

Re-Assembling the Web

Using Web Assembly - The New Way of Writing Fast Web Code

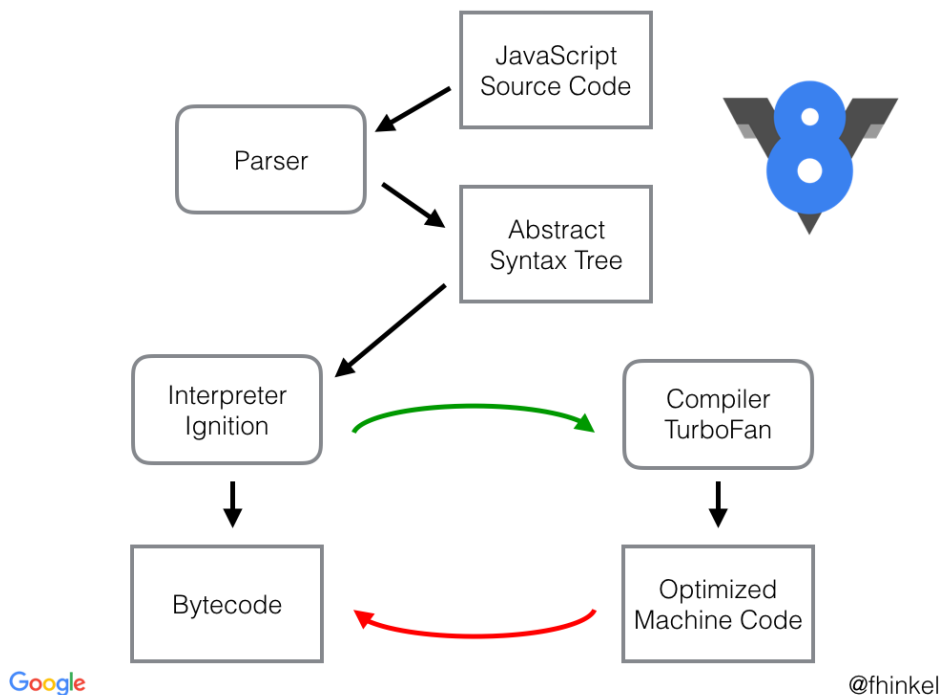
מאת רון דלאל

הקדמה

WebAssembly (או בקיצור - WASM) היא דרך חדשה להריץ קוד על ה-Browser שלכם, בצורה מהירה. כותבים בשפה כמו C/C++/Rust, מקמפלים את הקוד ל-WebAssembly, והוא רץ על הדפדפן שלכם, כחלק מהקוד של האתר. כדי להבין איך היא עובדת, ואת היתרונות שלה, נצלול קודם לסיפור מקדים על איך בכלל Browser Engines עובדים.

מבוא ל-Browser Engines

איך הקוד שלכם רץ על הדפדפן? נדגים את זה על Chrome:



Parser

לפני שנדבר על תהליך הקומפילציה של Javascript עוברת, נדבר מעבר על מאפיין חשוב בשפה, שמשפיע על צורת הקימפול שלה.

כפי שכבר כנראה אתם יודעים, Javascript היא שפה שהיא Dynamically Typed, שזה בדרך כלל כיף מאוד למפתחים, והרבה פחות כיף מאוד לכותבי הקומפילרים של השפה.

נסתכל על קוד Javascript לדוגמה:

```
const dog = {happy:true};  
const my_dog = Object.create(dog);  
console.log(my_dog.happy);
```

בכדי להריץ את השורה האחרונה, Parsern יבצע קודם כל את הבדיקה הבאה:

```
my_dog.hasOwnProperty("happy");
```

זאת אומרת, הוא יבדוק האם לאובייקט dog יש את השדה happy. אם לא, אז הוא ממשיך בשרשרת הירושה למעלה, ובודק (בדוגמה שלנו הוא יעצר ב-dog). יש עוד המון בדיקות שקורות בדרך, בחרתי להתמקד ספציפית בזו, כי היא דיי נפוצה.

זכרו את החלק הזה - הבדיקות האלו לוקחות זמן, ובשפות שהן Dynamically Typed, הן חלק לא מבוטל מזמן הריצה.

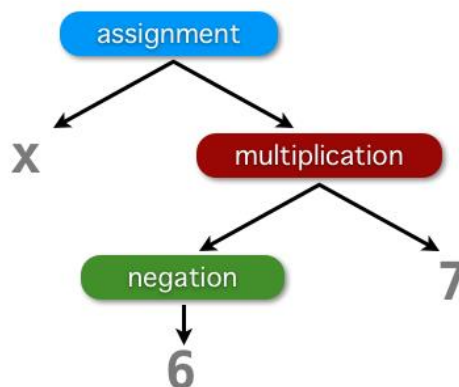
Abstract-Syntax-Tree

אז בדוגמה שלפנינו אפשר לראות את V8 (מנוע ה Javascript של Chrome), שלוקח את הקוד שכתבנו ומיצר ממנו Abstract Syntax Tree - בעצם, הצגה כללית מאוד, המתארת את מהלך הקוד.

לדוגמה, עבור קוד ה-Javascript:

```
let x = -6 * 7;
```

נקבל את ה-AST הבא:



מכאן, נוכל להעביר את ההצגה הכללית הזו, כמעט לכל שפה.



ByteCode & JIT

בשלב הבא, הקוד יגיע ל-Ignition Compiler, שזה שם יפה (יש המון שמות יפים לדברים ב-Chrome, כמו שלמודול שאחראי על העוגיות קוראים CookieMonster) למוצר שהופך את ה-Abstract Syntax Tree ל-Bytecode, ומשם, ל-Assembly.

- ByteCode - אבסטרקציה של קוד מכונה / סט פקודות מופשט שמוכר למי שאמור להעביר את זה ל-Assembly (במקרה של V8 זה TurboFan), אבל הוא לא מפורט כמו Assembly אלא בעיקר מהווה שפת ביניים זריזה ופשוטה. לדוגמה, עבור הקוד הבא:

```
let result = 1 + obj.x;
```

נקבל את ה-Bytecode הזה:

```
ldaSmi [1] ; load small integer 1 (Smi == small integer)
Star r0 ; into r0
LdaNamedProperty a0, [0], [4] ; get obj.x
Add r0, [6] ; add 1 to obj.x
```

בנוסף - אם תרצו לראות את ה-Bytecode שיוצא מ-v8 - אפשר פשוט לרשום את האופציה -print-bytecode בתור flag בשורת הרצה של Chrome או Node.js

הספקנים (או אנשי C\CPP), יכולים לשאול "למה שנרצה בכלל להעביר ל-ByteCode ולא ישר להפוך את כל ה-Javascript ל-Assembly?" שאלה מעולה.

1. לפני שבכלל נתחיל לרוץ, זה יקח לנו המון זמן להעביר את כל הקוד ל-Assembly
2. זוכרים ש-Javascript היא Dynamically Typed? זה אומר שצריך להעביר את כל האופציות האפשריות של הקוד ל-Assembly, וזה פוגע משמעותית בזמן הריצה (יעילות מקום וזמן ריצה)
3. יכול להיות שחלק מהקוד לא יקרא בריצה הנוכחית, או לפחות בזמן הקרוב (לדוגמה - פייסבוק יטענו לכם את liven של דודה שלכם ברקע, כשאתם רק באתם לראות מה הצעת החברות החדשה שקיבלתם).

אכן בעבר, כשאתרים היו קומפקטיים (הרבה) יותר, ההמרה הזו הייתה קורת, וכל ה-Javascript היה הופך ל-Assembly ישירות. הפסיקו עם זה מאז IE9.

אז אחרי שהשתכנענו בצדקת Bytecode, נרצה להריץ אותו. בכדי לעשות כך נקרא ל-TurboFan, שהוא JIT Compiler, שזה בעצם אומר שהוא אחראי על ההמרה מ-Bytecode ל-Assembly.



משמע, הקוד שראינו מקודם, יהפוך ל:

```
movl rbx, [rax+0x1b]
REX.W movq r10, 0x100000000
REX.W cmpq r10, rbx
jnc 0x30d119104275
REX.W movq rdx, 0x100000000
call 0x30d1191043e0
Int3laddl rbx, 0x1
....
```

שזה ממש Assembly שמתאים למעבד.

- JIT (Just-In-Time) Compiler - אחראי על העברה של ה-ByteCode ל-Assembly, בדיוק ברגע שבו קוראים לקטע הקוד הנחוץ. לכן הוא נקרא Just-In-Time, ולא Ahead-Of-Time (לדוגמה C היא Ahead-Of-Time, כיוון שהיא מתמקפלת מבעוד מועד ל-Assembly).
- יכול מאוד להיות שאתם חושבים לעצמכם "רגע...זה לא כזה חכם, אם אני קורא לפונקציה המון פעמים, אז כל פעם להמיר אותה ל-Assembly מ-Byte-Code זה בזבז זמן רציני".
- אם חשבתם את זה לעצמכם - קודם כל, כל הכבוד. תחשבו על להגיש משרה לצוות שעוסק בקומפילרים, ועל הדרך, נסו לחשוב על פתרון (מפורט בנקודה הבאה).
- במקרה של קטע קוד שנקרא המון פעמים - הפתרון היה לסמן אותו כ"קוד חם", ולשמור בצד המרה שלו ל-Assembly, ולהשתמש בה כ-Cache

בואו נסכם את איך ה-JIT עובד, בדוגמה קצרה:

זוכרים את זה ש-JS היא Dynamically Typed? אם מריצים את הקוד הבא:

```
function get_x(obj) {return obj.x;}
for (dog of my dogs) {get_x(dog)}
```

אז בהתחלה, המנוע יבדוק האם x קיים אצל obj, משמע:

```
obj.hasOwnProperty("x");
```

אם לא, הוא יתחיל לחפש בעץ הירושה שלו (ממי ש-obj יורש ממנו, עד ל-Object, האובייקט שכולם יורשים ממנו ב-JS). ורק אחרי כל מיני בדיקות כאלו ואחרות - הוא ימיר את הקוד ל-Assembly, ויריץ אותו.

בגלל שאנחנו קוראים ל-get_x המון פעמים, TurboFan (ה-JIT ב-V8), יסמן את הפונקציה כ"פונקציה חמה" וישמור עותק של ה-Assembly מיד, וכל פעם נקרא לאותה פונקציה - מבלי להמיר מחדש.

חשוב לציין - ב-Assembly של get_x, עבור ארגומנט מסוג dog נקבל אסמבלי שונה מקריאה ל-get_x עם ארגומנט מסוג cat. לכן יש צורך "לקמפל" את הקוד מחדש, כל פעם.

אז אם הקוד שלנו היה מכיל המון קריאות ל-get_x עם המון סוגי אובייקטים שונים?

```
// Define a lot of different classes (dog, cat, duck, etc...)
for (animal of animals) {get_x(animal);}
```

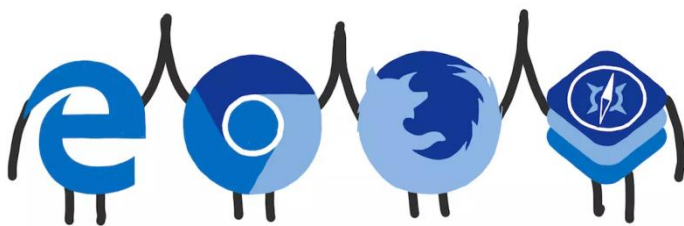
האם TurboFan היה יוצר Assembly עבור כל קריאה של אובייקט שונה?

התשובה היא: לא. בשלב מסויים הוא יפסיק לייצר Assembly חדש לאותה פונקציה. (בב TurboFan Default הוא אחרי 4 פעמים שונות).

ומעתה, עבור קריאה ל-get_x - היינו צריכים לעשות את הסדרת בדיקות (שלוקחת זמן בפני עצמה), ואת ההמרה מחדש, מה שהיה יקר מאוד מבחינת זמן ריצה.

זה אחד מהאתגרים שהמנוע מתמודד איתם בשפה שהיא Dynamically Typed, וזה אחד מהאתגרים, שנראה בהמשך - שלא קיימים ל-Webassembly

הקשר ל-WebAssembly

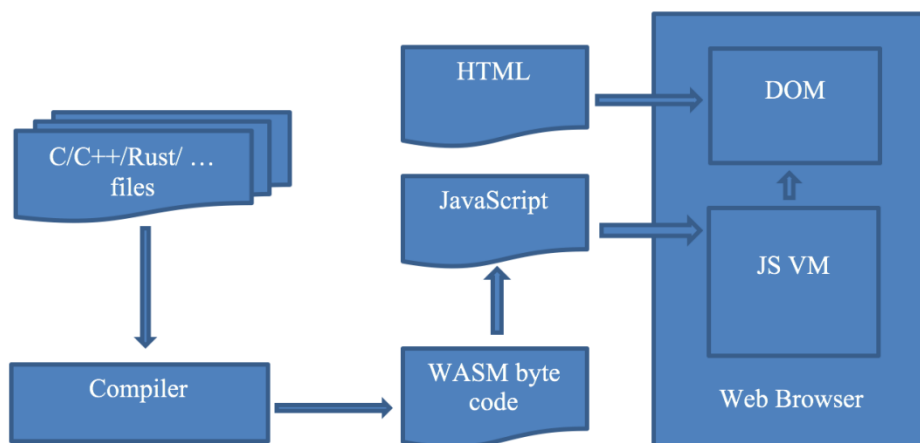


לפני כמה שנים, החליטו כל יצרניות הדפדפנים להתאחד - במטרה לגרום לאפליקציות Web לרוץ מהר יותר, ובכך - לפתוח את עולם ה-Web לאפשרויות רבות הרבה יותר (עיבוד גרפי, משחקים מורכבים, וכו').

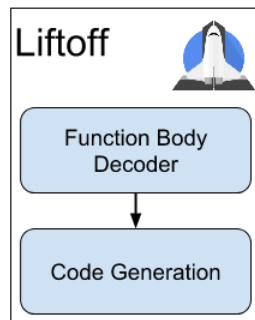
המוצר שיצא הוא ה-Web Assembly, או Wasm בקיצור.

Web Assembly הוא קוד שאפשר לכתוב בשפה כמו C/C++/Rust, "לקמפל" אל ByteCode*, ופשוט להכניס אל תוך הדפדפן, והוא ירוץ כחלק מהאפליקציה. הן חשבו על זה שזה יהיה הרבה יותר מהיר, אם הם פשוט יקבלו את הקוד "מקומפל" אל הJON שלהם, ויריצו אותו כשהם צריכים.

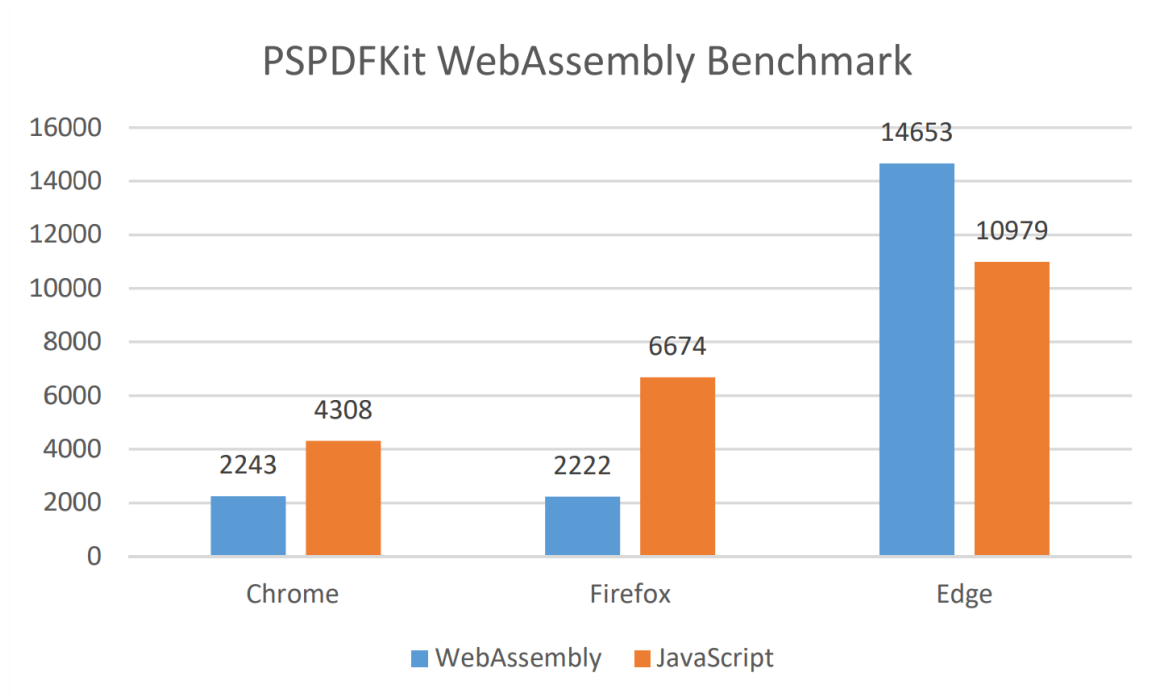
זוה בערך נראה ככה:



- קוד ה-WASM שלנו, לאחר הקימפול, נכנס עם ה-Javascript אל V8, לחלק משלו, ולא צריך לעבור Parsing מעמיק (אין צורך ב-Parsing, פשוט כי הקוד הוא Statically Typed). הוא מתקבל בתור Bytecode*, ומומר מאוד מהר לשפת מכונה.
 - Bytecode* - ה-Bytecode המתקבל מה-Compiler של WASM שונה מזה שראינו עד כה, מהמון סיבות (לדוגמה, הוא בכלל נוצר משפות שהן Statically Typed).
 - רגע... למה בכלל Bytecode?
 - כל הכבוד, שוב שאלה טובה.
 - הדפדפנים צריכים לרוץ על כל מיני ארכיטקטורות (ARM בטלפונים, x86 ב-PC).
 - כל הדפדפנים צריכים לתמוך בתקן, ולכן יש ליצור שפה אחידה ביניהן
- מבחינת יצירת ה-Assembly, אפשר לראות את Liftoff (עוד רכיב בתוך v8), שממיר את ה-bytecode של WASM בצורה פשוטה וזריזה:



יש בו משמעותית פחות חלקים, מה שתורם מאוד גם ביעילות מקום, וגם יעילות זמן ריצה.



[חוץ מב-Edge שעומד בכל מקרה להתמזג לתוך Chrome]

אתם יכולים להריץ את ה-Benchmark בעצמכם, שמריץ ממש את המוצר עצמו, ולראות את השיפור. נכון לכתיבת שורות אלו, ב-Chrome 75 על לינוקס, ראיתי שיפור של 25%, נתון מעודד מאוד, במיוחד בהינתן שהBenchmark בוחן את המוצר עצמו, ולא סט פקודות תיאורתית.

בכל מקרה, השיפור שנראה בגרף הוא ראשוני יחסית, וישתפר עם הזמן. בעיקר בגלל שהטכנולוגיה עדיין לא הבשילה, והמלאכה עוד מרובה במנועי ה-WebAssembly, בטח בהשוואה למנועי ה-JS הנהוותיקים.

השיפורים שנראה הם גם באזורי ה-Compilers (אלו שממירים את ה-C/C++/Rust ל-Byte-Code של WebAssembly), וגם באזורי הדפדפנים (לדוגמה, המעבר מ-TurboFan אל Liffoff) שאמורים להאיץ את הקוד WebAssembly לרוץ מהר כמעט כמו המקביל ה-Native-י שלו.



מימושים של WebAssembly

כמו תמיד, עם טכנולוגיות חדשות יש כבר דוגמאות יצירתיות שמסתובבות ברשת - [החל מ-VM של Windows2000](#) שרץ על הדפדפן, ועד [למשחקים](#) או [עריכת סרטים](#).

אם נסתכל קצת על העתיד:

- תחשבו על שרתים ב-NodeJS שהחלקים שדורשים פחות חישובים יהיו ב-Javascript, ומה שהכרחי יהיה ב-WASM
- Progressive Web Apps - טכנולוגיה קיימת (נתמכת ב-iOS ו-Android) שמאפשרת לאפליקציות Web לרדת כאפליקציות למכשיר, ולהתנהג בצורה דיי דומה לאפליקציות רגילות.
 - ביצועי האפליקציות האלו יכולים להיות דומים (אם לא טובים יותר) מאפליקציות Native, ולהכתב רק פעם אחת לכל שלושת הפלטפורמות הגדולות.

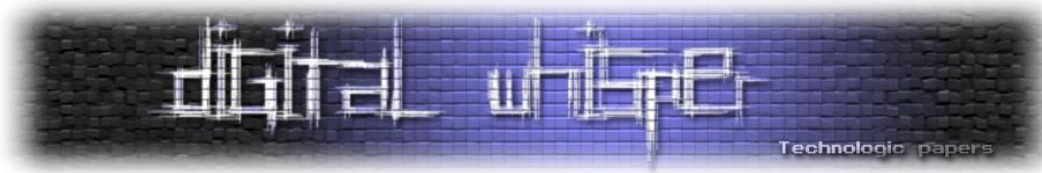
Wasm Hello World

שימו לב לדוגמה הבאה:

C++11 -Os	COMPILE	Wat	ASSEMBLE	DOWNLOAD	Firefox x86 Assembly
<pre>1 int doubler(int num) { return num*2; }</pre>		<pre>1 (module 2 (table 0 anyfunc) 3 (memory \$0 1) 4 (export "memory" (memory \$0)) 5 (export "_Z7doubleri" (func \$_Z7doubleri 6 (func \$_Z7doubleri (; 0 ;) (param \$0 i32) 7 (result i32) 8 (i32.shl 9 (get_local \$0) 10 (i32.const 1) 11) 12) 13)</pre>			<pre>- wasm-function[0]: sub rsp, 8 mov ecx, edi mov eax, ecx shl eax, 1 nop add rsp, 8 ret</pre>

כפי שניתן לראות בה, אנו יוצרים פונקציה חדשה בשם doubler, תחילה היא הופכת להיות ה- WebAssembly ByteCode (שנקרא גם Wat), ולאחר מכן (ממש בדפדפן עצמו) מתקמפלת עבור Firefox ועוברת להיות x86 Assembly מוכר וידוע.

יש כמה [מדריכים פשוטים](#) באינטרנט שאיתם אפשר להתחיל לשחק עם WASM, שווה לנסות.



סיכום

במאמר זה, פתחנו בהסבר קצר על מנועי דפדפנים, על איך שפות שהן Dynamically Typed (ובתוכן Javascript) רצות, מהו מנוע ה-JIT, ואיך כל זה נראה ב-Chrome.

לאחר מכן, התעמקנו בחסרונות שלהם, וביתרונות של שפות שמתקמפלות Ahead Of Time. משם עברנו ישירות לפתרון המוצע בעולם ה-Web, שהוא ה-WebAssembly. הראנו את השיתוף פעולה (הדיי יחודי) של יצרניות הדפדפנים השונות, בכדי ליצור אפשרות לאפליקציות Web לרוץ מהר יותר.

התעמקנו ביתרונות שלו במצבים מסוימים על פני Javascript, ועל איך הוא עובד. קינחנו בהסתכלות קצרה על הביצועים של WebAssembly כיום, ועל המימושים שלו.

אני מאמין שנמשיך לראות את הטכנולוגיה הזו מאפשר לעוד אפליקציות לרוץ בדפדפן בצורה חלקה וללא הבדלים בין ה-Web ל-Native, ועד מהרה, נמצא את עצמנו בעולם שבו הסטנדרט הוא Write-Once-Run-Everywhere, לפחות בשלושת הפלטפורמות הגדולות (Android, iOS, Web).

מוזמנים לפנות אלי במייל לכל שאלה: rondalal54@gmail.com