WebAssembly 系统接口（WASI）

黄梓淳

Jun 12, 2026

WebAssembly 最初诞生于浏览器环境，旨在提供接近原生的执行速度，同时通过严格的沙箱机制保障安全。它能够执行事先编译好的字节码，但却缺少与宿主操作系统直接交互的能力。这意味着在浏览器里运行的 WebAssembly 程序无法直接访问文件系统、读取时钟、生成随机数或者发起网络请求。开发者若想实现这些功能，只能借助 JavaScript 作为桥梁，造成性能损耗和复杂性上升。
WASI 的出现正是为了填补这一空白。它将 POSIX 风格的系统调用以标准化接口的形式引入 WebAssembly，让同一份二进制文件能够在浏览器之外的多种环境中运行。无论是云原生服务、边缘计算节点，还是 Serverless 平台，WASI 都提供了「一次编译、到处运行」的基础。接下来的内容将依次梳理 WASI 的设计动机、核心概念、架构细节、与传统操作系统的差异，以及实际落地时的工具链与案例。

1　背景与动机

WebAssembly 的安全模型建立在 capability-based 安全之上。程序只能访问显式授予的资源句柄，而无法随意读取全局命名空间。这种设计在浏览器里行之有效，但在脱离浏览器后，缺失的系统调用便成为瓶颈。云原生与边缘计算场景要求工作负载能够快速迁移、弹性伸缩，同时保持强隔离。开发者自然希望用同一种二进制格式覆盖从数据中心到物联网设备的全链路。
字节码联盟、Mozilla、Fastly 与英特尔等组织在 2019 年左右启动了 WASI 标准化工作。他们的目标是定义一套最小、能力受控的系统接口，让 WebAssembly 真正成为通用计算平台。WASI 既要保留 WebAssembly 的安全优势，又要提供文件、时钟、随机数等基础能力，从而在多语言、多运行时之间实现互操作。

2　WASI 核心概念

Capability-based Security 是 WASI 的灵魂。每一个文件描述符本质上都是一个能力句柄，宿主在创建句柄时就决定了该句柄允许执行的操作集合。程序无法通过猜测路径或枚举描述符来突破限制，这与传统操作系统中「环境权限」模型形成鲜明对比。
接口描述则依赖 WIT（Wasm Interface Types）。WIT 用声明式语法描述函数签名、资源类型与错误处理，编译器据此生成跨语言绑定。Preview 0 版本仅覆盖文件系统与时钟，Preview 1 增加了套接字与随机数，Preview 2 则引入组件模型（Component Model），允许把多个 WebAssembly 模块组合成更复杂的应用。系统调用表被模块化拆分，文件系统、时钟、随机数、Poll 与路径查找各自独立，便于运行时按需实现或裁剪。

3 技术架构剖析

WASI 的分层模型自上而下依次为应用层、接口层、宿主层与系统层。应用层是编译后的 WebAssembly 字节码，接口层则由 WIT 定义的函数签名和 Canonical ABI 组成。Canonical ABI 负责在 WebAssembly 线性内存与宿主原生类型之间完成参数编解码，保证不同语言实现的一致性。宿主层通常是 Wasmtime、Wasmer 或 WAMR 等运行时，它们把 WASI 调用映射为真实操作系统调用。底层系统层可以是 Linux、macOS、Windows，也可以是 Unikernel 或微内核。

组件模型解决了多模块组合问题。通过定义资源、接口与适配器，开发者可以将不同语言编写的模块像乐高一样拼装，同时保持类型安全与资源隔离。线性内存、函数表与栈在模块之间默认隔离，必要时可通过显式共享机制传递数据。异步方面，WASI 使用 Poll 与 Epoll 风格的接口，把单线程事件循环映射到多线程或协程模型，降低上下文切换开销。

4 与 POSIX、Linux 的对比

WASI 的文件操作接口与 POSIX 存在映射关系：`fd_read` 对应 `read`，`fd_write` 对应 `write`，`path_open` 对应 `open`。但 WASI 缺少 `fork`、`exec`、`signal` 等进程管理调用，也不支持传统共享内存。这些缺失既出于安全考虑，也因为 WebAssembly 本身是单地址空间模型。

安全模型上，WASI 采用显式能力列表，而 POSIX 依赖进程凭证与访问控制列表。性能方面，WASI 调用通常需要一次从 WebAssembly 到运行时的上下文切换，若使用 AOT 编译可减少 JIT 开销，但仍需权衡沙箱边界带来的额外成本。总体而言，WASI 在安全与可移植性上更进一步，而 POSIX 在功能完备性上更胜一筹。

5 运行时与工具链生态

主流运行时包括 Wasmtime、Wasmer、WAMR 与 WasmEdge。Wasmtime 由字节码联盟维护，强调安全与标准符合；Wasmer 提供开箱即用的命令行工具与语言绑定；WAMR 针对嵌入式与实时场景优化；WasmEdge 则侧重边缘 AI 与网络函数。浏览器也在逐步原生支持 WASI 子集，例如 Chromium 的 SPKI 与 Firefox 的实验性实现。

语言支持矩阵持续扩大。Rust 可直接以 `wasm32-wasi` 为目标编译，C/C++ 通过 wasi-sdk，Go 借助 TinyGo，AssemblyScript 与 SwiftWasm 则提供更高层次抽象。打包工具如 `cargo-wasi` 可自动生成 WASI 兼容的 wasm 文件。调试方面，DWARF 信息可映射回源代码，OpenTelemetry 集成则提供分布式追踪能力。

6 典型应用场景与案例

在 Serverless 领域，Fastly Compute@Edge 与 Fermyon Spin 均基于 WASI 构建函数平台。开发者上传 WebAssembly 模块后，平台按需实例化并注入文件与网络能力，实现毫秒级冷启动。插件系统同样受益，Istio Wasm Filter 把 WebAssembly 模块作为数据面扩展，Envoy 通过 WASI 接口实现自定义过滤逻辑。

边缘 AI 推理是另一热点。WasmEdge 结合 OpenVINO，可在边缘节点上运行 TensorFlow Lite 模型，WASI-NN 接口负责张量数据传递。安全沙箱方面，Shopify Functions 使用 WASI 隔离第三方代码，Cloudflare Workers 也在部分模块中采用类似机制。物联网领域，WASI 与 seL4、Unikraft 等 Unikernel 结合，可在资

源受限设备上实现安全、可验证的应用加载。

7 挑战与未来演进

Preview 2 组件模型的落地仍需解决兼容性与工具链成熟度问题。网络套接字与异步 I/O 的标准化正在进行，线程支持也处于提案阶段。WASI-NN、WASI-Crypto、WASI-Filesystem 等子系统提案将进一步丰富生态。与 eBPF、gVisor、Firecracker 的关系既是互补也是竞争：eBPF 更适合内核态可编程，gVisor 与 Firecracker 提供更重的虚拟化边界，而 WASI 则在轻量级与语言无关性上占据优势。最终的权衡仍需在性能、安全与易用性之间找到平衡。

8 实践指南：从零写一个 WASI 程序

以 Rust 为例，首先安装稳定版工具链并添加 `wasm32-wasi` 目标。

```
rustup target add wasm32-wasi
```

接着新建项目并在 `Cargo.toml` 中声明依赖。

```
[package]
name = "wasi-demo"
version = "0.1.0"
edition = "2021"

[dependencies]
sha2 = "0.10"
```

示例代码演示如何打开文件、计算 SHA-256 并写回结果。

```
use std::fs::File;
use std::io::{Read, Write};
use sha2::{Sha256, Digest};

fn main() {
    // 以只读方式打开输入文件，WASI 会检查能力句柄
    let mut input = File::open("/input.txt").expect("open input");
    let mut data = Vec::new();
    // 读取全部内容到内存
    input.read_to_end(&mut data).expect("read");

    // 计算 SHA-256 摘要
    let mut hasher = Sha256::new();
    hasher.update(&data);
    let hash = hasher.finalize();
```

```
    // 以写方式打开输出文件
    let mut output = File::create("/output.txt").expect("create output");
    // 将十六进制结果写入文件
    write!(output, "{:x}", hash).expect("write");
}
```

编译生成 WebAssembly 模块。

```
cargo build --target wasm32-wasi --release
```

使用 Wasmtime 在本地运行。

```
wasmtime --mapdir /::/tmp run target/wasm32-wasi/release/wasi-demo.wasm
```

发布到 Spin 或 Fastly 时，需确保文件路径映射与随机数熵池已正确配置，并检查时区环境变量是否被正确传递。常见问题包括文件路径被沙箱重写、缺少随机数源导致哈希不一致，以及时区信息缺失导致时间解析错误。逐一排查后即可实现从开发到生产的完整闭环。

9 结论与行动号召

WASI 正在把 WebAssembly 从浏览器特性转变为通用计算平台。对开发者而言，它提供了安全、可移植的运行时选择；对平台工程师而言，它降低了多语言函数的集成成本；对架构师而言，它为云原生与边缘计算提供了新的抽象层。建议持续关注 WASI 官方仓库与字节码联盟社区，参与 RFC 讨论，并在实际项目中尝试 WASI 运行时，以把握这一技术演进带来的机遇。