21 WebAssembly

什么是 WebAssembly

WebAssembly 是一种把源代码编译成接近机器代码的，并运行在 WebAssembly 虚拟机上的技术。既然是一种虚拟机技术，那么就是说，只要有对应的虚拟机，就可以运行 WebAssembly，不局限在浏览器中。只不过目前来说，主流的支持场景还是浏览器。就像 Java 的源代码被编译成 `.class` 文件，然后运行在 Java 虚拟机上，所以 Java 有一句名言：“Write once, run anywhere”。可以模仿一下这句话，Rust 编写的 WebAssembly 是：

> *Write once, run faster, safe and anywhere*

WebAssembly 的设计目标有：

1. 安全性和通用性。Web 平台上的代码安全性实际上并不高，使用 WebAssembly 可以提升代码安全性。另外，WebAssembly 也是在沙盒中运行的，限制代码访问特定资源，提高了安全性
2. 高性能。据说 WebAssembly 的性能是 JavaScript 的 5 倍，这里应该是包含了加载、解释和执行的时间
3. 代码复用。有一些使用原生代码开发的模块，可以将其编译成 WebAssembly 来实现代码复用

既然 WebAssembly 是一种技术规范，也就是说我们可以用不同的原生语言开发 WebAssembly，例如：Emscripten + C/C++ 、wasm-pack + Rust、J2CL + Java 都可以用来开发 WebAssembly。

WebAssembly 不是用来取代 JavaScript 的，它是在对于性能和安全性要求高的应用中，对 JavaScript 的有效补充。

从 Hello World 开始吧

使用 Rust 开发 WebAssembly 需要安装 `wasm-pack` 组件。在命令行窗口中执行：

```
$ cargo install wasm-pack
```

接下来，我们创建一个项目：

```
$ cargo new hello-wasm --lib
```

首先，修改 `Cargo.toml` 增加项目类型段：

```
[package]
name = "hello-wasm"
version = "0.1.0"
edition = "2021"

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

[dependencies]
wasm-bindgen = "0.2.92"

```

行 6、行 7 是手动加入的。

然后进入到项目目录，添加 `wasm-bindgen` 依赖。

```
$ cargo add wasm-bindgen
```

接下来，把 `src/lib.rs` 中的代码删除，改为下面的代码：

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

// 将该函数导出为 WebAssembly 模块
#[wasm_bindgen]
pub fn greet(name: &str) -> String {
    format!("Hello, {}!", name)
}
```

行 4 的属性宏 `#[wasm_bindgen]` 表明这是一个要导出到前端使用的函数。

下一步，构建项目。这次，我们不再使用 `cargo` 直接构建，那样太麻烦。我们使用工具包 `wasm-pack` 来构建：

```
$ wasm-pack build --target web
```

第一次构建的时候，Rust 会下载相关的组件，如果由于网络原因失败了，可以尝试下面的方式：

使用 `cargo` 安装和项目中的 `wasm-bindgen` 版本一致的 `wasm-bindgen-cli`：

```
$ cargo install wasm-bindgen-cli --version 0.2.92
```

如果在 `wasm-pack` 出现了下面的错误消息：

```
    Finished `release` profile [optimized] target(s) in 0.01s
Error: failed to download from https://github.com/WebAssembly/binaryen/releases/download/version_111/binaryen-version_111-x86_64-macos.tar.gz
To disable `wasm-opt`, add `wasm-opt = false` to your package metadata in your `Cargo.toml`.
```

有两种解决办法：

解决方案一

在 `Cargo.toml` 中增加这么一段，禁用 `wasm-opt`。`wasm-opt` 是一个优化工具，可以让最终输出的 `.wasm` 文件更小。

```
[package.metadata.wasm-pack.profile.release]
wasm-opt = false
```

解决方案二

自己在浏览器中前往 https://github.com/WebAssembly/binaryen/releases ，找到报错信息中提供的对应的版本号，下载对应的版本，然后解压缩，并且把解压缩后的，例如在我电脑上是 `/Users/yuan/Applications/binaryen-version_111/bin` 这个路径加入到系统的环境变量 `PATH` 中去。

成功构建之后，在项目目录下会生成 `pkg` 目录以及这样的一些文件：

```
.
├── hello_wasm.d.ts
├── hello_wasm.js
├── hello_wasm_bg.js
├── hello_wasm_bg.wasm
├── hello_wasm_bg.wasm.d.ts
└── package.json
```

然后，我们就可以编写一个测试页面了：

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Hello WebAssembly</title>
</head>
<body>
    <h1 id="output"></h1>
    <script type="module">
        import init, { greet } from './pkg/hello_wasm.js'; // 确保路径正确
```

```
11
12          async function run() {
13              const wasm = await init();
14              const greeting = greet("World");
15              document.getElementById("output").innerText =
    greeting;
16          }
17          run();
18      </script>
19  </body>
20  </html>
```

`wasm-pack` 在构建项目的时候，可以 使用 `--target` 参数指定构建目标。目前支持的有：

选项	解释
不指定 `--target` 参数，或者指定为 `bundler`	输出和诸如 Webpack 之类的打包器适配的 JS。在 `pkg/package.json` 文件中可以找到 `import` 时候对应的包名
`nodejs`	输出给 Node.js 使用的包。支持使用 `require` 引入到 Node.js 项目
`web`	输出 ESM 的包，可以在浏览器中使用，但是需要手动初始化 WASM。上面的测试页面的 HTML 源码中演示的就是这种情况
`no-modules`	和 `web` 类似，但是会在 Global 空间注入一个名为 `wasm_bindgen` 的变量，导出的函数可以从这个对象解构出来。不过这种模式相对较老了
`deno`	输出给 Deno 用的包，也是 ESM 的

完整的信息可以参考 wasm-bindgen 的官方文档：https://rustwasm.github.io/docs/wasm-bindgen 。

在 WebAssembly 中操作 DOM

在 WebAssembly 中，借助 `web-sys` crate ，可以实现直接操作 DOM 的能力。首先，我们在 `Cargo.toml` 中增加 `web-sys`。这次我们采用另外一种写法来增加依赖：

```
[dependencies.web-sys]
version = "0.3.4"
features = [
  "Document",
  "Element",
  "HtmlElement",
  "Node",
  "Window",
]
```

这种写法和 `web-sys={ version = "0.3.4", features = ["Document", "Element", "HtmlElement", "Node", "Window"]}` 是等效的，就看自己喜欢那种方式了。

然后，修改的 `src/lib.rs` 文件，增加一个方法：

```
#[wasm_bindgen(start)]
fn run() -> Result<(), JsValue> {
    let window = web_sys::window().expect("no global `window` exists");
    let document = window.document().expect("should have a document on window");
    let body = document.body().expect("document should have a body");

    // 组装要添加的 DOM 元素
    let val = document.create_element("p")?;
    val.set_text_content(Some("Hello from Rust!"));

    body.append_child(&val)?;

    Ok(())
}
```

行 1 中的属性宏 `#[wasm_bindgen(start)]` 这里多了一个 `(start)`，表示这个函数是在前端加载 wasm 模块之后立即执行的函数。注意，标注为 `#[wasm_bindgen(start)]` 的函数必须无任何输入参数，并且返回值是 `()` 或者 `Result<(), JsValue>`。并且，一个 wasm 包中最多只允许一个函数带有这个标记。另外，由于这个标记比较新，所以使用的时候可能会遇到问题，所以还是建议编写传统的函数，然后在前端页面中调用。

行 3 到行 5 是通过 `web-sys` crate 提供的能力，在 Rust 代码中访问前端相关的对象。

行 8 到行 11 的代码对于前端同学来说，那就太熟悉了，不再解释了。

使用 `wasm-pack build --target web` 构建一下项目，再访问一下测试用的 HTML，就可以看到我们从 Rust 一侧创建了一个 DOM 元素，并且赋予了一段内容。

实战：使用 WebAssembly 隐藏数据加密过程

根据国家《信息系统安全等级保护备案》相关规定，在公共网络上进行通信的时候，对于涉及到用户隐私的信息，要进行加密之后再传输。比如 Web 应用中常见的用户手机号、密码、身份证号、银行账号等，都属于用户隐私信息。所以，当我们开发一个用户注册登录的功能的时候，就需要对这些数据进行加密再提交给后台 API。

网上能够找到很多使用 JavaScript 在浏览器中加密数据的代码，很多人会选择 DES 之类的对称加密。这样的话，密钥必须得放到 JavaScript 代码中。虽然可以通过混淆代码等技术让 JavaScript 代码不容易看懂，但是有经验的攻击者还是可以找到你的算法以及密钥。如果你只是为了合规，那么也就够了。但是如果你真的想加强 Web 应用的安全性，可以考虑采用 Rust 来编写加密算法，然后发布成 wasm 给前端使用。但是，现在也有针对 wasm 内存分析的工具，所以对称加密算法还是存在密钥泄露的风险。这个时候，优先选择非对称加密，例如：RSA。公钥随便拿，私钥只有一份在服务器上。然后我们在前端使用公钥加密，在后端 API 使用私钥解密。

但是非对称加密有一个明显的特征：它可以支持的数据量都不大，例如：RSA 加密算法一次可以加密的数据长度是：密钥长度（字节）- 11。也就是说，如果我使用 2048 bit 的密钥，可以一次加密的数据大小是 2048/8 - 11 = 245。虽然不是很大，但是对于手机号和密码数据来说，足够了。因为手机号 11 位，密码我们可以限制到最长 30 字符，那么已经是足够了。

首先，配置我们的 `hello-wasm` 项目，增加需要的 crate：

```
[dependencies]
base64 = "0.22.1"
rand = "0.8.5"
rsa = "0.9.6"
wasm-bindgen = "0.2.92"

[dependencies.getrandom]
version="0.2.15"
features = ["js"]
```

这里，`rand` 是 `rsa` 所需要的，`base64` 用来编解码，`getrandom` 是 `rand` 中用到的一个随机数生成工具，它支持 wasm 构建，但是需要打开 `js` 这个特性。

然后我们就可以编写 `src/lib.rs` 中的代码了。注意：下面的代码仅供演示，并没有做完备的数据有效性检查、错误检查等。大家在真实的开发工作中还是需要对输入的数据及调用外部函数返回的结果进行检查的。

```

#[wasm_bindgen]
pub fn encrypt_message(msg: &str) -> String {
    let public_key_pem = "-----BEGIN PUBLIC KEY-----
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAuWsnnaSME/h32TAGaqup
Tn9U1IXHYWvVxZK7b0jHZkv2M1+Tc9b34pWyWXPmCo3+q5GuKWYWRk6bbflFXFgO
QtIYbNS10172i/ka/7yrKS3/xHzPGhQEGpdvrCn7Ki2jHLBY7pP4xz1CgXJlN3PG
ouPUHD1HRHEVo9HdKvDGqOuphyUsNhwFS6z6dcReWit7RoVFj/tI62a1uGCb9h1N
3vCjV0O9IsWqS1gF0u5elbmsTSRAJvUkTsVN1oauf9sGlA1w7+5zEhta5FAz1z/2
MU2XXhfwSizDlrpMGfN3qWbktyqbhkkm8TFJvn8H3CXXbJ8wJ5ChAvE5O2NzCINl
mwIDAQAB
-----END PUBLIC KEY-----";

    let public_key =
rsa::RsaPublicKey::from_public_key_pem(public_key_pem).unwrap();
    let mut rng = rand::thread_rng();

```

```
    let enc_data = public_key.encrypt(&mut rng, Pkcs1v15Encrypt, msg.as_bytes()).unwrap();
    BASE64_STANDARD.encode(enc_data)
}

#[wasm_bindgen]
pub fn decrypt_message(msg: &str) -> String {
    let private_key_pem = "-----BEGIN RSA PRIVATE KEY-----
MIIEpgIBAAKCAQEAuWsnnaSME/h32TAGaqupTn9U1IXHYWvVxZK7b0jHZkv2M1+T
c9b34pWyWXPmCo3+q5GuKWYWRk6bbflFXFgOQtIYbNS10172i/ka/7yrKS3/xHzP
GhQEGpdvrCn7Ki2jHLBY7pP4xz1CgXJlN3PGouPUHD1HRHEVo9HdKvDGqOuphyUs
NhwFS6z6dcReWit7RoVFj/tI62a1uGCb9h1N3vCjV0O9IsWqS1gF0u5elbmsTSRA
JvUkTsVN1oauf9sGlA1w7+5zEhta5FAz1z/2MU2XXhfwSizDlrpMGfN3qWbktyqb
hkkm8TFJvn8H3CXXbJ8wJ5ChAvE5O2NzCINlmwIDAQABAoIBAQC07yupWXiXxE6v
UX3xebGtN/O/rwvwtRnVwmnFHm/2Ewoc95cb4xnhsdJoGADTK9zn7zDUlRoHYY2q
syRMCGFiM0JCcQ89Q92ymVMPWzxq4shDJMez2vRmPoBqsEy3y/DvfeZKaXrKca0E
RG0gv4mGSmwhYmq5DuDfC1uXG33ANw218Y2JLHCEr2sIeFqldkRJFJBa3GS/8A0v
5PApxlOsRtjaqqWHAZ3YUur5rI1We2ifGp/qb78KCkvsguaCBfuExNaBN16674fA
XJocuXAZT334u5FyYZbe186qGT41iSXkEWdJwFE1SRp3DS5tsSnU42kA1uIPK/yB
wE4uGfEBAoGBAOE7IKQ13s0oVHDQThlHEfW2zW1K1B/SNTYhbtoAAY9vnTJ4RwyD
46Tzod4624GZr1NPE6mGizGIQxjRYPZOL6tnd40b3pS6ghaZq6XTb8TAo31jYAI6
cOHK0x+R5kZueQ15Mz0NKW8DsSqnEM5eLA4eqf5nkjEHU9oUri/NHoCjAoGBANK/
sP+p7S87G6EObFR0cvUzIiB0XpBLY0Z1gLZwGAoQr1RMr/WoZb+N9Qlgg49IOPpq
W2aBm6z86X4ivgN+q01L97XRUQFuaCGT9dqUOjSRRDxnf3Z98sxWs04VwhhBn+mi
jnoY5K+1CGqq5w+wcnydY23wKJHGp6L8tI7xcz6pAoGBANPzmXcOftmfrz7ut+AP
wnXxjgXgmhhL7+k9sxyRTuaVSWAaUfoUD2DGKLiBuNmQL659RuqMg2acbU/okY+X
6kIpvxZaeK6LiSoNeLs3awHDqtavcsUGC/5fqSgXW5VCFsa4HKDCRxCCb/HMgANE
Gb872gtfmfJri9w5A+ZV5/NzAoGBAKsYgUQu2qaboCSzJvOxzpqOtsgOca9H5QMQ
5jw26O1S+mTAFaKk4mYPg73nIyeyBMAYlLe1xYM9mPqUjhscPUp1wFdV5iP5VITe
MJ0R9eczgGhhxjOWsQV/5fYg6AD1VjRBitZJW2/i00B2Gzy7jVbqGAzrCqS+2N4T
tyrNjzhxAoGBAItDo2VDjZZM8/snwiEdFLSc6HTh/7MBrobemG2LviB0vYMYFlC3
bhhLg2a52aLbBHFgKoYRihMtrrJbplmm3Rv4/+irxaz0fEpp4AArK25+iu/TydWz
w8foHI6I9lOs3pvI1mDxB/WkWjylB6lm3Jg3QpF660Y80DbysecdAWUg
-----END RSA PRIVATE KEY-----";
    let private_key = rsa::RsaPrivateKey::from_pkcs1_pem(private_key_pem).unwrap();
    let data = BASE64_STANDARD.decode(msg).unwrap();

```

```
53      let plain_data = private_key.decrypt(Pkcs1v15Encrypt,
    &data[..]).unwrap();
54      String::from_utf8(plain_data).unwrap()
55  }
```

核心代码其实很少，也是很直观的。另外，出于演示的目的，我们把私钥以及解密的函数也放到这个 wasm 中了，真实场景中，私钥和解密的逻辑一定是在后端的 API 项目中，而不会存在于 wasm 中的。

实战：

大家自己完成前端页面的调用。

附：使用 OpenSSL 生成密钥

首先需要在电脑上安装 OpenSSL 工具。如果你使用的是 Windows 系统，可以前往 https://openssl.org 下载源代码然后本地构建，或者前往 https://slproweb.com/products/Win32OpenSSL.html 下载构建好的安装包。如果是 macOS 或者 Linux，系统已经自带。

生成一个 2048 位的私钥：

```
1  $ openssl genrsa -out private.pem 2048
```

然后使用这个私钥文件生成一个公钥文件：

```
1  $ openssl rsa -in private.pem -outform PEM -pubout -out
   public.pem
```