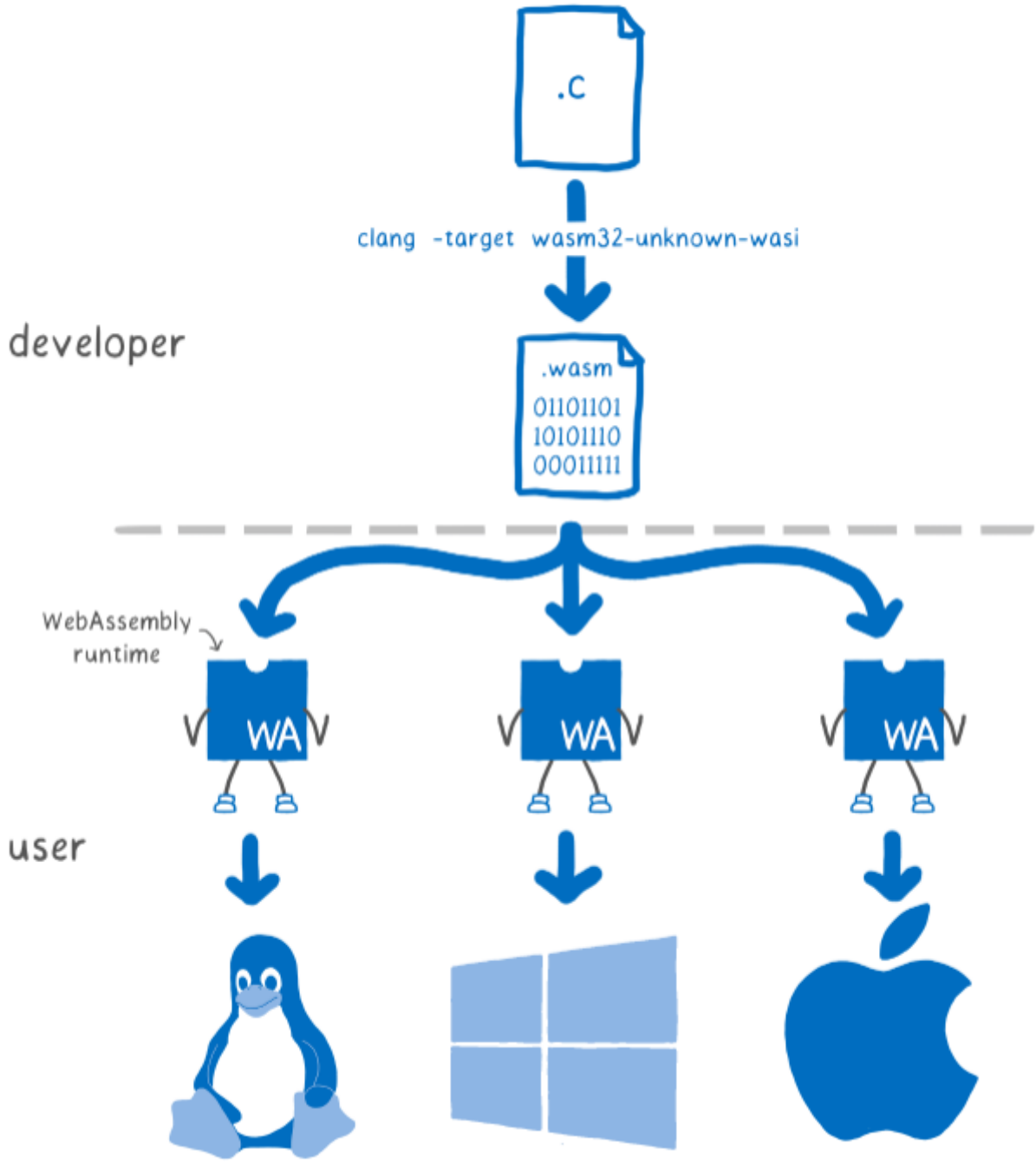
WebAssembly 与 Go 实践

wasm 简介

webassembly 的定义

WASM 是“WebAssembly”的缩写。WebAssembly 是一种新的代码格式，是一种低级别、代码密集型的与 JavaScript 类似的虚拟机，但它并不基于文本，而是一种二进制格式，无需编译器处理即可运行 WebAssembly 可以为那些不想或难以通过 JavaScript 实现的计算任务提供一种性能非常优秀的运行方式，其中包括数据密集型任务、游戏以及机器学习应用。



WebAssembly 支持多种编程语言，包含 C/C++、 Rust、 Go、 Swift、 Kotlin、 AssemblyScript、 Grain，甚至还有 JavaScript 和 Python。

- 对于编译型语言（比如 C 和 Rust）来说，WasmEdge WebAssembly 提供了一个相比于原生客户端（NaCl）更安全的、受保护的、隔离的并且容器化的运行时。
- 对于解释型语言或者是受控型语言（比如 JavaScript 和 Python）来说，WasmEdge WebAssembly 提供了一个比 Docker + 客人操作系统（Guest OS） + 原生解释器这种组合更安全、快速、轻量且容器化的运行时。

WebAssembly（WASM）的优势

- 高效性：WASM 可以实现高效的解释和编译，因此可以快速执行计算密集型任务，提高 Web 应用程序的性能。与 JavaScript 相比，它更加接近计算机本身的机器语言，因此可以在不增加时间和内存开销的情况下加快许多任务的执行速度。
- 可移植性：WASM 是一种与平台无关的格式，可以在任何浏览器和操作系统上运行。WASM 应用程序不需要为每个平台编写代码或重新编译代码，从而大大降低了开发和维护成本。
- 安全性：WASM 代码在运行之前需要通过一系列的检查程序，以确保其不会访问到不安全的资源或内存，并且其执行不会导致不安全的行为。因此，WASM 在安全性方面有着很大的优势。
- 可组合性： WebAssembly 可以与 JavaScript 代码无缝协作，可以简化现有技术栈之间的迁移和相互支持，可以将现有的 JavaScript 库包装成 WebAssembly 模块，以更快、更高效地执行计算密集型任务。
- 扩展性：WASM 支持多种编程语言，如 C++、Rust 和 Go 等，这可以帮助开发者来扩展现有的技术栈并更快地将不同的代码整合到一个应用程序中。

什么是 wasi

wasi 官网：https://wasi.dev/

WASI 全称 The WebAssembly System Interface 通俗来讲就是一个对接规范，将 wasm 的应用领域从 web 中拓展到更广阔的各个平台中去。

WASI 允许 WebAssembly 环境访问操作系统提供的功能例如文件，文件系统，网络套接字，计时器和随机数生成器。与 WebAssembly 在导入功能级别限制访问的方式类似，WASI 控制对系统功能的访问。

环境部署

安装 Tinygo

TinyGo 支持一小部分的 Go 语言标准库和 Go 语言的大部分语法和特性，相对于 Go 本身编译具有轻量的特点，用 TinyGo 编译可以大大缩小编译后程序大小。

安装过程参照官方文档 https://tinygo.org/getting-started/overview/ 有两种安装方式

1. 通过 Scoop 快速安装

这种安装方式 $PATH 环境变量将通过 scoop 包更新。默认情况下，在 ~/scoop/shims/tinygo.

```
Shell

1 > scoop install tinygo
```

可以通过 scoop 升级到最新的 TinyGo 版本：

```
Shell

1 > scoop update tinygo
```

可以通过运行应显示版本号的命令来测试安装是否成功：

```
Shell

1 > tinygo version
2 tinygo version 0.27.0 windows/amd64 (using go version go1.20 and LLVM v
  ersion 15.0.0)
```

2. 手动安装

从 https://github.com/tinygo-org/tinygo/releases/download/v0.27.0/tinygo0.27.0.windows-amd64.zip 下载适用于 Windows 的 TinyGo 二进制文件

然后解压文件，文件夹解压完成后，需要添加 your-path\tinygo\bin 到 PATH 当中。

```
Shell

1 > set PATH=%PATH%;"your-path\tinygo\bin";
```

python 环境

wasm-opt 工具用于优化 WebAssembly 二进制文件，通常随 Emscripten SDK 一起提供。Emscripten SDK 安装过程中需要使用 Python 解释器来执行一些操作。

https://www.python.org/downloads/windows/ 官网下载相应版本 installer

安装过程中记得勾选将 python 加入到环境变量选项，否则需要手动加入。

Emscripten SDK

Emscripten 是一种基于 LLVM 的编译器，理论上能够将任何能够生成 LLVM 位码的代码编译成 javascript 的严格子集 asm.js，

从源码编译安装十分麻烦，推荐安装核心的 Emscripten SDK。以 Windows 为例，先使用如下命令下载 emsdk。

Shell

```
# Get the emsdk repo
git clone https://github.com/juj/emsdk.git

# Enter that directory
cd emsdk
```

再使用如下命令安装配置 Emscripten。

Go

```
# Fetch the latest registry of available tools.
git pull

# Download and install the latest SDK tools. Need install Python first.
./emsdk.bat install latest

# Make the "latest" SDK "active" for the current user. (writes ~/.emscripten file)
./emsdk.bat activate latest

# Activate PATH and other environment variables in the current terminal
./emsdk_env.bat
```

完成上述步骤以后，需要将 upstream 内的 bin 目录加入到环境变量，或者手动设置 WASMOPT 环境变量。

Shell

```
export WASMOPT=path/to/wasm-opt
```

注意，path/to/wasm-opt 应替换为实际的 wasm-opt 可执行文件路径。

安装 wasm 的运行时环境

浏览器

运行 wasm 需要有一个运行时环境，Firefox Quantum、Safari、Edge 和 Chrome 等主流浏览器都支持 WebAssembly，内置了 wasm 的 runtime，因此大部分浏览器内可以直接运行 wasm。而在非浏览器环境，则需要有一个 runtime 来执行 wasm。

wazero

wazero 是一种使用 Go 编写的零依赖 wasm 运行时环境。它允许 Go 开发人员在应用程序中轻松执行 wasm 代码，无需进行额外的依赖。

https://tetrate.io/blog/introducing-wazero-from-tetrate/#h-a-short-history-of-running-webassembly-in-go

点击上述链接自动下载 windows 安装版本，默认安装到 C:\Program Files\wazero, 将对应可执行文件加入到环境变量以便可以在其他文件夹运行该文件。

将 Go 编译为 wasm

将 Golang 程序编译为 wasm 格式，一般有两种方式：

1. 使用第三方编译工具，例如 TinyGo 编译器进行编译。
2. 采用 Go 原生编译器(Go1.11 及以后)进行编译：无需安装第三方编译库，但此种编译的 wasm 不支持 wasi 规范；在 Go1.21 起，Go 将会支持 WASI 的特性。预计先支持 WASI Preview1 标准，WASI Preview2 目前距离标准成熟还为时尚早，后续 Preview2 标准成熟后会继续支持新标准

TinyGo 编译

创建一个 hello 文件

Go

```go
package main
import "fmt"
func main() {
    fmt.Println("Hello world!")
}
```

构建命令

Shell

```shell
tinygo build -o hello.wasm -target=wasi hello.go
```

- target 有两种目标，一种是 wasm，主要是运行在 web 浏览器环境；一种是 wasi，即非 web 环境下运行。
- -o 参数指定输出的文件路径及名称，-target 参数指定编译格式，最后 main.go 代表需要编译的文件入口。

运行文件

```Shell
wazero run hello.wasm
```

得到输出

```Plain Text
    Hello world!
```

Go 编译

官方 wiki 指南：https://github.com/golang/go/wiki/WebAssembly，Go 编译器官方从 Go1.11 开始支持将代码编译为 wasm 模块。

将编写好的 go 程序编译成 wasm 格式，在编译时需要指定编译参数：

```Plain Text
GOOS=js GOARCH=wasm go build -o ./main.wasm
```

获取 go 官方准备的 js 胶水文件：

```Plain Text
cp "$(go env GOROOT)/misc/wasm/wasm_exec.js" .
```

PS: JS 胶水文件（JavaScript glue file）是指一类用于连接 JavaScript 代码与其他语言的代码的文件。

常见的使用 JS 胶水文件的场景包括使用 JavaScript 调用 Go 程序，使用 JavaScript 调用 C/C++ 程序，使用 JavaScript 调用 Python 程序等等。JS 胶水文件通常会包含一些 Go，

C/C++ 或 Python 的函数定义，以及这些函数的映射关系。

在 JavaScript 代码中调用这些函数时，JS 胶水文件会将函数调用转换为对应的 Go，C/C++ 或 Python 函数调用。

JS 胶水文件的实现方式有多种，比较常见的方式包括使用 Node.js 的 ffi 模块、使用 Emscripten 编译 Go， C/C++ 代码为 WebAssembly 等。

创建一个 html 文件，引入 wasm：

XML/HTML

```
<html>
<head>
    <meta charset="utf-8"/>
    <script src="wasm_exec.js"></script>
    <script>
        const go = new Go();
        WebAssembly.instantiateStreaming(fetch("hello.wasm"), go.importObject).then((result) => {
            go.run(result.instance);
        });
    </script>
</head>
<body>
    <h1>WebAssembly Demo</h1>
</body>
</html>

```

当我们运行程序的时候，我们不能直接在浏览器中打开 HTML 文件，因为跨域请求是不支持 `file` 协议的。我们需要将我们的输出文件运行在 HTTP 协议上。Web 服务器启动 index.html、wasm_exec.js 和 main.wasm，例如，goexec：

Shell

```
#install goexec:
go get -u github.com/shurcooL/goexec

goexec 'http.ListenAndServe(`:8080`, http.FileServer(http.Dir(`.`)))'
```

这个命令将在当前目录启动一个 Go 的 HTTP 服务器，用于提供静态文件。在终端中运行该命令后，可以通过 http://localhost:8080/ 访问网页，其中 . 是当前目录的路径。如果网页文件是 index.html，那么可以通过 http://localhost:8080/index.html 访问它。

导航到 http://localhost:8080/index.html，打开 JavaScript 调试控制台，会看到输出。可以修改程序、重建 main.wasm 和刷新来查看新的输出。

注意：要使 goexec 命令在类 Unix 系统上运行，则必须将 Go 的路径环境变量添加到 shell 的 profile。

Go 编译 wasi（实验）

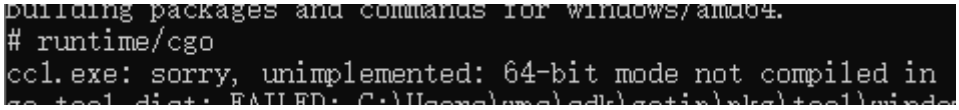
在 Go1.21 起，Go 将会支持 WASI 的特性。预计先支持 WASI Preview1 [1] 标准，后续 WASI Preview2 成熟后会继续支持新标准。

目前最新版本是 1.20.3，我们可以通过安装 gotip 体验最新版本 Go，gotip 是从开发分支上编译并运行 go 的命令。安装命令如下：

Shell

```
go install golang.org/dl/gotip@latest
gotip download
```

注意，需要将本地的 cc1.exe 改为 64 位的。



下载地址 https://sourceforge.net/projects/mingw-w64/files/mingw-w64/mingw-w64-snapshot/

下载完成后，我们将对应路径加入到环境变量（cc1 的路径也需要加入）

我们写一个简单的 Go Wasm Demo 测试一下：

Go

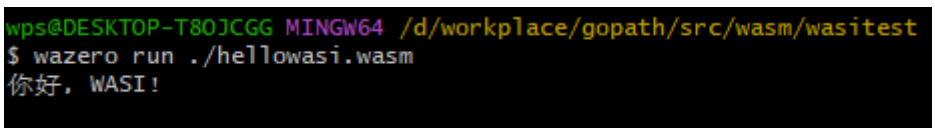
```go
package main

func main() {
    println("你好，WASI！")
}
```

接下来我们将上述 Go 程序编译为 .wasm 文件。使用如下编译命令：

Shell

```shell
GOARCH=wasm GOOS=wasip1 gotip build -o hellowasi.wasm hellowasi.go
```

通过 wazero 运行生成的文件



wasm 和 go 交互

Go 调用 js 代码

新建文件 main.go，使用 js.Global().get（‘alert’）获取全局的 alert 对象，通过 Invoke 方法调用。等价于在 js 中调用 `window.alert("Hello World")` 。

Go

```
package main

import"syscall/js"

funcmain() {
    alert := js.Global().Get("alert")
    alert.Invoke("Hello World!")
}
```

然后参照上面步骤编译 main 程序的方法生成对应 wasm 文件，复制胶水文件。

创建 index.html，引用 `main.wasm` 和 `wasm_exec.js` 。

Go

```
<html>
<scriptsrc="wasm_exec.js"></script>
<script>
    const go = new Go();
    WebAssembly.instantiateStreaming(fetch("main.wasm"), go.importObject)
        .then((result) => go.run(result.instance));
</script>

</html>
```

使用 goexec 启动 Web 服务

浏览器访问 localhost:8080，则会有一个弹出窗口，上面写着 *Hello World!*。

js 调用 go 函数

JavaScript 代码调用其他语言生成的 wasm，需要使用注册函数

假设我们需要注册一个计算斐波那契数列的函数，通过 syscall/js 包提供的 API，将函数 fib 注册为 JavaScript 函数，并通过 js.Global().Set() 将其绑定到全局对象上。然后，在 JavaScript 中就可以通过调用 fibFunc() 函数来调用 Go 语言中的 fib 函数，并获得返回值。

Go

```go
// main.go
package main

import "syscall/js"

func fib(i int) int {
    if i == 0 || i == 1 {
        return 1
    }
    return fib(i-1) + fib(i-2)
}

func fibFunc(this js.Value, args []js.Value) interface{} {
    return js.ValueOf(fib(args[0].Int()))
}

func main() {
    done := make(chan int, 0)
    js.Global().Set("fibFunc", js.FuncOf(fibFunc))
    <-done
}
```

- 定义了 fibFunc 函数，为 fib 函数套了一个壳，从 args[0] 获取入参，计算结果用 js.ValueOf 包装，并返回。
- this 和 args。其中 this 表示函数的执行上下文，对于该函数来说，它没有被使用到，因此被省略了。而 args 是一个包含传递给函数的所有参数的数组。
- 使用 js.Global().Set() 方法，将注册函数 fibFunc 到全局，以便在浏览器中能够调用。
- js.ValueOf() 方法是 syscall/js 包提供的一个函数，它用于将 Go 的值转换为对应的 JavaScript 值，并返回一个 js.Value 类型的值。

fibFunc 如果在 JavaScript 中被调用，会开启一个新的子协程执行。使用 Go 的并发机制来处理复杂的任务，而不会阻塞 JavaScript 的事件循环。

接下来创建 index.html 在其中添加一个输入框(num)，一个按钮(btn) 和一个文本框(ans，用来显示计算结果)，并给按钮添加了一个点击事件，调用 fibFunc，并将计算结果显示在文本框(ans)中。

Go

```
<html>
...
<body>
    <input id="num" type="number" />
    <button id="btn" onclick="ans.innerHTML=fibFunc(num.value * 1)">Click</button>
    <p id="ans">1</p>
</body>
</html>
```

使用之前的命令重新编译 main.go，并启动 Web 服务，运行效果如下：

注：go 编写的函数如何暴露给 js 调用　https://pkg.go.dev/syscall/js

Go 实现 DOM 元素操作

在上面例子中，Go 仅仅注册了全局函数 fibFunc，事件注册，调用，对 DOM 元素的操作都是在 HTML 中通过 js 函数实现的。

首先修改 index.html，删除事件注册部分和 对 DOM 元素的操作部分。

XML/HTML

```
<html>
...
<body>
    <inputid="num"type="number" />
    <buttonid="btn">Click</button>
    <pid="ans">1</p>
</body>
</html>
```

修改 main.go：

Go

```go
package main

import (
    "strconv"
    "syscall/js"
)

func fib(i int) int {
    if i == 0 || i == 1 {
        return 1
    }
    return fib(i-1) + fib(i-2)
}

var (
    document = js.Global().Get("document")
    numEle   = document.Call("getElementById", "num")
    ansEle   = document.Call("getElementById", "ans")
    btnEle   = js.Global().Get("btn")
)

func fibFunc(this js.Value, args []js.Value) interface{} {
    v := numEle.Get("value")
    if num, err := strconv.Atoi(v.String()); err == nil {
        ansEle.Set("innerHTML", js.ValueOf(fib(num)))
    }
    return nil
}

func main() {
    done := make(chan int, 0)
    btnEle.Call("addEventListener", "click", js.FuncOf(fibFunc))
    <-done
}
```

上述代码在 go 中获取 js 元素对象对其进行操作

- 通过 `js.Global().Get("btn")` 或 `document.Call("getElementById", "num")` 两种方式获取到 DOM 元素。
- btnEle 调用 `addEventListener` 为 btn 绑定点击事件 fibFunc。
- 在 fibFunc 中使用 `numEle.Get("value")` 获取到 numEle 的值（字符串），转为整型并调用 fib 计算出结果。
- ansEle 调用 `Set("innerHTML", ...)` 渲染计算结果。

回调函数

假设 fib 的计算非常耗时，那么可以启动注册一个回调函数，待 fib 计算完成后，再把计算结果显示出来。

先修改 main.go，使得 fibFunc 支持传入回调函数。

Go

```go
package main

import (
    "syscall/js"
    "time"
)

func fib(i int) int {
    if i == 0 || i == 1 {
        return 1
    }
    return fib(i-1) + fib(i-2)
}

func fibFunc(this js.Value, args []js.Value) interface{} {
    callback := args[len(args)-1]
    go func() {
        time.Sleep(3 * time.Second)
        v := fib(args[0].Int())
        callback.Invoke(v)
    }()

    js.Global().Get("ans").Set("innerHTML", "Waiting 3s...")
    return nil
}

func main() {
    done := make(chan int, 0)
    js.Global().Set("fibFunc", js.FuncOf(fibFunc))
    <-done
}
```

- 使用 `go func()` 启动子协程，调用 fib 计算结果，计算结束后，调用回调函数 `callback`，并将计算结果传递给回调函数，使用 time.Sleep() 模拟 3s 的耗时操作。

接下来我们修改 index.html，为按钮添加点击事件，调用 fibFunc

XML/HTML

```
<html>
...
<body>
    <input id="num" type="number" />
    <button id="btn" onclick="fibFunc(num.value * 1, (v)=> ans.innerHTML=v)">Click</button>
    <p id="ans"></p>
</body>
</html>
```

- 为 btn 注册了点击事件，第一个参数是待计算的数字，从 num 输入框获取。
- 第二个参数是一个回调函数，将参数 v 显示在 ans 文本框中。

WebAssembly 线性内存

概述

线性内存是一个连续的、字节寻址的内存空间，由一系列的页组成。每一页的大小为 64KB，可以通过 WebAssembly 模块的内存限制指令来设置线性内存的最小页数和最大页数。

线性内存的寻址方式是通过一个内存索引值和一个偏移量来定位内存中的某个特定字节。可以使用 WebAssembly 指令来读取和写入线性内存中的字节。此外，WebAssembly 还提供了一些内存相关的指令，比如内存复制、内存填充、内存大小查询等。

线性内存是 WebAssembly 中实现高效内存访问的重要机制之一，它不仅可以被 WebAssembly 模块本身访问，还可以被与 WebAssembly 交互的宿主环境（如 JavaScript）访问。通过将数据从宿主环境复制到 WebAssembly 的线性内存中，可以使得 WebAssembly 代码能够对这些数据进行高效的处理，从而实现更高效的跨语言交互。

需要注意 WebAssembly 的线性内存大小是固定的，因此在 WebAssembly 模块实例化之后，无法再次改变它的大小。如果需要动态地增加或减少内存，可以通过重新实例化一个新的

WebAssembly 模块来实现，但这会涉及到复制原有的线性内存内容，会比较消耗性能。

示例

首先我们编写，main.go

Go

```go
package main

// 创建一个字节（uint8，而不是Go字节）缓冲区，该缓冲区将在Wasm Memory中可用。
// 然后我们可以与JS和Wasm共享这个缓冲区。
const BUFFER_SIZE int = 2;
var buffer [BUFFER_SIZE]uint8;

func main() {}

// 函数返回指向我们在wasm内存中的缓冲区的指针（索引）
// export getWasmMemoryBufferPointer
func getWasmMemoryBufferPointer() *[BUFFER_SIZE]uint8 {
    return &buffer
}

// 函数将传递的值存储在索引0处，
// 在我们的缓冲区中
// go:export storeValueInWasmMemoryBufferIndexZero
func storeValueInWasmMemoryBufferIndexZero(value uint8) {
    buffer[0] = value
}

// 函数从我们的缓冲区的索引1处读取
// 并返回该索引处的值
// go:export readWasmMemoryBufferAndReturnIndexOne
func readWasmMemoryBufferAndReturnIndexOne() uint8 {
    return buffer[1]
}

```

然后将其用 TinyGo 编译成 wasm 模块

之后我们编写 index.js 文件

JavaScript

```javascript
// 在 wasm_exec.js 中定义。别忘了在 index.html 中添加这个脚本。
const go = new Go();

const runWasm = async () => {
    const importObject = go.importObject;

    // 实例化 wasm 模块
    const wasmModule = await wasmBrowserInstantiate("./main.wasm", importObject);

    // 在进行任何其他操作之前，必须先运行 go 实例，否则 println 等操作将不起作用
    go.run(wasmModule.instance);

    /**
    * 第一部分：在 Wasm 中写入，在 JS 中读取
    */
    console.log("在 Wasm 中写入，在 JS 中读取，索引为 0：");
    // 首先，让 wasm 写入我们的缓冲区
    wasmModule.instance.exports.storeValueInWasmMemoryBufferIndexZero(24);

    // 接下来，让我们创建一个指向我们的 wasm 内存的 Uint8Array
    let wasmMemory = new Uint8Array(wasmModule.instance.exports.memory.buffer);

    // 然后，让我们获取指向在 wasmMemory 中的缓冲区的指针
    let bufferPointer = wasmModule.instance.exports.getWasmMemoryBufferPointer();

    // 接着，让我们通过访问 wasmMemory[bufferPointer + bufferIndex] 的索引来读取缓冲区中索引为 0 的已写入的值
    // 应该输出 "24"
    console.log(wasmMemory[bufferPointer + 0]);

    /**
    * 第二部分：在 JS 中写入，在 Wasm 中读取
    */
    console.log("在 JS 中写入，在 Wasm 中读取，索引为 1：");
    // 首先在缓冲区的索引 1 上写入值
```

```
    wasmMemory[bufferPointer + 1] = 15;

    // 然后，让 wasm 读取缓冲区中的索引 1，并返回结果
    // 应该输出 "15"
    console.log(
        wasmModule.instance.exports.readWasmMemoryBufferAndReturnIndexOne()
    );

};

runWasm();
```

利用共享内存空间我们可以在 js 以及 Go 之间传递数据，相对来说也很方便，实现高效的跨语言交互。

示例

格式化 json 对象

json.go：

Go

```go
package main

import (
    "encoding/json"
    "fmt"
    "syscall/js"
    "time"
)

func main() {
    fmt.Println("Hello WebAssembly!")
    //注册为全局变量
    js.Global().Set("prettyJSON", jsonWrapper())
    <-make(chan bool)
}

func prettyJson(input string) (string, error) {
    start := time.Now()
    var raw interface{}
    if err := json.Unmarshal([]byte(input), &raw); err != nil {
        return "", err
    }
    pretty, err := json.MarshalIndent(raw, "", " ")
    if err != nil {
        return "", err
    }
    cost := time.Since(start)
    fmt.Println("wasm json耗时: ", cost)
    return string(pretty), nil
}

// 将Go函数封装并返回js.Func对象，才可以被js所调用
func jsonWrapper() js.Func {
    jsonFunc := js.FuncOf(func(this js.Value, args []js.Value) interface{}{
        if len(args) != 1 {
            return "Invalid no of arguments passed"
        }
        inputJSON := args[0].String()
```

```
39          // fmt.Printf("input %s\n", inputJSON)
40          pretty, err := prettyJson(inputJSON)
41          if err != nil {
42              fmt.Printf("unable to convert to json %s\n", err)
43              return err.Error()
44          }
45          return pretty
46      })
47      return jsonFunc
48  }
49
```

定义 html 文件：

XML/HTML

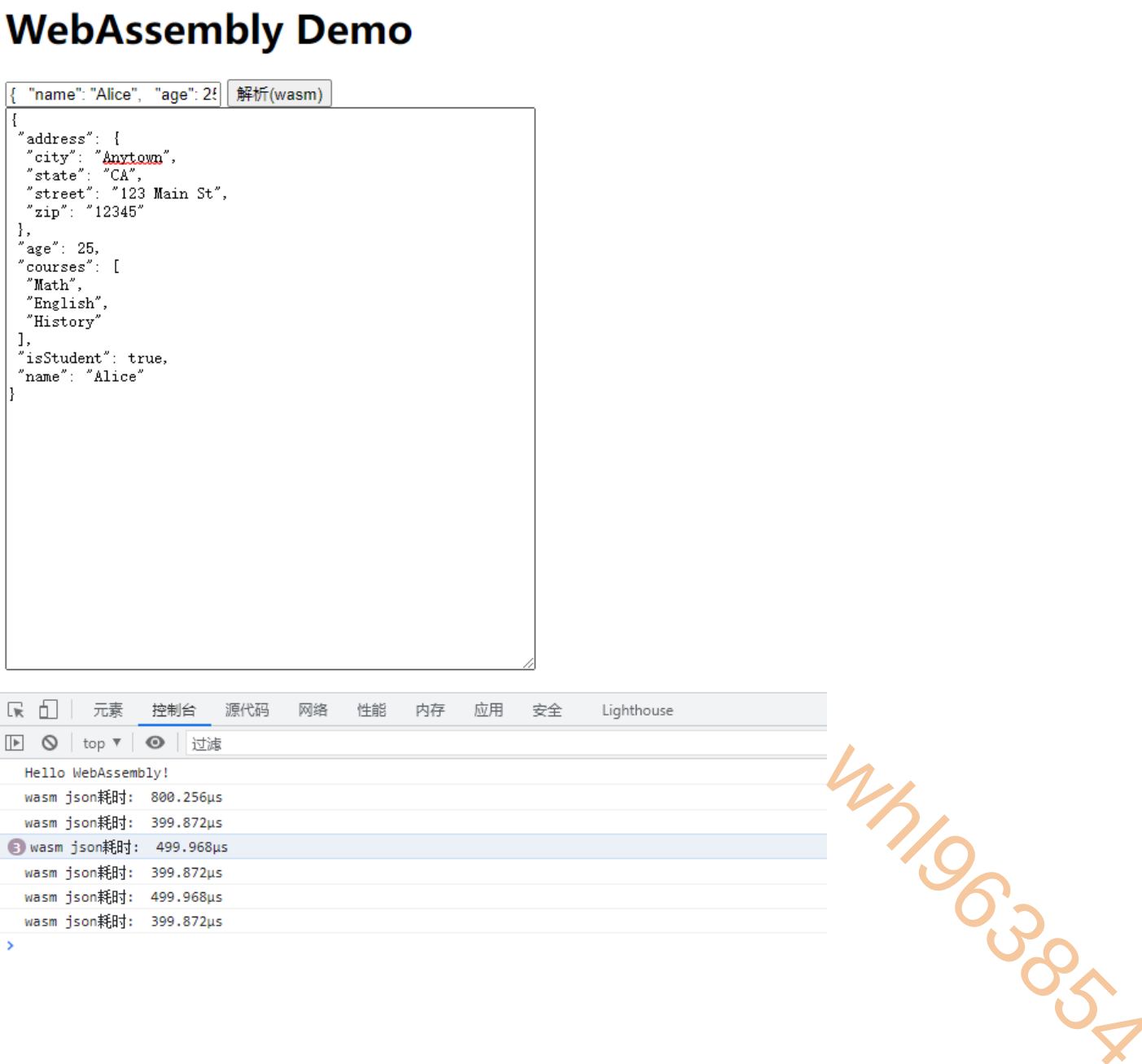
```
1  <html>
2  <head>
3      <meta charset="utf-8"/>
4      <script src="wasm_exec.js"></script>
5      <script>
6          const go = new Go();
7          WebAssembly.instantiateStreaming(fetch("main.wasm"), go.importO
   bject).then((result) => {
8              go.run(result.instance);
9          });
10     </script>
11 </head>
12 <body>
13 <h1>WebAssembly Demo</h1>
14 <input type="text" id="jsonInput">
15 <button id="btn-wasm">解析(wasm)</button>
16 <br />
17 <textarea id="prettyJsonArea"></textarea>
18 <script src="handle.js"></script>
19 </body>
20 </html>
```

定义 handle.js:

JavaScript

```
const convertBtnByWasm = document.getElementById('btn-wasm');
const jsonInput = document.getElementById('jsonInput')

convertBtnByWasm.addEventListener('click', () => {
const content = jsonInput.value;
const pretty = window.globalThis.prettyJSON(content);
prettyJsonArea.value = pretty
})
```

运行结果：



wasm 的不足

交互复杂

虽然 Wasm 可以独立运行，但由于 wasm 模块运行在一个隔离的沙盒环境中，与主体应用的 JavaScript 或其他语言的代码有着明确的内存分隔，在 Web 环境中，WASM 代码无法直接访问浏览器的 Web API（如 DOM 操作、网络请求等），而必须通过 JavaScript 来进行中介，Wasm 与 JavaScript 之间的交互是通过 JavaScript 的 API 进行的。复杂数据类型需要进行编解码，对于除“数字、字符串”以外的类型（例如：对象、数组）需要先编码成二进制再存放到 WASM 的内存段里。在使用 WebAssembly 进行开发时，仍然需要编写一定量的 JavaScript 代码来与浏览器交互，这会引入一些性能损耗和额外的复杂性。与 JavaScript 胶水代码的交互带来的性能损耗在一定程度上抵消了 WASM 本身带来的性能红利。

生态仍不完善

Wasm 技术的生态系统仍在不断发展和成熟，相关工具和库的数量和质量仍然有待提高，缺乏一些成熟的工具支持。这可能会影响到开发效率和稳定性，比如仍然需要通过 JavaScript 与实际的数据库进行通信，然后利用其他语言进行数据处理。

编译工具依赖：将普通的编程语言代码编译为 wasm 格式时，需要使用特定的编译工具，例如 Emscripten、TinyGo、wasm-pack 等。这些编译工具通常需要额外的配置和学习成本，并且可能在不同的平台和开发环境中具有不同的使用方式，可能会导致在开发和部署过程中出现一些困难。

运行时环境依赖：在运行 wasm 插件时，需要在主体应用中提供相应的运行时环境，例如浏览器的 JavaScript 运行时或者服务器端的 wasm 运行时。这意味着如果目标环境不支持 wasm 运行时，或者运行时环境的版本不兼容，那么 wasm 插件将无法正常运行。