运行结果如下图所示：

```
PS D:\workplace\rust\hello-double\target\wasm32-unknown-unknown\debug> node double.js  
0  
2  
4  
6  
8  
PS D:\workplace\rust\hello-double\target\wasm32-unknown-unknown\debug>
```

在 WebAssembly 中，函数的输入参数和返回值类型受到严格的限制。只有 `i32`（32 位整数）和 `f64`（64 位浮点数）这两种类型是安全的。如果需要处理字符串、结构体或其他复杂的数据类型，则需要通过内存对象进行数据交换。一种常见的方法是使用线性内存（Linear Memory），它是

WebAssembly 的内存模型之一。我们可以将数据存储在内存中，并通过导出的内存对象将数据传递给 WebAssembly 模块。

WebAssembly 处理复杂数据类型

上面我们已经通过导入和导出函数传递整数参数和返回整数结果。在 WebAssembly 中，字符串通常被表示为字节数组，而数组可以通过指针和长度进行传递。如果整数作为指针类型，再配合导出的内存对象，就可以交换字符串和数组等复杂数据结构。线性内存是一个连续的字节数组，用于在 Rust 和 WebAssembly 之间进行数据交换。线性内存由 Wasm 运行时分配和管理，并且可以通过导入和导出来在 Rust 和 WebAssembly 之间共享数据。在 Rust 中使用线性内存时，需要小心管理内存的正确性和安全性。确保在访问线性内存时遵循正确的索引和边界检查，以防止访问越界或悬垂指针等错误。

字符串处理

传递字符串参数

由于宿主向 Rust 传入字符串的过程比较复杂，我们先查看在 Rust 环境如何通过调用导入的宿主函数打印一个 Rust 字符串。

假设宿主导入了 `env_print_str()` 字符串打印函数，其声明如下：

Rust

```
1 extern "C" {  
2     fn env_print_str(s: *const u8, len: usize);  
3 }
```

在 Rust 中，指针类型 `*const u8` 表示一个不可变的字节指针，而 `usize` 类型表示无符号整数，用于表示长度或索引。该函数的第一个参数是字符串的开始内存地址，第二个参数是无符号整数类型表示的字符串字节长度。`main()` 函数可以使用 `env_print_str()` 打印 Rust 格式的字符串：

Rust

```
1 fn main() {  
2     let s = "你好, WebAssembly!";  
3     unsafe {  
4         env_print_str(s.as_ptr(), s.len());  
5     }  
6 }
```

调用前需要通过 `s.as_ptr()` 获取 Rust 字符串的内存地址，然后通过 `s.len()` 获取字符串的字节长度信息。由于涉及到使用裸指针和外部函数，这段代码被标记为 `unsafe`。在 `unsafe` 块中，我们需要自行确保操作的正确性和安全性，Rust 编译器无法对其进行静态检查。导入的 `env_print_str()` 函数是 `unsafe` 类型，因此必须放在 `unsafe` 代码块中调用。之后我们生成对应的 `.wasm` 文件。

准备就绪之后，通过以下命令生成 `.wasm` 文件：

Shell

```
1 $ cargo build --target wasm32-unknown-unknown
```

在 WebAssembly 中，字符串是一个复杂的数据类型，必须依赖 WebAssembly 实例的内存地址才能传递给宿主。WebAssembly 实例的内存地址只有在实例化之后才能获取。为了在 Rust 中将字符串传递给宿主环境，我们可以通过封装一些辅助的 JavaScript 类来实现。

下面介绍用于辅助的 `RustApp` 类：

wh1963854

JavaScript

```
1 class RustApp {
2     constructor() {
3         this._inst = null;
4     }
5     async Run(instance = null) {
6         if (instance !== null) { this._inst = instance; }
7         this._inst.exports.main();
8     }
9 }
```

- 构造函数：初始化 `RustApp` 对象，并将 `_inst` 属性设置为 `null`。
- `Run` 方法：用于运行 Rust WebAssembly 模块。如果传入了一个实例，则将该实例赋值给 `_inst` 属性；否则保持 `_inst` 属性的值不变。然后调用模块的 `main` 函数。

`RustApp` 类的使用方式如下：

JavaScript

```
1 const rustApp = new RustApp();
2 WebAssembly.instantiate(new Uint8Array(buf),
3     env: {
4         env_print_str: (s, len) => {
5             console.log(rustApp.GetString(s, len));
6         }
7     }
8 ).then((result) => {
9     return rustApp.Run(result.instance);
10 })
```

这样设计的目的是方便访问 WebAssembly 实例和相关的辅助方法。上述代码首先实例化一个 `RustApp` 对象，在传入的宿主导入对象中实现了 `env_print_str()` 函数的导入。`env_print_str()` 函数的参数是 Rust 字符串的内存地址和长度，内部首先通过 `rustApp.GetString(s, len)` 构建对应的 JavaScript 格式的字符串，然后通过 `console.log()` 输出字符串，最后在 WebAssembly 实例化成功之后通过 `rustApp.Run(result.instance)` 调用 `main()` 函数。

Rust 字符串的解码工作由 `GetString()` 方法完成：

wh1963854

JavaScript

```
1 class RustApp {
2     GetString(addr, len) {
3         return new TextDecoder("utf-8 ").decode(this.MemView(addr, len));
4     }
5     MemView(addr, len) {
6         return new DataView(this._inst.exports.memory.buffer,
7                               addr, len);
8     }
9 }
```

- **GetString** 方法用于将从 WebAssembly 实例中获取的字符串数据转换为 UTF-8 格式的字符串，
- **MemView** 方法用于获取在 WebAssembly 实例的内存中指定地址和长度范围的 DataView 对象。这样就可以在 Rust 程序中从导入的宿主 env_print_str 输出 Rust 字符串了。

为了从宿主环境向 Rust 传递字符串，我们可以使用 Rust 的内置 **String** 类型进行封装，并在宿主环境中导出函数来创建和删除这些字符串对象。

从宿主角度看，String 是一个 Rust 对象，因此首先需要导出对象创建和删除的函数：

whl963854

Rust

```
1 #[no_mangle]
2 pub fn string_new(size: usize) -> *mut String {
3     Box::into_raw(Box::new(String::from("\0").repeat(size)))
4 }
5
6 #[no_mangle]
7 pub fn string_delete(ptr: *mut String) {
8     if ptr.is_null() {
9         return;
10    }
11    unsafe {
12        Box::from_raw(ptr);
13    }
14 }
```

- `string_new` 是创建一个 `size` 字节大小的字符串，`Box::new` 来创建一个堆分配的 `String` 对象，其中包含由空字符（`\0`）重复 `size` 次组成的字符串。`Box::into_raw` 函数将 `Box<String>` 转换为原始指针，并返回该指针。
- `string_delete` 用于释放封装的字符串对象，内部将传入的 `String` 指针还原为 `Box` 引用计数对象，用 `Box::from_raw` 释放该指针所占用的内存。

有了 Rust 字符串对象之后，最重要的是获取底层的数据指针，这个工作由 `string_data_ptr()` 函数完成：

Rust

```
1 #[no_mangle]
2 pub fn string_data_ptr(ptr: *mut String) -> *mut u8 {
3     let me = unsafe {
4         assert!(!ptr.is_null());
5         &mut *ptr
6     };
7     return me.as_mut_ptr();
8 }
```

whl963854

- `string_data_ptr` 代码传入的是 `*mut String` 类型的字符串对象指针，内部先转换为 `&mut String` 类型，并将其绑定到变量 `me` 上。然后通过 `me.as_mut_ptr()` 获取底层的数据指针。

在得到底层的数据地址之后，就可以在 Node.js 环境向该地址写入数据了。

`RustString_new()`代码如下：

JavaScript

```
1 function RustString_new(rustApp, jsString) {
2     const bytes = new TextEncoder("utf-8").encode(jsString);
3     let rustString = rustApp.Exports().string_new(bytes.length);
4     let addr = rustApp.Exports().string_data_ptr(rustString);
5     rustApp.MemoryArray(addr, bytes.length).set(bytes);
6     return rustString;
7 }
```

- 将 JavaScript 字符串转换为 Rust 中的 `String` 对象，并通过原始指针的方式传递给 Rust 应用。第一个参数 `rustApp` 是 `RustApp` 类实例，第二个参数 `jsString` 是 JavaScript 字符串类型。内部首先通过 `TextEncoder()` 将 `jsString` 编码为 UTF8 字符串类型，然后通过 `rustApp.Exports().string_new` 调用导出的函数创建 Rust 字符串对象，接着通过 `rustApp.Exports().string_data_ptr` 函数获取字符串底层的数据指针，最后基于 Rust 环境的内存地址构建 `Uint8Array` 对象，并通过 `set` 方法写入 UTF8 字符串。

同时，我们还需要释放字符串，封装字符串释放函数：

JavaScript

```
1 function RustString_delete(rustApp, rustString) {
2     rustApp.Exports().string_delete(rustString);
3 }
```

我们给 `RustApp` 类增加了几个辅助方法：

whl963854

JavaScript

```
1 class RustApp {
2     Init(instance) {
3         this._inst = instance;
4     }
5     Exports() {
6         return this._inst.exports;
7     }
8     Mem() {
9         return this._inst.exports.memory;
10    }
11    MemUint8Array(addr, len) {
12        return new Uint8Array(this.Mem().buffer, addr, len);
13    }
14 }
```

- Init()方法用于在调用 Run()方法之前初始化内部的 WebAssembly 实例
- Exports 方法用于获取 WebAssembly 实例导出的各种对象（内存和函数等）
- MemUint8Array()方法则是基于指定内存地址和长度构建一个 Uint8Array 对象，这样宿主就可以读写 Rust 内存的任何数据。

现在宿主环境已经具备创建 Rust 字符串的能力。为了便于测试，我们再创建一个打印 Rust 字符串的函数：

Rust

```
1 #[no_mangle]
2 pub fn string_print(ptr: *mut String) {
3     let me = unsafe {
4         assert!(!ptr.is_null());
5         &mut *ptr
6     };
7     unsafe {
8         env_print_str(me.as_ptr(), me.len());
9     }
10 }
```

wh1963854

- 使用 `unsafe` 块将 `ptr` 转换为可变的 `&mut String` 引用，并将其赋值给 `me` 变量。首先将传入的字符串指针还原为字符串对象，然后通过宿主导入的 `env_print_str()` 函数打印。

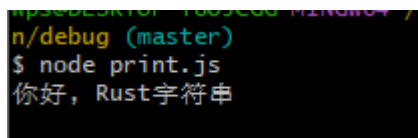
下面构建入口的 JavaScript 程序：

JavaScript

```
1 const fs = require('fs');
2 const buf = fs.readFileSync('./target/wasm32-unknown-unknown/debug/string.wasm');
3 const RustApp = require('./RustApp.js');
4
5 const theRustApp = new RustApp();
6 WebAssembly.instantiate(new Uint8Array(buf), {
7   env: {
8     env_print_str: (s, len) => {
9       console.log(theRustApp.GetString(s, len));
10     }
11   }
12 }).then((result) => {
13   theRustApp.Init(result.instance);
14   let rustString = RustString_new(theRustApp, "你好, Rust字符串 ");
15   theRustApp.Exports().string_print(rustString);
16 })
```

这个程序没有通过 `Run()` 调用 `main()` 函数，而是创建字符串之后直接通过 `string_print()` 打印。现在，我们就实现了 Rust 和宿主环境字符串类型双向传输，这也是打印命令行参数列表必要的前提工作。

运行结果如下：



```
mp3823101@1000000:~/n/debug (master)
$ node print.js
你好, Rust字符串
```

命令行参数封装

当使用 Rust 编写代码时，可以通过标准库的 `std::env::args()` 函数来获取和打印命令行参数：

whl963854

Rust

```
1 fn main() {  
2     println!("{:?}", std::env::args());  
3 }
```

可以通过 cargo run 传入命令行参数:

Shell

```
1 $ cargo run aa bb cc  
2 Args { inner: ["target/debug/demo", "aa", "bb", "cc"] }
```

运行结果如下:

```
PS D:\workplace\rust\command_line> cargo run aa bb cc  
Compiling command_line v0.1.0 (D:\workplace\rust\command_line)  
Finished dev [unoptimized + debuginfo] target(s) in 0.44s  
Running `target/debug/command_line.exe aa bb cc`  
Args { inner: ["target\\debug\\command_line.exe", "aa", "bb", "cc"] }
```

也可以通过 std::env::args().enumerate() 迭代的方式分别打印每个命令行参数:

Rust

```
1 fn main() {  
2     for (i, x) in std::env::args().enumerate() {  
3         println!("arg[{}]: {}", i, x);  
4     }  
5 }
```

运行的结果如下:

```
PS D:\workplace\rust\command_line> cargo run aa bb cc  
Compiling command_line v0.1.0 (D:\workplace\rust\command_line)  
Finished dev [unoptimized + debuginfo] target(s) in 0.47s  
Running `target/debug/command_line.exe aa bb cc`  
arg[0]: target/debug/command_line.exe  
arg[1]: aa  
arg[2]: bb  
arg[3]: cc
```

whl963854

前面我们已经实现了通过 WebAssembly 宿主函数打印 Rust 字符串的方法，并且实现了打印命令行参数。接下来我们需要考虑如何将外部的命令行参数传入 Rust 环境。

为此，我们构造一个全局的静态 `_ARGS` 变量，用于表示命令行参数对应的字符串列表：

Rust

```
1 static mut _ARGS: Vec<String> = Vec::new();
2
3 #[no_mangle]
4 pub fn ARGS_init(i: usize, s: *mut String) {
5     let me = unsafe {
6         assert!(!s.is_null());
7         &mut *s
8     };
9     unsafe {
10         if _ARGS.len() < (i+1) {
11             _ARGS.resize(i+1, String::from(""));
12         }
13         _ARGS[i] = me.clone();
14     }
15 }
```

同时，导出 `ARGS_init()` 函数，以便用于外部初始化 `_ARGS` 变量。`ARGS_init()` 函数的内部工作原理和封装 Rust 字符串对象类似，首先是从 `*mut String` 指针还原出字符串对象，然后将字符串记录到 `_ARGS` 变量。因为 Rust 环境默认禁止修改全局变量，因此需要在 `unsafe` 块中包含相关的代码。

为了方便在 Rust 内部访问命令行参数列表，需要再包装 `ARGS()` 函数：

Rust

```
1 fn ARGS() -> Vec<String> {
2     unsafe { _ARGS.clone() }
3 }
```

基于新的代码，我们可以通过以下方法打印命令行参数（初始化工作由宿主环境完成）：

whl963854

Rust

```
1 fn print_str(s: &str) {
2     unsafe {
3         env_print_str(s.as_ptr(), s.len());
4     }
5 }
6 fn main() {
7     for (i, x) in ARGS().iter().enumerate() {
8         print_str(format!("{}", i, x).as_str());
9     }
10 }
```

其中，print_str()函数是对导入 env_print_str()函数的封装，隐藏了 unsafe 块的特性。

传入命令行参数

为了便于使用，我们依然在外部宿主环境封装 Rust ARGS_init()函数：

Rust

```
1 function RustARGS_init(rustApp, ...args) {
2     args.forEach((v, i) => {
3         let rustString = RustString_new(rustApp, v);
4         rustApp.Exports().ARGS_init(i, rustString);
5         RustString_delete(rustApp, rustString);
6     })
7 }
```

封装的 RustARGS_init()函数的第一个参数 rustApp 依然是 RustApp 类实例，后面可选的参数是命令行参数字符串。封装函数首先遍历 args 参数，然后调用 rustApp.Exports().ARGS_init()导出函数将每个命令行参数字符串传入 Rust 环境。

然后构建入口的 JavaScript 程序：

whl963854

JavaScript

```
1 const theRustApp = new RustApp();
2 WebAssembly.instantiate(new Uint8Array(buf), {
3   env: {
4     env_print_str: (s, len) => {
5       console.log(theRustApp.GetString(s, len));
6     }
7   }
8 }).then((result) => {
9   theRustApp.Init(result.instance);
10  // 初始化 argv
11  RustARGS_init(theRustApp, ...process.argv);
12  // 运行main()函数
13  return theRustApp.Run();
14 })
```

上述代码中，首先通过 `theRustApp.Init(result.instance)` 初始化 `RustApp` 类的实例对象，然后调用 `RustARGS_init()` 方法将 `process.argv` 命令行参数传入 Rust 环境，最后通过 `theRustApp.Run()` 执行 Rust 环境的 `main()` 函数并打印字符串。至此，我们就实现了从宿主传入命令行参数，然后在 Rust 程序中遍历并打印命令行参数。

运行结果如下：

```
PS D:\workplace\rust\print\target\wasm32-unknown-unknown\debug> node print.js aa bb cc dd ee ff
0: C:\Program Files\nodejs\node.exe
1: D:\workplace\rust\print\target\wasm32-unknown-unknown\debug\print.js
2: aa
3: bb
4: cc
5: dd
6: ee
7: ff
```

no_std

通过启用 `no_std` 选项，可以在 Rust 代码中不依赖标准库（`std`）的情况下进行编译。这对于 WebAssembly 环境中的应用程序非常有用，可以大大减小生成的可执行文件的体积。启用 `no_std` 选项后，则需要手动选择和引入所需的库和功能，以替代标准库中的功能。通常

会使用 `core` 库，它提供了 Rust 语言的核心功能，如基本数据类型、操作符、切片和基本的内存操作等。此外，还可以使用 `alloc` 库来提供动态内存分配和释放的功能。

输出文件的大小

以最简单的“你好，WebAssembly!”例子为参考，实现代码如下：

Rust

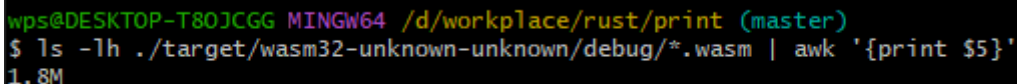
```
1 extern "C" {
2     fn env_print_str(s: *const u8, len: usize);
3 }
4 fn main() {
5     let s = "你好，WebAssembly!";
6     unsafe {
7         env_print_str(s.as_ptr(), s.len());
8     }
9 }
```

首先，采用默认的构建方式输出 Debug 版本的 .wasm 文件，该输出文件体积大约为 1.8 MB：

Shell

```
1 $ cargo build --target wasm32-unknown-unknown
2 $ ls -lh ./target/wasm32-unknown-unknown/debug/*.wasm | awk '{print
   $5}'
3 1.8M
```

运行结果如下：



```
wps@DESKTOP-T80JCGG MINGW64 /d/workplace/rust/print (master)
$ ls -lh ./target/wasm32-unknown-unknown/debug/*.wasm | awk '{print $5}'
1.8M
```

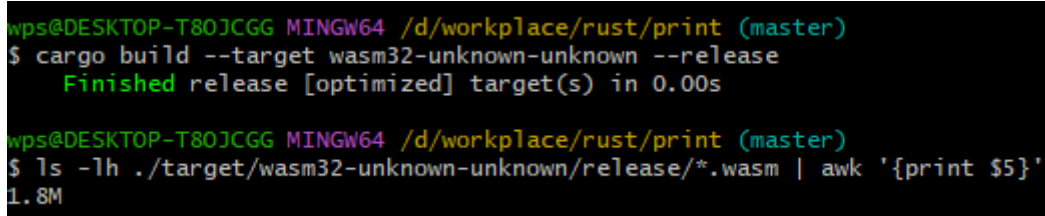
默认方式生成的 Release 版本的 .wasm 文件的体积大约也是 1.8MB：

whl963854

Shell

```
1 $ cargo build --target wasm32-unknown-unknown --release
2 $ ls -lh ./target/wasm32-unknown-unknown/release/*.wasm | awk '{print
   $5}'
3 1.8M
```

运行结果如下：



```
wps@DESKTOP-T80JCGG MINGW64 /d/workplace/rust/print (master)
$ cargo build --target wasm32-unknown-unknown --release
   Finished release [optimized] target(s) in 0.00s

wps@DESKTOP-T80JCGG MINGW64 /d/workplace/rust/print (master)
$ ls -lh ./target/wasm32-unknown-unknown/release/*.wasm | awk '{print $5}'
1.8M
```

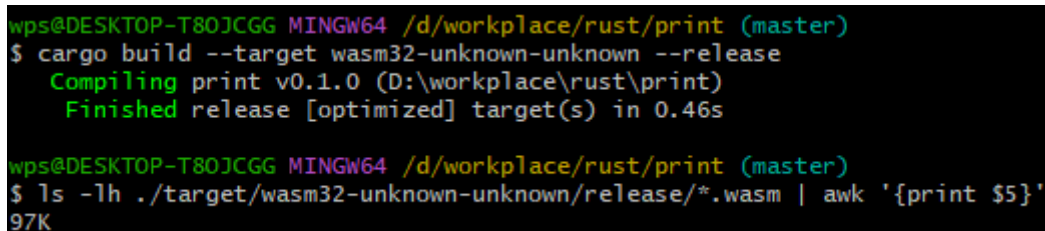
编译参数对输出 .wasm 文件的大小也有很大的影响。我们可以在 Cargo.toml 文件中增加以下内容：

Plain Text

```
1 [profile.release]
2 lto = true
3 opt-level = 's'
```

- [profile.release]表示针对 Release 版本的额外构建参数；
- lto 表示使用链接时优化进行编译，这样可以采用更多的内联优化，不仅可以提高运行速度，而且会减小输出 .wasm 文件的体积。
- optlevel 编译优化参数为's'，表示 LLVM 优先优化体积而不是速度。

以新的参数构建 Release 版本，可以生成大约 97K 大小的 .wasm 文件，如图所示：



```
wps@DESKTOP-T80JCGG MINGW64 /d/workplace/rust/print (master)
$ cargo build --target wasm32-unknown-unknown --release
   Compiling print v0.1.0 (D:\workplace\rust\print)
   Finished release [optimized] target(s) in 0.46s

wps@DESKTOP-T80JCGG MINGW64 /d/workplace/rust/print (master)
$ ls -lh ./target/wasm32-unknown-unknown/release/*.wasm | awk '{print $5}'
97K
```

whl963854

这是在不改变源代码的前提下完全由编译器和连接器取得的优化成果。如果希望进一步减小 .wasm 文件的大小，那么可以尝试去掉默认的标准库，通过启用 `no_std` 选项并手动选择所需的功能和库。

通过 no_std 关闭标准库

源文件中包含 `#![no_std]`，表示关闭标准库，以下是 `src/main.rs` 文件的内容：

Rust

```
1  #![no_std]
2  extern "C" {
3      fn env_print_str(s: *const u8, len: usize);
4  }
5  fn main() {
6      let s = "你好, WebAssembly!";
7      unsafe {
8          env_print_str(s.as_ptr(), s.len());
9      }
10 }
```

不幸的是，编译时出现了以下错误：

Shell

```
1 $ cargo build --target wasm32-unknown-unknown
```

错误提示：缺少 `#[panic_handler]` 指定的异常处理函数。

```
PS D:\workplace\rust\print> cargo build --target wasm32-unknown-unknown
Compiling print v0.1.0 (D:\workplace\rust\print)
error: `#[panic_handler]` function required, but not found
error: could not compile `print` due to previous error
PS D:\workplace\rust\print>
```

我们可以通过以下方式指定异常处理函数：

whl963854

Rust

```
1 #[panic_handler]
2 fn panic(_info: &core::panic::PanicInfo) -> ! {
3     loop {}
4 }
```

该异常处理模式虽然简单粗暴，但是没有其他额外的依赖，比较适合 no_std 这种场景使用。重新编译时出现了新的错误：

Shell

```
1 $ cargo build --target wasm32-unknown-unknown
```

运行结果如下：

```
PS D:\workplace\rust\print> cargo build --target wasm32-unknown-unknown
   Compiling print v0.1.0 (D:\workplace\rust\print)
error: requires `start` lang_item
error: could not compile `print` due to previous error
```

错误提示：构建环境要依赖于一个 start 语言特性，这是 WebAssembly 的一个特性（载入模块时自动执行 start 指令指定的函数）。该新特性需要通过 `#![feature(lang_items)]` 指令才能开启，并且不能用于稳定的 Rust 版本（未来可能转化为正式特性）。

总结

本文介绍了 Rust 语言开发 WebAssembly 模块的常用技术。随着 Rust 对 WebAssembly 平台支持逐渐成熟，Rust 应用已经成为 WebAssembly 生态的主力。Rust 语言安全的编程模型和 C/C++ 丰富的软件资源的结合，可以极大地加速 WebAssembly 软件的开发。

whl963854