Engineer

WebAssembly

サンドボックス内で実行できる、移植可能なバイナリコード形式

どのようなものか

WebAssemblyは、通常Wasmと略される、実行可能なコードのバイナリ形式です。コア仕様では「効率的な実行とコンパクトな表現を目的とした、安全で移植可能な低水準のコード形式」と定義され、仮想的な命令セットとして説明されています。つまり、架空のプロセッサー向けの命令を、ブラウザーなどのホストがまず検証し、実際のプロセッサーの機械語に変換します。Wasmは単語を縮めた表記で、頭文字を並べた略語ではありません。そのため、仕様でもWASMではなくWasmと書かれています。

`.wasm` ファイルは、通常、別の言語のコンパイラーが出力したものです。仕様によれば、主な目的はウェブ上のアプリケーションを高速に動かすことですが、形式自体はウェブを前提としていません。そのため、単独の仮想マシンとしても、別のプログラムの内部でも実行できます。標準もそれに合わせて層に分かれています。コア仕様が定めるのは命令、形式、検証、実行であり、特定の実行環境ではありません。モジュールのコンパイル、リンク、呼び出しを行うJavaScript APIと、ネットワークから読み込むWeb APIは、それぞれ別の文書で定義されています。

最初の版は2019年12月5日にW3C勧告になりました。後継版より3年早く実質的に完成していた2.0では、ベクトルを扱う命令、メモリーの一括操作、複数の戻り値を持つ関数、参照型の初歩的な仕組みが加わりました。2025年9月17日、プロジェクトは3.0を現行の標準として発表し、主要ブラウザーの大半ですでに提供されていると説明しました。同じ発表では、Wasmのガベージコレクターを利用してWasm向けにコンパイルできる言語として、Java、OCaml、Scala、Kotlin、Scheme、Dartなどが挙げられています。

基本用語は多くありません。コードの単位はモジュールです。関数、テーブル、メモリー、グローバル変数を持ち、必要なものをインポートします。各インポートはモジュール名と項目名で指定し、外部に提供するものは独自の名前でエクス

ポートします。数値型は、32ビットと64ビットの整数および浮動小数点数である `i32` 、 `i64` 、 `f32` 、 `f64` の4種類です。このほか、128ビットのベクトル型1種類と、内部構造を直接扱えない参照があります。整数自体には符号の区別がなく、各演算が符号付きか符号なしかを決めて解釈します。32ビット整数は真偽値にも使われ、メモリーアドレスには `i32` を使い、64ビットメモリーでは `i64` を使います。命令は暗黙のスタックからオペランドを取り出し、結果を同じスタックに置きます。制御フローは構造化されており、ブロック、ループ、条件分岐が入れ子になり、分岐先にできるのはそのいずれかだけです。

バイト単位でアドレスを指定するデータは、線形メモリーに置かれます。これは連続した書き込み可能なバイト配列で、宣言したサイズから始まり、実行中に拡張できます。現在の範囲を外れた読み書きが、別の領域に届くことはありません。その場合はトラップが発生し、実行が直ちに停止します。トラップはWasm内で処理できず、周囲のホストに報告されるだけです。以下の例では、仕様が人間の可読性のために定めたテキスト形式を使います。

概念を判断に生かす

「WebAssemblyとは？」を実際の判断に生かしましょう。その概念を、開発または購入するシステムに結び付けて考えるお手伝いをします。 読み込みが速く、スマートフォンでも使え、お客様がどのブラウザーを使っていても同じように動く画面を提供します。

サービスを見る お問い合わせ →

用語の背後にある前提を検証する

解説とリンク先の実績を参考に、プロジェクトについての質問を整理してください。ツールを選ぶ前に、前提を一緒に検討できます。 フルスタックのGIS地図開発を指揮 — DBからブラウザーまでのフルスタック地図開発

フルスタックのGIS地図開発を指揮 お問い合わせ →

なぜ重要なのか

第1の理由は速度です。設計目標には、ネイティブコードに近い性能と、JITでもAOTでも高速な1回の走査でデコード、検証、コンパイルできる形式が掲げられています。バイナリは、テキストやネイティブの機械語より小さくなるよう設計され、ストリーミングにも対応します。最後のバイトが届く前に処理を始められます。バイナリ形式では、各関数の型宣言と本体さえ別のセクションに置くため、本体を受信しながら並列にコンパイルできます。別の言語で十分にテストされたライブラリーを持つチームなら、JavaScriptで書き直さず、ブラウザー向けにコンパイルできます。

第2の理由は移植性です。同じモジュールを、デスクトップでもモバイルでも、現在のどのプロセッサーでも実行できます。また、エンジンを組み込んだホストなら、ブラウザー、単独のランタイム、別のプログラムのいずれでも動きます。

仕様は特定の言語、プログラミングモデル、オブジェクトモデルを優遇しません。したがって、この形式は1つの言語専用のランタイムではなく、共通のコンパイル先になります。

第3の理由は隔離です。モジュールには、実行先のコンピューターを暗黙に操作する権限がありません。ホストが対応する関数をモジュールへのインポートとして渡さない限り、ファイル、ネットワーク、ページ、時計にはアクセスできず、触れてよいものをホストが厳密に決めます。Web APIでは、WasmはJavaScriptを実行できる場所で実行でき、脅威モデルも基本的に同じで、JavaScriptで公開できる範囲を超えて機能を公開できないと説明しています。3.0では、この境界の内側で扱えるものが増えました。64ビットでアドレスを指定するメモリーは、ウェブ上では16ギガバイトに制限されます。このほか、1モジュール内の複数メモリー、ガベージコレクションを必要とする言語向けの仕組み、ホストまで逃がさずに扱える例外が加わりました。発表では、例外をホストに逃がす方法は、移植性にも効率にも欠けると説明されています。

実際の活用例

次は、32ビット整数2つを加算する関数を1つだけエクスポートする、テキスト形式のモジュールです。

```
(module
  (func (export "add") (param i32 i32) (result i32)
    local.get 0
    local.get 1
    i32.add))
```

関数の型は、2つの `i32` を受け取り、1つを返すことを示しています。本体はスタックを操作する3命令です。 `local.get 0` が第1引数を、 `local.get 1` が第2引数を積み、 `i32.add` が両方を取り出して和を積みます。その和が結果としてスタックに残ります。 `(export "add")` は外部から見える関数名を指定します。バイナリ形式に変換すると、ファイルは `0x00 0x61 0x73 0x6D` という4バイト、つまりゼロと文字列asmで始まり、その後にバイナリ形式のバージョンが続きます。このバージョンは今も1です。

ブラウザーでは、JavaScript APIで読み込みます。

```
const url = "/add.wasm";
const { instance } = await WebAssembly.instantiateStreaming(fetch(url));
instance.exports.add(2, 3); // 5
instance.exports.add(2147483647, 1); // -2147483648
```

`instantiateStreaming` は、応答を受信している間にモジュールをコンパイルし、コンパイル済みモジュールとそのインスタンスを返してPromiseを解決します。インスタンスのエクスポートは、通常のJavaScript関数です。2回目の呼び出しは、型の意味を示しています。Wasmの整数加算は2の32乗を法として循環し、JavaScript APIは `i32` の結果を符号付き整数として解釈します。そのため、正の最大値に1を足すと、最小の負の値が返ります。例外は投げられません。一方、トラップが起きると例外になります。関数が自身のメモリーの範囲外を読むと、呼び出しは `WebAssembly.RuntimeError` で終了します。

ホストから何かを必要とするモジュールは、その要求を宣言します。モジュールに `(import "env" "log" (func (param i32)))` があれば、インスタンス化には第2引数として `{ env: { log: console.log } }` が必要で、モジュールは `log` を自身の関数と同様に呼び出せます。第2引数を省くと `TypeError` になり、 `env` オブジェクトに呼び出せる `log` がなければ `WebAssembly.LinkError` になります。どちらも、モジュールのコードが動き始める前に失敗します。境界を構成するのはインポートとエクスポートだけで、その型は宣言され、数値型や参照型を含みます。モジュールのメモリー内のアドレスは `i32` で、64ビットメモリーでは `i64` です。

Swift 6でネイティブmacOSクライアントを構築した実績は、別の言語のコードを再利用する、もう1つの方法を説明しています。2026年8月のこの業務では、Rustのコアを静的アーカイブとしてSwiftアプリケーションにリンクし、契約として扱うCインターフェースを通して利用しました。境界を通るポインターと文字列はすべて薄いラッパーが所有し、インターフェースから返るものには、それぞれ所有権の規則があります。扱いを誤ると、コンパイラーの警告もなくリークやクラッシュになります。この業務ではWasmを使っていません。だからこそ比較になります。2つの言語の間に狭い境界を設けるという同じ問題を、インポート一覧とサンドボックス内のメモリーではなく、ネイティブリンクで解決した例です。

問題が起きるところ

コンパイル前にPromiseが拒否される。 `Content-Type` がない場合や、大文字と小文字を区別せずに比較しても `application/wasm` と一致しない場合は、 `TypeError` になります。仕様は追加のパラメーターを認めておらず、 `application/wasm;` のようにパラメーターが空でも不可としています。成功を表さないステータスや、

CORSでアクセスが許可されていない別オリジンの応答でも、同じように失敗します。サーバーのメディアタイプとヘッダーを修正します。自分で取得したバイト列を `WebAssembly.instantiate` に渡せばメディアタイプの検査は省けますが、ストリーミングコンパイルも使えなくなります。

メモリーのビューが使えなくなる。 JavaScriptは、 `memory.buffer` と、その上に作った配列を通してモジュールのメモリーを参照します。非共有メモリーでは、このバッファーは初期状態では固定長です。メモリーが拡張されると、JavaScript APIは固定長バッファーを切り離し、新しいものに置き換えるため、古いバッファー上の配列はメモリーから切り離されます。これはJavaScriptから `memory.grow()` を呼んだ後にも、外からは無害に見える呼び出しの途中で、モジュール内の `memory.grow` 命令が実行された後にも起きます。ページが存在する間ずっと同じビューを保持せず、メモリーを拡張する可能性のある呼び出しの後には、 `memory.buffer` からビューを作り直します。

Wasmが単独でページを操作できると思う。 モジュールには暗黙のアクセス権がありません。ページ、ネットワーク、ストレージへの作用はすべて、JavaScriptがインポートとして渡す関数を通ります。型にも、この境界で制約があります。 `i64` はNumberではなく `BigInt` として通り、 `v128` を引数または戻り値に持つ関数をJavaScriptから呼ぶと、毎回 `TypeError` になります。インポートとエクスポートの一覧を1つのインターフェースとして設計し、そこでは単純な数値を使い、ベクトルはWasm側にとどめます。

サンドボックス内ならプログラムも安全だと思う。 サンドボックスは、モジュールがホストに害を及ぼさないように守ります。モジュール自体を、その内部の誤りから守るわけではありません。仕様でも、メモリーの安全性を保証しない言語をWasmにコンパイルしたときに、線形メモリー内の自身の配置を壊すことまでは防げないと明記しています。コンパイルしたコードのバッファーオーバーフローは、モジュール自身のデータを上書きします。トラップになるのは、メモリー全体の末尾を超えたアクセスだけです。元の言語の検査とテストを保ち、モジュールが外部から受け取るデータは、信頼できない入力として扱います。

バイナリを保護せずに配信する。 `application/wasm` の登録文書は、この形式には完全性や機密性を保護する仕組みがなく、必要ならHTTPSなどの外部の仕組みで保護しなければならないとしています。JavaScriptと同じく、読み込み方や実行方法の一部は、Content Security Policyや同一オリジンポリシーで制限されます。また、Wasmは実際のハードウェアで動く機械語になるため、コア仕様はハードウェア層のサイドチャネル攻撃を受ける可能性も指摘しています。モジュールはHTTPSで配信し、スクリプトと同じ厳格さで、モジュールを変更できる人を管理します。

参考資料

- WebAssembly仕様 — バージョン3.0のコア仕様。構造、検証、実行、バイナリ形式、テキスト形式を定めています。
- 概要 — 定義、設計目標、各文書の対象範囲、セキュリティ上の考慮事項。
- WebAssembly JavaScriptインターフェース — JavaScriptからのコンパイルとインスタンス化、値の受け渡し、`Memory.grow`、3種類のエラークラス。
- WebAssembly Web API — `instantiateStreaming`、応答の検査、`application/wasm`の登録。
- Wasm 3.0の完成 — 2025年9月17日の発表と、3.0で追加された機能。
- WebAssemblyコア仕様1.0 — 2019年12月5日のW3C勧告。

用語集

実際の制約に合わせて選ぶ

業務の流れ、既存のインターフェース、変更を必要とする利用者をお知らせください。検証可能な納品内容と、お客様のチームが保守を続けられる引き継ぎ内容を定められます。

料金を見る お問い合わせ →

用語を実務に結び付ける

「WebAssemblyとは？」がお客様の組織にどう当てはまるかをご相談ください。現在のシステムの具体例が一つあれば、話を始められます。

サービスを見る

welcome@engineer.company +45 50 10 61 84 デンマーク、コペンハーゲン

このページを持ち歩く

連絡先カード

心を込めて丁寧に制作： D.

CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N

著作権 © 2025 – 現在 · Engineer ApS · すべての権利を留保します。

印刷元： https://engineer.company/ja/glossary/webassembly/