Capitolo 1

Introduzione a WebAssembly

Parlando di sviluppo web, l'aspetto che è al primo posto per la maggior parte degli sviluppatori web sono le prestazioni, dalla velocità di caricamento della pagina web alla reattività più in generale. Numerosi studi hanno dimostrato che se la vostra pagina web non si carica entro tre secondi, il 40% dei vostri visitatori se ne andrà altrove. Tale percentuale, poi, aumenta per ogni secondo in più necessario per caricare la pagina.

Ma il tempo necessario per caricare una pagina web non è l'unico problema. Secondo un articolo di Google, se una pagina web ha scarse prestazioni, il 79% dei visitatori afferma che è meno probabile che tornerà a fare acquisti su quel sito web (Daniel An e Pat Meenan, "Why marketers should care about mobile page speed" [luglio 2016], `http://mng.bz/MOlD`).

Con l'avanzamento delle tecnologie web, c'è stata una spinta a spostare sempre più applicazioni sul Web. Ciò ha costretto gli sviluppatori a fronteggiare un'altra sfida, perché i browser web supportano un solo linguaggio di programmazione: JavaScript.

Avere un unico linguaggio di programmazione fra tutti i browser è positivo, in un certo senso: basta scrivere il codice una sola volta e verrà eseguito su tutti i browser. Resta comunque da sottoporre a test la pagina in ogni browser che intendete supportare, perché i produttori a volte implementano le cose in modo leggermente differente. Inoltre, a volte un produttore di browser non aggiunge una nuova funzionalità contemporaneamente ad altri. Nel complesso, però, il fatto di dover supportare un solo linguaggio facilita le cose. Immaginate che cosa significherebbe averne quattro o cinque. Lo svantaggio dei browser

In questo capitolo

- **Che cos'è WebAssembly?**
- **Quali problemi risolve?**
- **Come funziona?**
- **Struttura di un modulo WebAssembly**
- **Il formato testuale di WebAssembly**
- **In che senso WebAssembly è sicuro?**
- **Quali linguaggi possiamo usare per creare un modulo WebAssembly?**
- **Dove possiamo usare il nostro modulo?**
- **Riepilogo**

che supportano solo JavaScript, tuttavia, è che le applicazioni che vogliamo trasferire sul Web non sono scritte in JavaScript, ma in linguaggi come il C++.
JavaScript è un ottimo linguaggio di programmazione, ma ora stiamo spingendo troppo oltre le nostre pretese, per esempio i pesanti calcoli dei giochi, e gli chiediamo di funzionare molto velocemente.

Che cos'è WebAssembly?

Mentre i produttori di browser cercavano nuovi modi per migliorare le prestazioni di JavaScript, Mozilla (il produttore del browser Firefox) ha definito un sottoinsieme di JavaScript chiamato asm.js.

Asm.js, il precursore di WebAssembly

Asm.js ha portato i seguenti vantaggi.

- *Non si programma direttamente in asm.js*. Piuttosto, si programma la logica in C o C++ e poi la si converte in JavaScript. La conversione del codice da un linguaggio a un altro è chiamata *transpiling*.
- *Esecuzione del codice più veloce per calcoli di un certo peso*. Quando l'engine JavaScript di un browser trova una speciale stringa, l'istruzione `asm pragma` (`"use asm ";`), la considera un flag, dicendo al browser che può utilizzare le operazioni di basso livello, invece delle più lente operazioni JavaScript.
- *Esecuzione del codice più veloce fin dalla prima chiamata*. I *type-hint* indicano al codice JavaScript quale tipo di dati conterrà una variabile. Per esempio, `a | 0` suggerisce che la variabile `a` conterrà un valore intero a 32 bit. La cosa funziona perché un'operazione di OR bit a bit con zero non cambia il valore originale, quindi non produce effetti collaterali.
- Questi type-hint servono come una promessa per l'engine JavaScript: se il codice dichiara una variabile come un numero intero, non la trasformerà mai in una stringa, per esempio. Di conseguenza, l'engine JavaScript non deve monitorare il codice per scoprire quali sono i tipi utilizzati. Può semplicemente compilare il codice così come è dichiarato.

Il seguente frammento di codice mostra un esempio di codice asm.js:

```
function AsmModule() {
    "use asm";  ❶
    return {
        add: function(a, b) {
            a = a | 0;  ❷
            b = b | 0;
            return (a + b) | 0;  ❸
        }
    }
}
```

❶ Flag che dice a JavaScript che il codice che segue è asm.js.

❷ Type-hint che indica che il parametro è un numero intero a 32 bit.

❸ Type-hint che indica che il valore restituito è un numero intero a 32 bit.

Nonostante i vantaggi, asm.js presenta ancora alcune carenze.

- Tutti questi type-hint possono far crescere le dimensioni dei file.
- Il file asm.js è un file JavaScript, quindi deve comunque essere letto e subire il parsing dell'engine JavaScript. Questo può diventare un problema su dispositivi come gli smartphone, perché tutta questa elaborazione rallenta il tempo di caricamento e consuma energia.
- Per aggiungere nuove funzionalità, i produttori di browser dovrebbero modificare lo stesso linguaggio JavaScript, il che non è desiderabile.
- JavaScript è un linguaggio di programmazione che non prevedeva compilazione.

Da asm.js a MVP

Nel tentativo di migliorare asm.js, i produttori di browser hanno escogitato un compromesso: MVP (*Minimum Viable Product*) WebAssembly, che mirava a conservare gli aspetti positivi di asm.js risolvendone le carenze. Nel 2017, i quattro principali produttori di browser (Google, Microsoft, Apple e Mozilla) hanno aggiornato i propri browser con il supporto per MVP, chiamato anche *Wasm*.

- WebAssembly è un linguaggio assembly-like di basso livello che può essere eseguito a velocità quasi native da tutti i più recenti browser desktop e da molti browser mobili.
- I file WebAssembly sono progettati per essere compatti e, di conseguenza, possono essere inviati e scaricati velocemente. I file sono inoltre progettati per un parsing e un'inizializzazione rapidi.
- WebAssembly è progettato per la compilazione, in modo che il codice scritto in linguaggi come C++, Rust e altri possa essere eseguito sul Web.

Gli sviluppatori di backend possono sfruttare WebAssembly per migliorare il riutilizzo del codice o per portare il proprio codice sul Web senza doverlo riscrivere. In più traggono un vantaggio anche dalla creazione di nuove librerie, dai miglioramenti alle librerie esistenti e dall'opportunità di migliorare le prestazioni in quelle sezioni del codice a elevata intensità di calcolo. Sebbene WebAssembly sia principalmente rivolto ai browser web, è progettato anche pensando alla portabilità, quindi potete usarlo anche al di là di un browser.

Quali problemi risolve?

Il WebAssembly MVP risolve i seguenti problemi di asm.js.

Miglioramenti prestazionali

Uno dei principali problemi che WebAssembly punta a risolvere sono le prestazioni, dal tempo necessario per scaricare il codice alla sua velocità di esecuzione. Con i linguaggi di programmazione, invece di programmare in linguaggio macchina che il microprocessore del computer è in grado di comprendere (fatto di uni e zeri, codice nativo), di solito scrivete un programma che è più vicino a una lingua umana. Sebbene sia più facile lavorare con un codice che rappresenti un'astrazione rispetto ai minuti dettagli del funzionamento del computer, i microprocessori non possono comprendere direttamente tale codice. Pertanto, quando arriva il momento di eseguirlo, è necessario convertire il programma in codice macchina.

JavaScript è un *linguaggio di programmazione interpretato*: legge il codice che avete e traduce al volo le sue istruzioni in codice macchina. Con i linguaggi interpretati, non è necessario attraversare una fase preliminare di compilazione del codice, il che significa anche che il codice inizia a "funzionare" prima. Lo svantaggio, tuttavia, è che l'interprete deve convertire le singole istruzioni in codice macchina, e questo ogni volta che il codice viene eseguito. Se il codice sta eseguendo un ciclo, per esempio, ogni riga di quel ciclo dovrà essere re-interpretata ogni volta che viene eseguito il ciclo. Poiché durante il processo di interpretazione non sempre c'è tempo per fare altro, non sempre le ottimizzazioni sono possibili.

Altri linguaggi di programmazione, come il C++, non sono interpretati. Con questi tipi di linguaggi, prima di eseguire un programma è necessario convertire in codice macchina le sue istruzioni, utilizzando particolari programmi chiamati *compilatori*. Con i linguaggi di programmazione compilati, ci vuole un po' di tempo per convertire in codice macchina le istruzioni prima di poterle eseguire, ma il vantaggio è che così c'è tutto il tempo per eseguire ogni ottimizzazione del codice; e poi, una volta compilato in codice macchina, il programma non deve essere nuovamente compilato.

Nel corso del tempo, JavaScript è passato dall'essere semplicemente un linguaggio nato per legare insieme i componenti, fatto solo per piccoli programmi, a un linguaggio utilizzato da molti siti web per eseguire elaborazioni anche complesse, con programmi di centinaia o migliaia di righe di codice; inoltre, con l'aumento delle applicazioni single-page, spesso tale codice è di lunga durata. Internet stesso è passato da siti web che mostravano solo del testo con alcune immagini a siti web altamente interattivi e addirittura a siti che funzionano da applicazioni Web, tanto sono simili alle applicazioni desktop, pur essendo eseguiti in un browser web.

Mentre gli sviluppatori continuavano a spingere sempre più in là i limiti di JavaScript, sono emersi alcuni evidenti problemi prestazionali. I produttori di browser hanno deciso di provare a trovare una via di mezzo per avere i vantaggi di un interprete, in cui il codice inizia a essere eseguito non appena viene richiamato, ma trovando anche il modo di eseguirlo più velocemente. Per rendere più veloce il codice, i produttori di browser hanno introdotto il concetto di *compilazione JIT* (*just-in-time*), in cui l'engine JavaScript monitora il codice mentre lo esegue; se una sezione di codice viene riutilizzata più volte, l'engine tenta di compilare quella sezione in codice macchina, in modo da poter bypassare l'engine JavaScript e utilizzare invece i metodi di basso livello del sistema, molto più veloci.

L'engine JavaScript deve monitorare il codice più volte prima di decidere di compilarlo in codice macchina, perché JavaScript è anche un linguaggio di programmazione di-

namico. In JavaScript, una variabile può contenere qualsiasi tipo di valore. Per esempio, inizialmente una variabile può contenere un numero intero, ma in seguito le può venire assegnata una stringa. Fino a quando il codice non viene eseguito più volte, il browser non sa cosa aspettarsi. Anche una volta compilato, il codice deve comunque essere monitorato, perché c'è la possibilità che qualcosa cambi, e così il codice compilato di quella sezione dovrà essere eliminato e così il processo ricomincerà.

Tempi di avvio più rapidi rispetto a JavaScript

Come asm.js, anche WebAssembly non è progettato per essere scritto a mano e non è pensato per essere letto da noi esseri umani. Quando il codice viene compilato in WebAssembly, il bytecode risultante viene rappresentato in formato binario, anziché testuale, cosa che riduce le dimensioni del file, consentendone un invio e un download più rapido. Il file binario è progettato in modo tale che la convalida del modulo possa essere effettuata in un unico passaggio. La struttura del file, inoltre, consente di compilare in parallelo più sezioni del file.

Implementando la compilazione JIT, i produttori di browser hanno fatto molti progressi nel migliorare le prestazioni di JavaScript. Ma l'engine JavaScript può compilare il codice JavaScript in codice macchina solo dopo che il codice stesso è stato monitorato più volte. Il codice WebAssembly, al contrario, impiega tipi statici, il che significa che il tipo dei valori delle variabili è noto in anticipo. Per questo motivo, il codice WebAssembly può essere compilato in codice macchina fin da subito, senza che sia necessario monitorarlo: i miglioramenti prestazionali sono efficaci fin dalla prima esecuzione del codice.

Dalla versione iniziale dell'MVP, i produttori di browser hanno trovato il modo di migliorare ulteriormente le prestazioni di WebAssembly. Uno di questi miglioramenti è stata l'introduzione della *streaming compilation*: un processo che compila il codice WebAssembly in codice macchina mentre il file viene scaricato e ricevuto dal browser. La streaming compilation consente l'inizializzazione di un modulo WebAssembly non appena termina il download, cosa che accelera notevolmente il tempo di avvio del modulo.

Possibilità di utilizzare linguaggi diversi da JavaScript nel browser

Finora, per poter utilizzare nel Web un linguaggio diverso da JavaScript, tale codice doveva essere convertito in JavaScript, un linguaggio che non è un output standard di un compilatore. WebAssembly, al contrario, è stato progettato proprio per essere un output di un compilatore, quindi gli sviluppatori che desiderano utilizzare un determinato linguaggio per sviluppare per il Web potranno farlo senza dover trasporre il loro codice in JavaScript. Poiché WebAssembly non è legato al linguaggio JavaScript, diventa possibile apportare miglioramenti alla sua tecnologia molto più facilmente e senza preoccuparsi di interferire con il funzionamento di JavaScript. Questa indipendenza fa sì che WebAssembly possa essere migliorato molto più velocemente.

I linguaggi C e C++ sono già stati focalizzati come linguaggi che possono avere come target codice WebAssembly, ma anche Rust ha aggiunto il supporto per WebAssembly e molti altri linguaggi stanno procedendo in questo senso.

Opportunità di riutilizzo del codice

Essere in grado di prendere del codice scritto in linguaggi diversi da JavaScript e compilarlo in WebAssembly offre agli sviluppatori una maggiore flessibilità, in termini di riutilizzo del codice. Ora, un programma che un tempo era condannato a essere riscritto in JavaScript può essere utilizzato su desktop o su server e anche essere eseguito in un browser.

Come funziona?

Come vediamo nella Figura 1.1, con JavaScript il codice è incluso nel sito web e viene interpretato durante l'esecuzione della pagina. Poiché le variabili JavaScript sono dinamiche, osservando la funzione `add` dell'illustrazione, non possiamo sapere con quale tipo di valori abbiamo a che fare. Le variabili `a` e `b` potrebbero essere numeri interi, in virgola mobile, stringhe o anche una combinazione di essi: una variabile potrebbe essere una stringa e l'altra un numero in virgola mobile, per esempio.
L'unico modo per conoscere con certezza il tipo delle variabili è monitorare il codice durante l'esecuzione, che è esattamente ciò che fa l'engine JavaScript. Una volta che l'engine scopre il tipo delle variabili, può convertire quella sezione di codice in codice macchina.

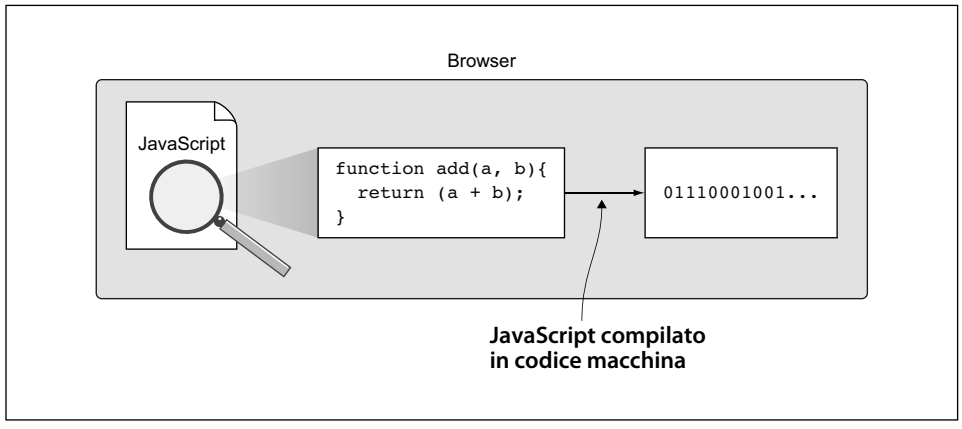

Figura 1.1 JavaScript compilato in codice macchina durante l'esecuzione.

WebAssembly non viene interpretato: viene compilato in formato binario WebAssembly da uno sviluppatore, come vediamo nella Figura 1.2. Poiché il tipo di tutte le variabili è noto in anticipo, quando il browser carica il file WebAssembly, l'engine JavaScript non ha bisogno di monitorare il codice. Può semplicemente compilare il formato binario del codice in codice macchina.

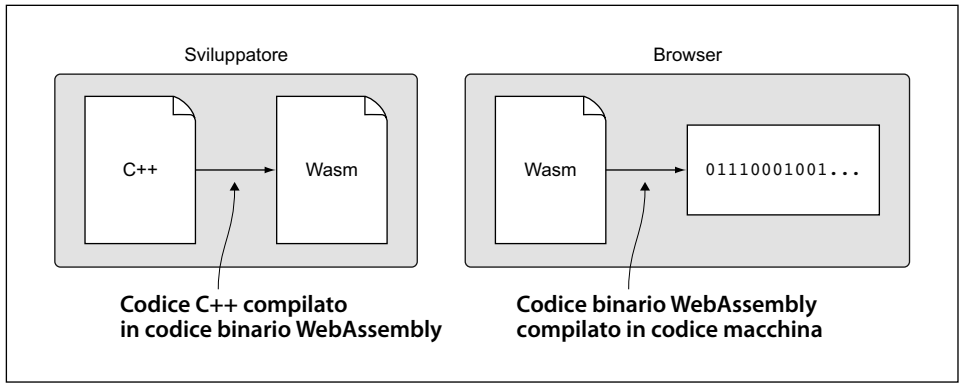

Figura 1.2 Il codice C++ viene trasformato prima in WebAssembly e poi, nel browser, in codice macchina.

Come funzionano i compilatori

Abbiamo già accennato al fatto che gli sviluppatori programmano in un linguaggio più vicino a quello umano, ma i microprocessori comprendono solo il linguaggio macchina. Di conseguenza, il codice del programma deve essere convertito in *codice macchina* per poter essere eseguito. Quello che non ho menzionato è che ogni tipo di microprocessore impiega il proprio tipo di codice macchina.

Sarebbe inefficiente compilare ogni programma scritto in ogni linguaggio di programmazione nel codice macchina di ogni microprocessore. Quello che accade di solito è rappresentato nella Figura 1.3: una parte del compilatore, il *frontend*, converte il codice del programma in una rappresentazione intermedia (IR). Una volta creato il codice intermedio, la parte *backend* del compilatore prende questo codice, lo ottimizza e lo trasforma nel codice macchina desiderato.

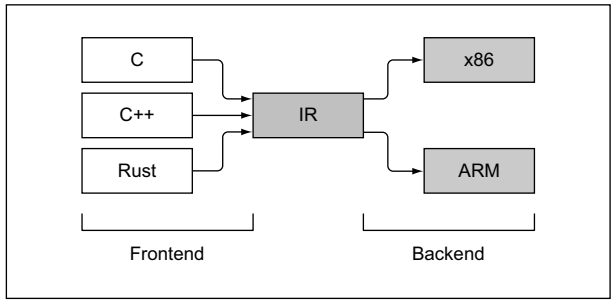

Figura 1.3 Il frontend e il backend del compilatore.

Poiché un browser può essere eseguito su diversi microprocessori (per computer desk top, smartphone e tablet, per esempio), distribuire una versione compilata del codice WebAssembly per ogni potenziale microprocessore sarebbe un problema. La Figura 1.4 mostra invece cosa possiamo fare: possiamo prendere il codice intermedio IR ed eseguirlo attraverso uno speciale compilatore che lo converte in uno speciale bytecode binario e inserisce quel bytecode in un file con estensione `.wasm`.

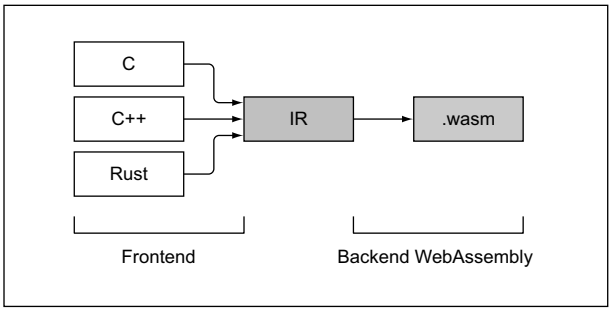

Figura 1.4 Il frontend del compilatore con un backend WebAssembly.

Il bytecode del file Wasm non è ancora codice macchina. È semplicemente un insieme di istruzioni virtuali che è compreso dai browser che supportano WebAssembly. Come vediamo nella Figura 1.5, quando il file viene caricato in un browser che supporta Web Assembly, il browser verifica che il codice sia valido; il bytecode viene poi finalmente compilato nel codice macchina del dispositivo su cui è in esecuzione il browser.

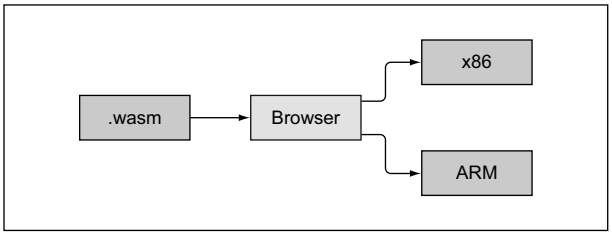

Figura 1.5 File Wasm caricato in un browser e poi compilato in codice macchina.

Caricamento, compilazione e istanziazione di un modulo

Al momento attuale, il processo di download del file Wasm nel browser e poi di compilazione da parte del browser viene eseguito utilizzando chiamate a funzioni JavaScript. C'è l'intenzione di consentire ai moduli WebAssembly di interagire con i moduli *ES6*, in futuro, il che includerebbe la possibilità di caricare i moduli WebAssembly tramite uno speciale tag HTML (`<script type="module">`), che tuttavia non è ancora disponibile. A questo proposito, *ES* è l'abbreviazione di *ECMAScript* (il nome ufficiale di JavaScript) e *6* è la sua versione.

Prima che il bytecode binario del modulo WebAssembly possa essere compilato, è necessario convalidarlo per assicurarsi che il modulo sia strutturato correttamente, che il codice non possa fare nulla di vietato e che non possa accedere a quelle aree di memoria cui non ha accesso. Vengono inoltre effettuati controlli a runtime, per garantire che il codice rimanga all'interno della memoria cui ha accesso. Il file Wasm è strutturato in modo tale che la convalida possa essere eseguita in un unico passaggio, per garantire che i processi di convalida, compilazione in codice macchina e istanziazione avvengano il più rapidamente possibile.

Una volta che il browser ha compilato il bytecode WebAssembly in codice macchina, il modulo compilato può essere passato a un web worker (approfondiremo l'argomento nel Capitolo 9; i web worker sono un modo per creare thread in JavaScript) o a un'altra finestra del browser. Il modulo compilato può anche essere utilizzato per creare nuove istanze del modulo.
Una volta che un file Wasm è stato compilato, deve essere istanziato prima di poter essere utilizzato. L'*istanziazione* è semplicemente il processo di ricezione di tutti gli oggetti importati necessari, l'inizializzazione degli elementi del modulo, la chiamata alla funzione di avvio, se è stata definita, e infine la restituzione dell'istanza del modulo all'ambiente di esecuzione.

WebAssembly e JavaScript

Finora, l'unico linguaggio eseguibile all'interno della macchina virtuale (VM) JavaScript era JavaScript. Quando, nel corso degli anni, sono state provate altre tecnologie, come i *plug-in*, è stato necessario creare una VM a *sandbox*, che ha aumentato sia la superficie di attacco sia il consumo di risorse del computer. Per la prima volta in assoluto, la macchina virtuale JavaScript viene aperta, per consentire l'esecuzione nella stessa macchina virtuale anche del codice WebAssembly. Questo presenta diversi vantaggi. Uno dei principali è che nel corso degli anni la macchina virtuale è stata ampiamente sottoposta a test e rafforzata contro le vulnerabilità. Se al suo posto fosse stata creata una nuova macchina virtuale, avrebbe senza dubbio avuto dei problemi di sicurezza da sistemare. WebAssembly è stato progettato come complemento a JavaScript, non come un sostituto. Anche se probabilmente alcuni sviluppatori proveranno a creare interi siti web utilizzando solo WebAssembly, difficilmente questa sarà la norma. Ci saranno situazioni in cui JavaScript sarà ancora la scelta migliore. Ci saranno anche situazioni in cui un sito web potrebbe dover includere WebAssembly per eseguire i calcoli più rapidamente o per un supporto di basso livello. Per esempio, la tecnica SIMD (*Single Instruction, Multiple Data*), ovvero la capacità di elaborare più dati con una singola istruzione, è stata incorporata nel JavaScript di diversi browser, ma i produttori di browser hanno scelto di deprecare l'implementazione JavaScript e rendere disponibile il supporto SIMD solo tramite moduli WebAssembly. Di conseguenza, se il vostro sito web necessita del supporto SIMD, dovrete includere un modulo WebAssembly con il quale comunicare.

Quando si programma per l'esecuzione in un browser web, vi sono fondamentalmente due componenti principali: la VM JavaScript, in cui viene eseguito il modulo WebAssembly, e le API web (per esempio, DOM, WebGL, i web worker e così via). Essendo WebAssembly solo un MVP (*Minimum Viable Product*), gli mancano alcune cose. Il vostro modulo WebAssembly può comunicare con JavaScript, ma non è ancora in grado di comunicare direttamente con nessuna API web. È in corso di preparazione una funzionalità post-MVP che consentirà a WebAssembly di accedere direttamente alle API web. Nel frattempo, i moduli possono interagire con le API web indirettamente, richiamando JavaScript e facendogli eseguire l'azione richiesta, per conto del modulo.

Struttura di un modulo WebAssembly

Attualmente WebAssembly offre solo quattro tipi di valori:

- valori interi a 32 bit;
- valori interi a 64 bit;
- valori in virgola mobile a 32 bit;
- valori in virgola mobile a 64 bit.

I valori booleani sono rappresentati tramite un numero intero a 32 bit, dove `0` è `false` e un qualsiasi valore diverso da `0` è `true`. Tutti gli altri tipi di valori, come le stringhe, devono essere rappresentati nella memoria lineare del modulo.

L'unità principale di un programma WebAssembly è il *modulo*, termine utilizzato sia per la versione binaria del codice sia per la versione compilata nel browser. Un modulo WebAssembly non è qualcosa che si debba creare a mano, ma è comunque importante sapere come è strutturato e come funziona, una conoscenza che può tornare utile quando si interagisce con alcuni aspetti del modulo durante l'inizializzazione e per tutta la sua vita. La Figura 1.6 mostra una rappresentazione di base della struttura di un file WebAssembly. Esplorerete in modo più dettagliato la struttura di un modulo nel Capitolo 2, ma, per ora, questa è solo una rapida panoramica.

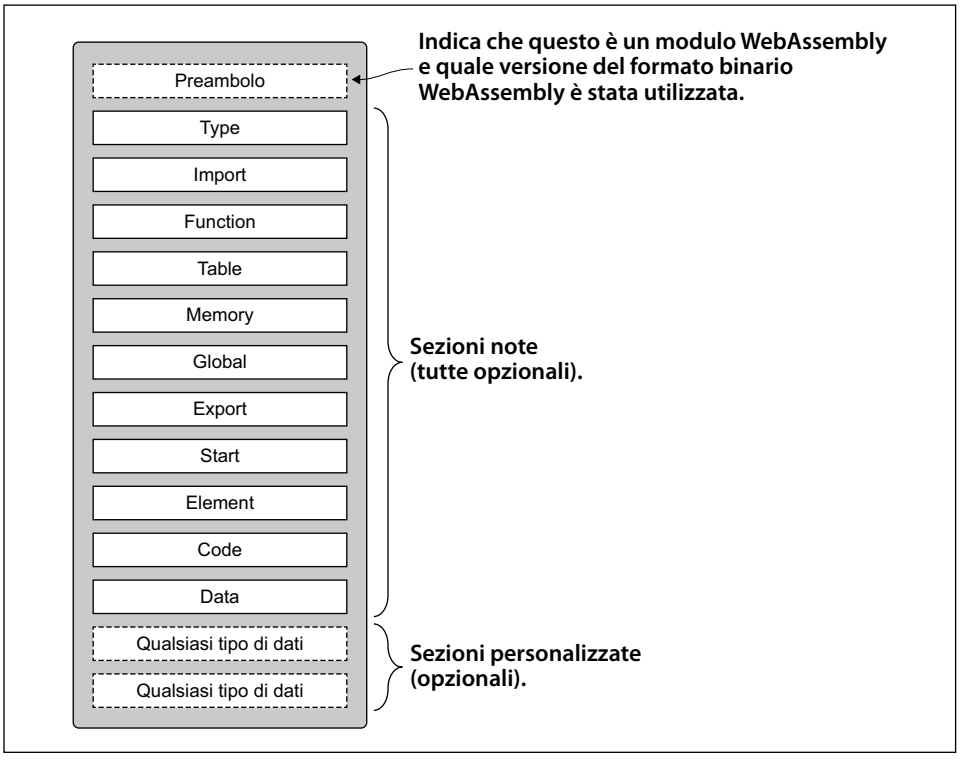

Figura 1.6 Una rappresentazione generale della struttura di un file WebAssembly.

Un file Wasm inizia con una sezione chiamata preambolo.

Preambolo

Il *preambolo* contiene un numero magico (`0x00 0x61 0x73 0x6D`, ovvero `\0asm`) che distingue un modulo WebAssembly da un modulo ES6. Questo numero magico è seguito da una versione (`0x01 0x00 0x00 0x00`, ovvero `1`) che indica quale versione del formato binario WebAssembly è stata utilizzata per creare il file.
Al momento esiste una sola versione del formato binario. Uno degli obiettivi di Web Assembly è mantenere tutto compatibile con le versioni precedenti quando vengono aggiunte nuove funzionalità, ed evitare di dover incrementare il numero di versione. Se mai si presentasse una funzionalità che non può essere implementata senza modificare le cose, il numero di versione verrà incrementato.
Dopo il preambolo, un modulo può avere più sezioni, tutte opzionali. Quindi, teoricamente, potreste avere un modulo vuoto senza sezioni. Scoprirete un caso d'uso per un modulo vuoto nel Capitolo 3, quando implementerete il rilevamento delle funzionalità per verificare se WebAssembly è supportato in un browser web.
Sono disponibili due tipi di sezioni: sezioni note e sezioni personalizzate.

Sezioni note

Le sezioni note possono essere incluse una sola volta e devono apparire in un ordine ben preciso. Ogni sezione nota ha uno scopo ben determinato, è ben definita e viene convalidata quando il modulo viene istanziato. Il Capitolo 2 approfondisce l'argomento delle sezioni note.

Sezioni personalizzate

Una sezione personalizzata fornisce un modo per includere i dati all'interno del modulo per usi che non valgono per le sezioni note. Le sezioni personalizzate possono apparire ovunque nel modulo (prima, fra o dopo le sezioni note), un numero qualsiasi di volte e più sezioni personalizzate possono perfino riutilizzare lo stesso nome.
A differenza delle sezioni note, se una sezione personalizzata non è disposta correttamente, ciò non causerà un errore di convalida. Le sezioni personalizzate possono essere caricate dal framework in modo "pigro" (*lazy*), il che significa che i dati che contengono potrebbero essere disponibili solo a un certo punto, dopo l'inizializzazione del modulo.
Per il WebAssembly MVP è stata definita una sezione personalizzata denominata "Name". Lo scopo di questa sezione: potreste avere una versione di debugging del vostro modulo WebAssembly, e questa sezione conterrebbe i nomi delle funzioni e delle variabili sotto forma di testo, da utilizzare durante il debugging. A differenza di altre sezioni personalizzate, questa sezione deve apparire una sola volta e solo dopo la sezione Data.

Il formato testuale di WebAssembly

WebAssembly è stato progettato pensando all'apertura del Web. Solo perché il formato binario non è progettato per essere scritto o letto da esseri umani, questo non significa che i moduli WebAssembly consentano agli sviluppatori di provare a nascondere il pro-

prio codice. In realtà, è vero proprio il contrario. Per WebAssembly è stato definito un formato testuale che utilizza *s-expression* e che corrisponde al formato binario.

> **NOTA**
>
> Le espressioni simboliche, o s-expression, sono state inventate per il linguaggio di programmazione Lisp. Una s-expression può essere un atomo o una coppia ordinata di s-expression che consentono di annidare più s-expression. Un atomo è un simbolo che non sia una lista: `pippo` o `23`, per esempio. Una lista è rappresentata da una coppia di parentesi e può essere vuota o contenere atomi o altre liste; ogni elemento è delimitato da spazi: `()` o `(pippo)` o `(pippo (bar 132))`, per esempio.

Questo formato testuale abilita il comando Visualizza sorgente per mostrare il codice nel browser, per esempio, oppure può essere utilizzato per il debugging. Potete perfino scrivere s-expression a mano e, utilizzando uno speciale compilatore, compilare il codice nel formato binario WebAssembly.
Poiché il formato testuale WebAssembly verrà utilizzato dai browser quando si sceglie di visualizzare il codice sorgente e per scopi di debugging, sarà utile conoscerlo, almeno un po'. Per esempio, poiché tutte le sezioni di un modulo sono opzionali, potete definire un modulo vuoto usando la seguente s-expression:

```
(module)
```

Se doveste compilare la s-expression `(module)` nel formato binario WebAssembly e osservare i valori binari risultanti, il file conterrebbe solo i byte di preambolo: `0061 736d` (il numero magico) e `0100 0000` (il numero di versione).

> **RIFERIMENTO**
>
> Nel Capitolo 11, creerete un modulo WebAssembly usando il solo formato testuale, in modo da farvi un'idea di quello che state guardando, se doveste mai eseguire il debugging di un modulo in un browser.

In che senso WebAssembly è sicuro?

WebAssembly è il primo linguaggio in assoluto a condividere la macchina virtuale JavaScript, che è in sandbox rispetto al runtime e vanta anni di rafforzamento e test di sicurezza. I moduli WebAssembly non hanno accesso a nulla a cui non abbia accesso JavaScript e rispettano anche le stesse politiche di sicurezza, che includono l'applicazione di obblighi come la politica della stessa origine.
A differenza di un'applicazione desktop, un modulo WebAssembly non ha alcun accesso diretto alla memoria del dispositivo. Al contrario, durante l'inizializzazione l'ambiente runtime passa al modulo un `ArrayBuffer`. Il modulo utilizza questo `ArrayBuffer` come memoria lineare e il framework WebAssembly verifica che il codice funzioni entro i limiti di tale array.
Un modulo WebAssembly non ha alcun accesso diretto agli elementi, come i puntatori a funzione, che sono archiviati nella sezione Table. Il codice chiede al framework Web

Assembly di accedere a un elemento in base al suo indice. Il framework, a sua volta, accede alla memoria ed esegue l'elemento per conto del codice.
In C++, lo stack di esecuzione è in memoria insieme alla memoria lineare e, sebbene il codice C++ non debba modificare lo stack di esecuzione, è possibile farlo utilizzando i puntatori. Anche lo stack di esecuzione di WebAssembly è separato dalla memoria lineare e non è accessibile dal codice.

APPROFONDIMENTO

Per maggiori informazioni sul modello di sicurezza di WebAssembly, potete visitare il seguente sito: `https://webassembly.org/docs/security`.

Quali linguaggi possiamo usare per creare un modulo WebAssembly?

L'obiettivo iniziale di WebAssembly era di considerare i linguaggi C e C++, ma da allora anche linguaggi come Rust e AssemblyScript hanno aggiunto il loro supporto. È anche possibile scrivere codice utilizzando il formato testuale WebAssembly, che utilizza s-expression, e poi compilarlo in WebAssembly utilizzando uno speciale compilatore.
In questo momento, l'MVP di WebAssembly non dispone di *garbage collection*, un aspetto che limita ciò che alcuni linguaggi possono fare. Se ne sta lavorando come funzionalità post-MVP, ma, finché non arriverà, diversi linguaggi stanno sperimentando con Web Assembly compilando la propria VM in WebAssembly o, in alcuni casi, includendo il proprio garbage collector.
I seguenti linguaggi stanno sperimentando o hanno il supporto di WebAssembly.

- C e C++.
- Rust punta a diventare il linguaggio di programmazione preferito per WebAssembly.
- AssemblyScript è un nuovo compilatore che prende TypeScript e lo trasforma in WebAssembly. La conversione di TypeScript ha perfettamente senso, considerando che è un linguaggio tipizzato e dotato di transpilazione in JavaScript.
- TeaVM è uno strumento che trasferisce Java in JavaScript, ma ora può anche generare WebAssembly.
- Go 1.11 ha aggiunto un porting sperimentale in WebAssembly che include un garbage collector come parte del modulo WebAssembly compilato.
- Pyodide è un porting di Python che include i pacchetti principali del suo stack scientifico: Numpy, Pandas e matplotlib.
- Blazor è uno sforzo sperimentale di Microsoft per portare C# in WebAssembly.

NOTA

Il seguente repository GitHub gestisce un elenco curato di linguaggi che sono compilabili o hanno le loro macchine virtuali in WebAssembly. L'elenco indica anche come si colloca il linguaggio in termini di supporto a WebAssembly: `https://github.com/appcypher/awesome-wasm-langs`.

Per studiare WebAssembly in questo libro, useremo i linguaggi C e C++.

Dove possiamo usare il nostro modulo?

Nel 2017, tutti i produttori di browser hanno rilasciato nuove versioni dei loro browser che supportano l'MVP di WebAssembly; fra questi Chrome, Edge, Firefox, Opera e Safari. Anche diversi browser web mobili supportano anche WebAssembly, inclusi Chrome e Firefox per Android e Safari.

Come ho accennato all'inizio del capitolo, WebAssembly è stato progettato pensando alla portabilità, in modo da poter essere utilizzato in più situazioni, non solo in un browser. È in fase di sviluppo un nuovo standard, WASI (*WebAssembly Standard Interface*) per garantire che i moduli WebAssembly funzionino in modo coerente su tutti i sistemi supportati. Il seguente articolo offre una buona panoramica su WASI: Lin Clark, "Standardizing WASI: A system interface to run WebAssembly outside the web" (27 marzo 2019), `http://mng.bz/gVJ8`.

> **APPROFONDIMENTO**
>
> Per maggiori informazioni su WASI, il seguente repository GitHub offre un elenco curato di link e articoli correlati: `https://github.com/wasmerio/awesome-wasi`.

Una situazione non-browser che supporta i moduli WebAssembly è Node.js, a partire dalla versione 8. Node.js è un runtime JavaScript creato utilizzando l'engine JavaScript V8 di Chrome che consente di utilizzare il codice JavaScript server-side. Un po' come gli sviluppatori vedono WebAssembly come un'opportunità per utilizzare nel browser il codice con cui hanno più familiarità, anziché JavaScript, Node.js consente agli sviluppatori che preferiscono JavaScript di utilizzarlo anche sul lato server. Per illustrare l'utilizzo di WebAssembly anche al di fuori di un browser, il Capitolo 10 vi mostrerà come lavorare con un modulo WebAssembly in Node.js.

WebAssembly non è un sostituto, ma piuttosto un complemento di JavaScript. Ci sono situazioni in cui sarà preferibile usare un modulo WebAssembly e altri in cui sarà meglio utilizzare JavaScript. L'esecuzione nella stessa macchina virtuale di JavaScript consente a entrambe le tecnologie di sfruttarsi a vicenda.

WebAssembly consente agli sviluppatori esperti in linguaggi diversi da JavaScript di rendere disponibile il loro codice sul Web. Consente inoltre agli sviluppatori web che non sanno programmare in linguaggi come il C o il C++ di accedere a librerie più aggiornate, più veloci e, potenzialmente, più ricche di funzionalità rispetto alle attuali librerie JavaScript. In alcuni casi, i moduli WebAssembly potrebbero essere utilizzati da una libreria per velocizzare l'esecuzione di alcuni aspetti della libreria stessa; oltre ad avere un codice più veloce, la libreria funzionerebbe come sempre.

La cosa più interessante di WebAssembly è che è già disponibile in tutti i principali browser desktop, in molti dei principali browser mobili e anche al di fuori del browser in Node.js.

Riepilogo

WebAssembly apporta tutta una serie di miglioramenti prestazionali, nonché miglioramenti nella scelta del linguaggio e nel riutilizzo del codice.

- I tempi di trasmissione e download sono più rapidi, grazie alle dimensioni più compatte dei file, merito della codifica binaria.

- Grazie al modo in cui sono strutturati i file Wasm, il parsing e la convalida possono essere eseguiti rapidamente. Per il modo in cui sono strutturati, le varie porzioni dei file possono essere compilate in parallelo.
- Con la *streaming compilation*, i moduli WebAssembly possono essere compilati mentre vengono scaricati, in modo che siano pronti per essere istanziati al termine del download, cosa che accelera notevolmente il tempo di caricamento.
- L'esecuzione del codice è più veloce per attività come i calcoli grazie all'uso di chiamate a livello macchina al posto delle più costose chiamate dell'engine JavaScript.
- Non è necessario monitorare il codice prima di compilarlo, per determinare come si comporterà. Il risultato è che il codice viene eseguito ogni volta alla stessa velocità.
- Essendo separato da JavaScript, è possibile apportare miglioramenti a WebAssembly più velocemente, perché essi non influiranno sul linguaggio JavaScript.
- Diventa possibile utilizzare in un browser codice scritto in un linguaggio diverso da JavaScript.
- Esiste una maggiore opportunità per il riutilizzo del codice, strutturando il framework WebAssembly in modo tale che possa essere utilizzato sia nel browser sia indipendentemente dal browser.