УДК 004.582

WEBASSEMBLY – ПУТЬ К НОВЫМ ГОРИЗОНТАМ ПРОИЗВОДИТЕЛЬНОСТИ В WEB

М.А. БАЛАБАШ
(Представлено: Д.В. ПЯТКИН)

Рассматривается технология WebAssembly – эффективный низкоуровневый байт-код, предназначенный для исполнения в браузере. Для компиляции кода в формат WebAssembly используется Emscripten, который позволяет достичь высокой производительности.

Технология WebAssembly – это инициатива, направленная на создание безопасного, переносимого и быстрого для загрузки и исполнения формата кода, подходящего для Web. WebAssembly – цель компиляции, у которой имеются спецификации текстового и бинарного форматов. Это означает, что другие низкоуровневые языки, такие как C/C++, Rust, Swift, и так далее, можно скомпилировать в WebAssembly. WebAssembly дает доступ к тем же API, что и браузерный JavaScript, органично встраивается в существующий стек технологий. Это отличает wasm от чего-то вроде Java-апплетов. Архитектура WebAssembly – это результат коллективной работы сообщества, в котором имеются представители разработчиков всех ведущих веб-браузеров. Для компиляции кода в формат WebAssembly используется Emscripten.

Основная часть. Emscripten – это компилятор из байт-кода LLVM в JavaScript. То есть с его помощью можно скомпилировать в JavaScript программы, написанные на C/C++ или на любых других языках, код на которых можно преобразовать в формат LLVM. Emscripten предоставляет набор API для портирования кода в формат, подходящий для веб. Этому проекту уже много лет, в основном его используют для преобразования игр в их браузерные варианты. Emscripten позволяет достичь высокой производительности благодаря тому, что он генерирует код, соответствующий стандартам Asm.js, о котором ниже, но недавно его успешно оснастили поддержкой WebAssembly.

Asm.js – это низкоуровневое оптимизированное подмножество JavaScript, обеспечивающее линейный доступ к памяти с помощью типизированных массивов и поддерживающее аннотации с информацией о типах данных. Asm.js позволяет повысить производительность решений. Это – тоже не новый язык программирования, поэтому, если браузер его не поддерживает, Asm.js-код будет выполняться как обычный JavaScript, то есть, от его использования не удастся получить прироста производительности.

WebAssembly, по состоянию на 10.01.2017, поддерживается в Chrome Canary и Firefox. Для того чтобы wasm-код заработал, нужно активировать соответствующую возможность в настройках.

В Safari поддержка WebAssembly пока в стадии разработки. В V8 wasm включен по умолчанию. Вот интересное видео о движке V8, о текущем состоянии поддержки JavaScript и WebAssembly с Chrome Dev Summit 2016.

WebAssembly это:

- *Улучшение JavaScript*: Реализуйте все критичные вещи на wasm и импортируйте его как стандартный JavaScript модуль.
- *Новый язык программирования*: WebAssembly определяет абстрактное синтаксическое дерево (как и JavaScript) в бинарном формате. Вы можете писать код и чистить его от ошибок в текстовом формате. WebAssembly легко читаем.
- *Улучшение для браузеров*: Браузеры будут понимать бинарный формат, а это значит, что разработчики смогут компилировать бинарники, которые можно сжать гораздо больше, чем используемые сегодня текстовые файлы с JavaScript. Чем меньше файл, тем быстрее загрузка. В зависимости от возможностей оптимизации времени компиляции, код на WebAssembly может передаваться и запускаться быстрее, чем на JavaScript!
- *Целевая компиляция*: Возможность другим языкам, получить первоклассную двоичную поддержку через весь стек веб-платформы.

WebAssembly добавляет вещи, которые большинство JS разработчиков не хотят видеть в JavaScript. Сама функциональность нужна, но вот в JavaScript ей места точно нет. Тем более что мы можем получить все эти функции с помощью компиляции с других языков программирования.

Фактически, WebAssembly предоставляет нам альтернативный компилятор – созданный специально для этих целей.

Теперь нам будет гораздо легче портировать код, который сильно зависит от, например, совместно используемых цепочек памяти. Представляется, что написать компилятор для WebAssembly будет легче, чем написать компилятор для JavaScript, а все потому, что первый гарантирует лучший перенос функций языка в заданное абстрактное синтаксическое дерево.

Среди всего прочего, его можно будет использовать для простой работы с тредами и SIMD (single instruction, multiple data) – проще говоря, с одним потоком команд и несколькими потоками данных.

Вы можете поставить в очередь множество блоков данных, а затем прописать одну команду для одновременной работы с ними. Это значит, что параллельная обработка потокового видео будет обрабатываться процессором. Реализуем функцию умножения на два на языке программирования C.

Листинг 1 – Реализация функции умножения на два

```
int doubler(int x) {
return 2 * x;
}
```

Займемся преобразованием программы, написанной на C, в формат wasm. Для того чтобы это сделать, решено воспользоваться возможностью создания автономных модулей WebAssembly. При таком подходе на выходе компилятора мы получаем только файл с кодом WebAssembly, без дополнительных вспомогательных .js-файлов.

Такой подход основан на концепции дополнительных модулей (side module) Emscripten. Здесь имеет смысл использовать подобные модули, так как они, в сущности, очень похожи на динамические библиотеки. Например, системные библиотеки не подключаются к ним автоматически, они представляют собой некие самодостаточные блоки кода, выдаваемого компилятором.

Листинг 2 – Преобразованием программы, написанной на C, в формат wasm.

```
emcc hello_world.c -Os -s WASM=1 -s SIDE_MODULE=1 -o hello_world.c
```

После получения бинарного файла нам нужно лишь загрузить его в браузер. Для того чтобы это сделать, API WebAssembly предоставляет объект верхнего уровня WebAssembly, который содержит методы, нужные для того, чтобы скомпилировать и создать экземпляр модуля.

Листинг 3 – универсальный загрузчик модулей

```
function loadWebAssembly(filename, imports) {
  return fetch(filename)
    .then(response => response.arrayBuffer())
    .then(buffer => WebAssembly.compile(buffer))
    .then(module => {
      imports = imports || {};
      imports.env = imports.env || {};
      imports.env.memoryBase = imports.env.memoryBase || 0;
      imports.env.tableBase = imports.env.tableBase || 0;
      if (!imports.env.memory) {
        imports.env.memory = new WebAssembly.Memory({ initial: 256 });
      }
      if (!imports.env.table) {
        imports.env.table = new WebAssembly.Table({ initial: 0, element: 'anyfunc' });
      }
      return new WebAssembly.Instance(module, imports);
    });
}
```

Сначала мы берем содержимое файла и конвертируем его в структуру данных формата ArrayBuffer. Буфер содержит исходные двоичные данные фиксированной длины. Напрямую исполнять их мы не можем, именно поэтому на следующем шаге буфер передают методу WebAssembly.compile, который возвращает WebAssembly.Module, экземпляр которого, в итоге, можно создать с помощью WebAssembly.Instance.

Листинг 4 – Пример страницы с загрузкой и исполнением модуля

```
<html>
<head>
  <script>
    if (!('WebAssembly' in window)) {
      alert('you need a browser with wasm support enabled :(');
    }

    function loadWebAssembly(filename, imports) {
```

```
  return fetch(filename)
    .then(response => response.arrayBuffer())
    .then(buffer => WebAssembly.compile(buffer))
    .then(module => {
      imports = imports || {};
      imports.env = imports.env || {};
      imports.env.memoryBase = imports.env.memoryBase || 0;
      imports.env.tableBase = imports.env.tableBase || 0;
      if (!imports.env.memory) {
        imports.env.memory = new WebAssembly.Memory({ initial: 256 });
      }
      if (!imports.env.table) {
        imports.env.table = new WebAssembly.Table({ initial: 0, element: 'anyfunc' });
      }
      return new WebAssembly.Instance(module, imports);
    });
}


loadWebAssembly('hello_world.wasm')
  .then(instance => {
    var exports = instance.exports;
    var doubler = exports._doubler;

    var button = document.getElementById('run');
    button.value = 'Call a method in the WebAssembly module';
    button.addEventListener('click', function() {
      var input = 21;
      alert(input + ' doubled is ' + doubler(input));
    }, false);
  }
 );
 </script>
</head>
<body>
 <input type="button" id="run" value="(waiting for WebAssembly)"/>
</body>
</html>
```

Выполнение WebAssembly занимает меньше времени, так как можно обойтись без хитростей и приемов, которые должен использовать разработчик для повышения производительности JavaScript. Кроме того, бинарный формат WebAssembly значительно ближе к машинному коду. Также в WebAssembly не требуется применение сборщика мусора, так как применяется явное управление памятью.

Заключение

Более компактное представление WebAssembly позволяет сократить время загрузки по сравнению с загрузкой даже сжатого JavaScript. Декодирование WebAssembly занимает значительно меньше времени по сравнению с парсингом исходных текстов JavaScript. Компиляция и оптимизация выполняются быстрее, так как WebAssembly ближе к машинному коду и уже прошел стадии оптимизации на этапе компиляции разработчиком. Не требуется выполнение операции повторной оптимизации, учитывающей статистику о переменных, полученную при выполнении приложения, так как в WebAssembly изначально присутствует информация о типах, которую JavaScript вынужден вычислять на ходу в зависимости от контекста. Все это делает будущее веб-технологий более производительным и еще более функциональным.

ЛИТЕРАТУРА

1. Собираем ваш первый WebAssembly-компонент [Электронный ресурс] / Хабрахабр. – Режим доступа: https://habrahabr.ru/company/infopulse/blog/304362/. – Дата доступа: 27.09.17.
2. WebAssembly [Электронный ресурс] // Wikipedia. – Режим доступа: https://ru.wikipedia.org/wiki/WebAssembly. – Дата доступа: 27.09.2017.
3. WebAssembly [Электронный ресурс] / DeveloperMozilla. – Режим доступа: https://developer.mozilla.org/en-US/docs/WebAssembly. – Дата доступа: 27.09.2017.