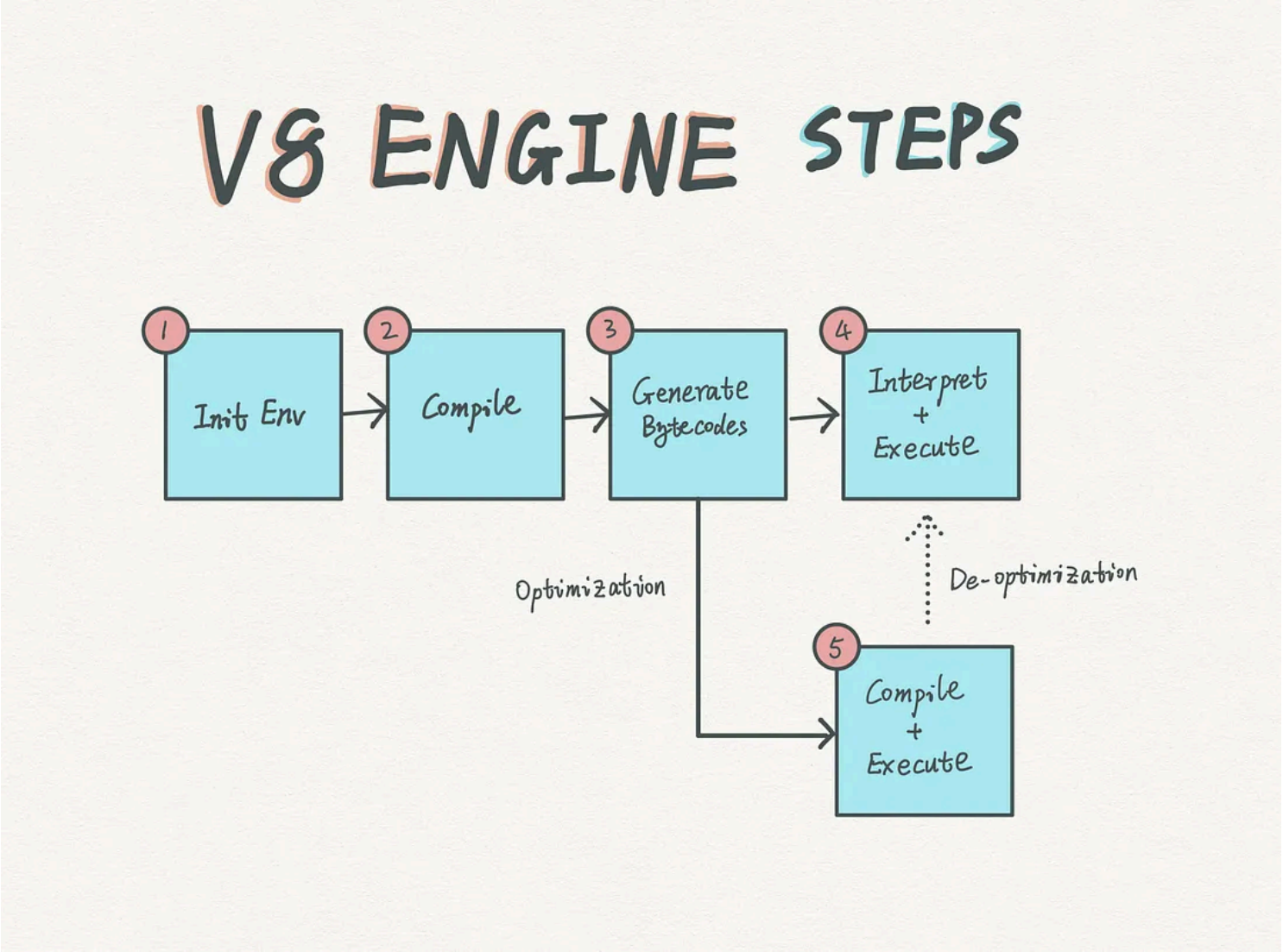
How V8 JavaScript engine works step by step [with diagram]

Carson · Follow

8 min read · Nov 5, 2020



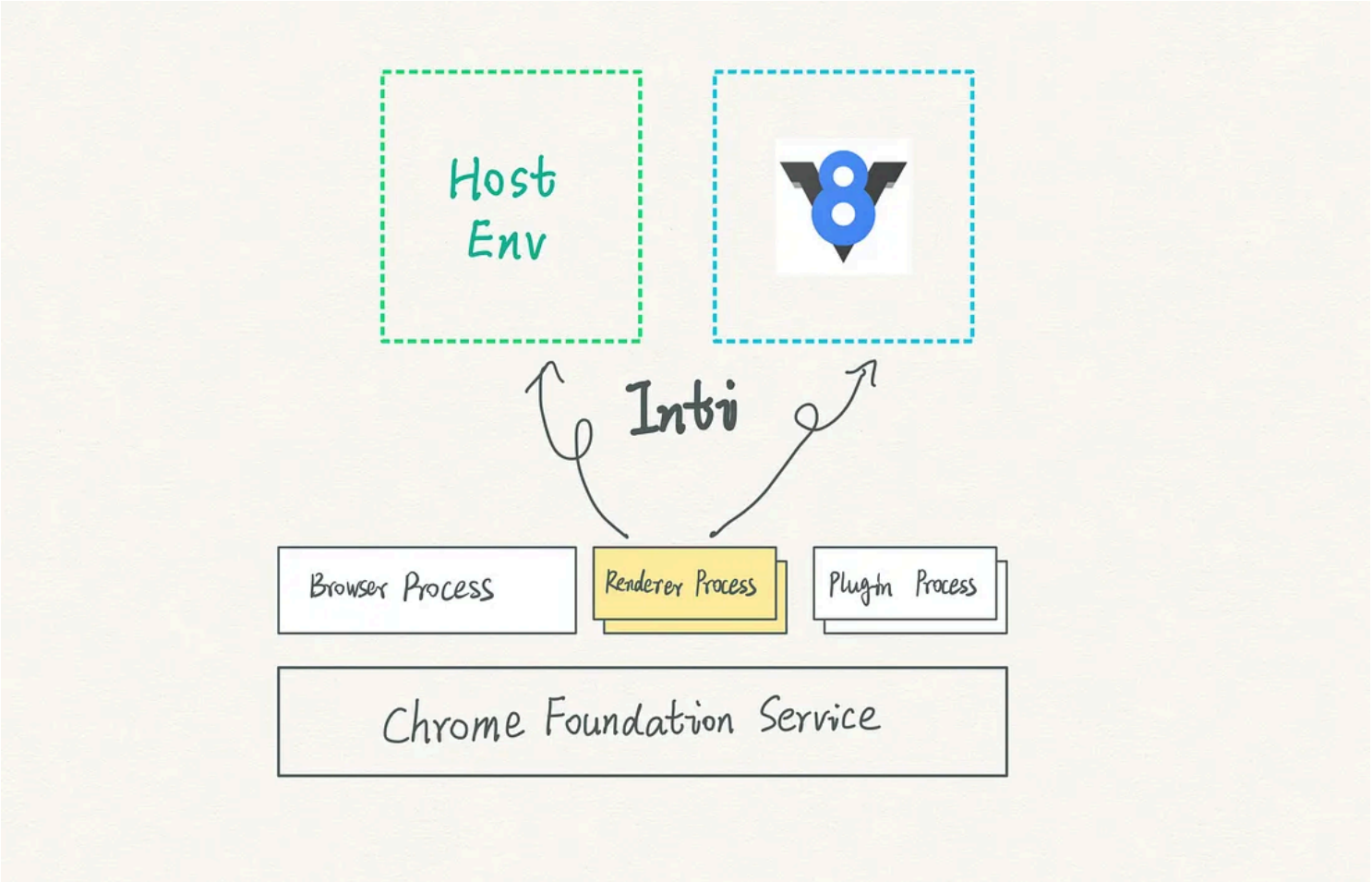
From a high-level view, the V8 JavaScript engine execution consists of 5 steps.

1. Initialize environment in the host
2. Compile JavaScript codes
3. Generate bytecodes
4. Interpret and execute bytecodes
5. Optimize some bytecodes for better performance

1. Initialize environment

Technically, this is not part of V8's job. It is the renderer process of the browser initializes the following two items:

- Host environment
- V8 engine

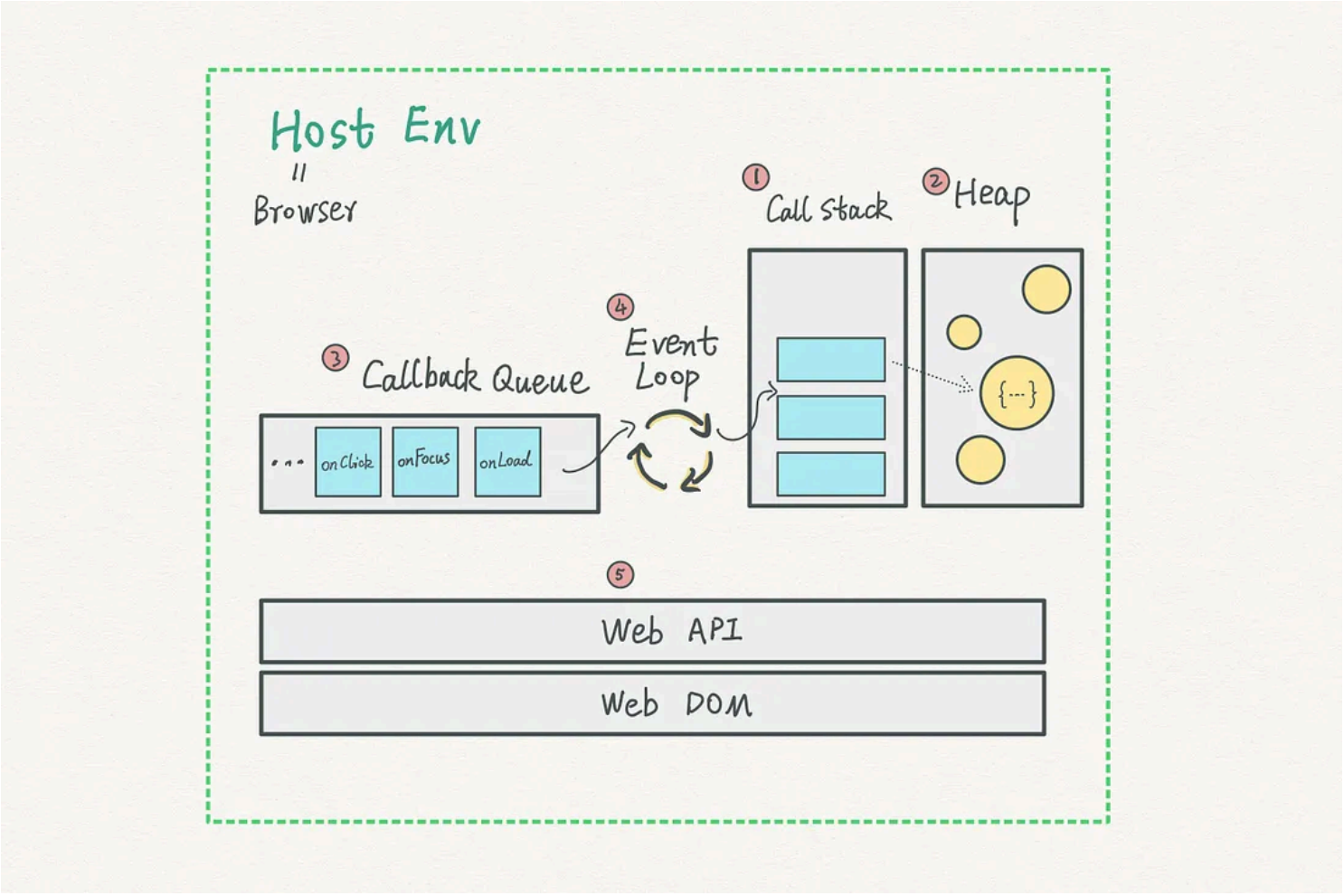


A browser has multiple renderer processes. Usually, each browser tab has a renderer process and initializes a V8 instance.

If you are interested in the renderer process and more about how the browser works, check this post.

What is the host environment? In our context, it is the browser. Therefore, we will use the “browser” and the “host environment” interchangeably in this post. However, keep in mind that a browser is merely one of the host environments for JavaScript. Another famous one is the Node host environment.

What is in the host environment?



The host environment provides everything a JavaScript engine relies on, including:

1. Call stack

2. Heap
3. Callback queue
4. Event loop
5. Web API and Web DOM

User interactions on a web page trigger a series of events. The browser added them to the callback queue along with associated callback functions. The event loop working like an infinite while-loop keeps fetching a callback from the queue. Then the JavaScript in the callback is compiled and executed. Some intermediate data is stored in the call stack. Some are saved in the heap, such as an array or an object.

Why does the browser store data in two different places?

- **Trading space for speed:** A call stack requires continuous space in memory, making it fast to process. However, continuous space is rare in memory. To resolve the issue, browser designers restrain it with max size. This is where the stack-overflow error comes from. Usually, the browser saves data with limited size in the call stack, such as an integer and other primary data types.
- **Trading speed for space:** Heap doesn't require continuous space to save extensive data like an object. The tradeoff is that the heap is relatively slow to process the data.

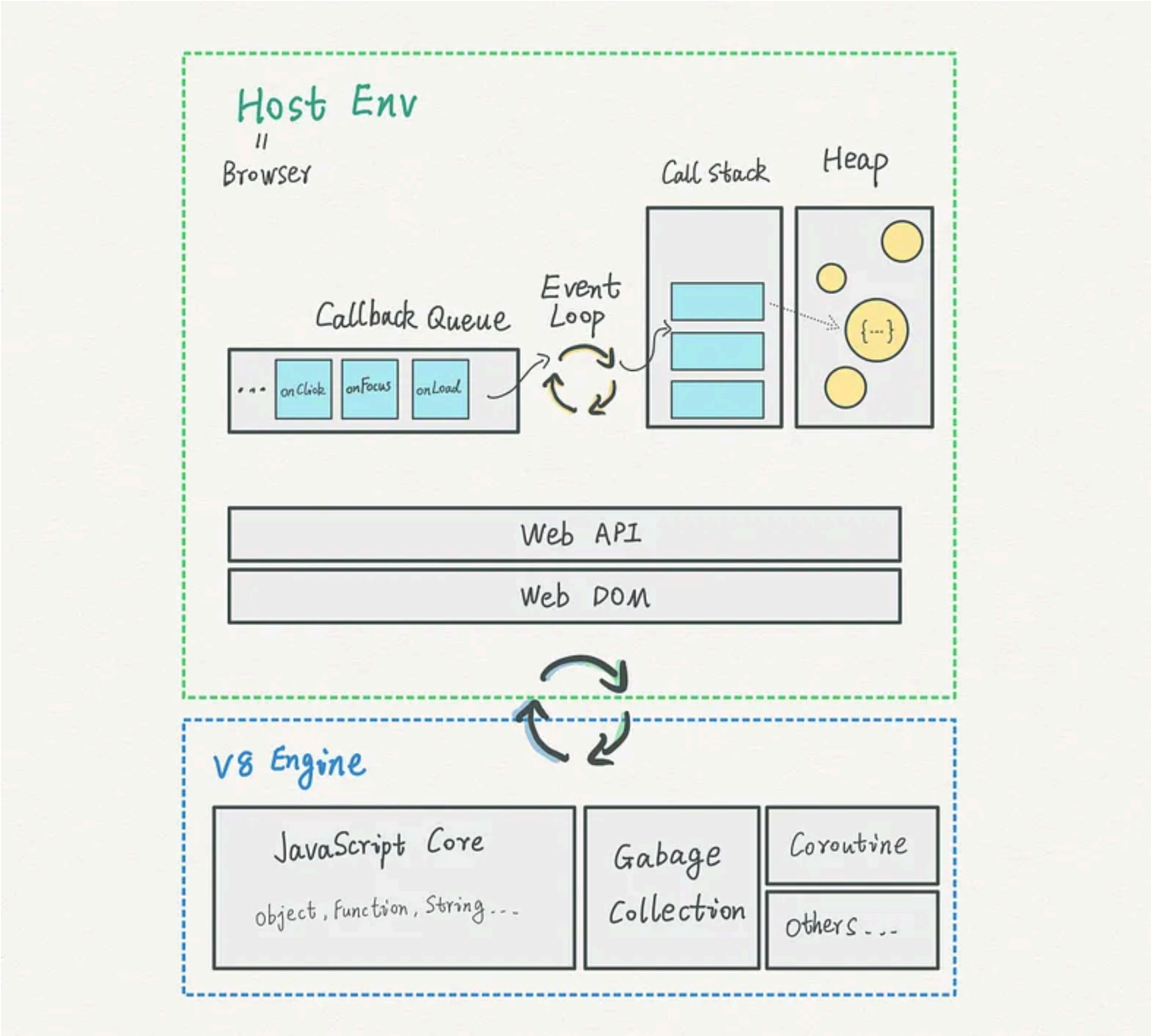
In my opinion, the call stack and the event loop are two critical mechanisms to understand how JavaScript works, which is beyond this post.

- I have a post explaining how the call stack works. More reading resources are attached at the end of the post if you want to learn more about the mechanism.
- Regarding the event loop, this post from Jake Archibald is the best one with interactive examples.

V8 engine relies on and empowers the host environment

The host environment to V8 is like your computer's operating system to software. Softwares rely on the operating system to run. Meanwhile, they empower your system to do so many advanced tasks.

Take Photoshop, for example. It needs to be run on Windows or macOS. Meanwhile, your operating system cannot make a beautiful poster for you, but Photoshop can.

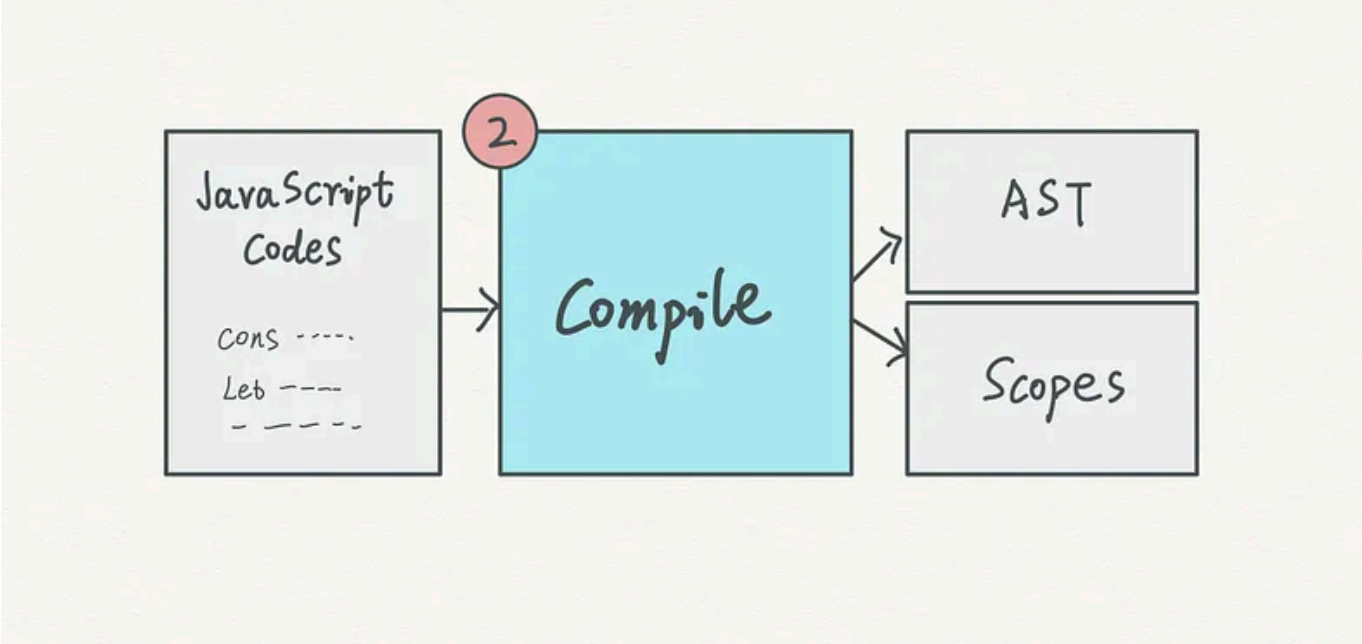


Same as the V8 engine, it provides additional features on top of the host environment:

- JavaScript core features based on the ECMAScript standard, such as the creation of Object and Function
- Garbage collection mechanism
- Coroutine features
- And more...

When the host environment and V8 engine are ready, the V8 engine starts its next step.

2. Compile JavaScript codes



At this step, the V8 engine converts the JavaScript codes to Abstract Syntax Tree (AST) and generates scopes.

The V8 engine doesn't speak JavaScript language. The script needs to be structured before processing.

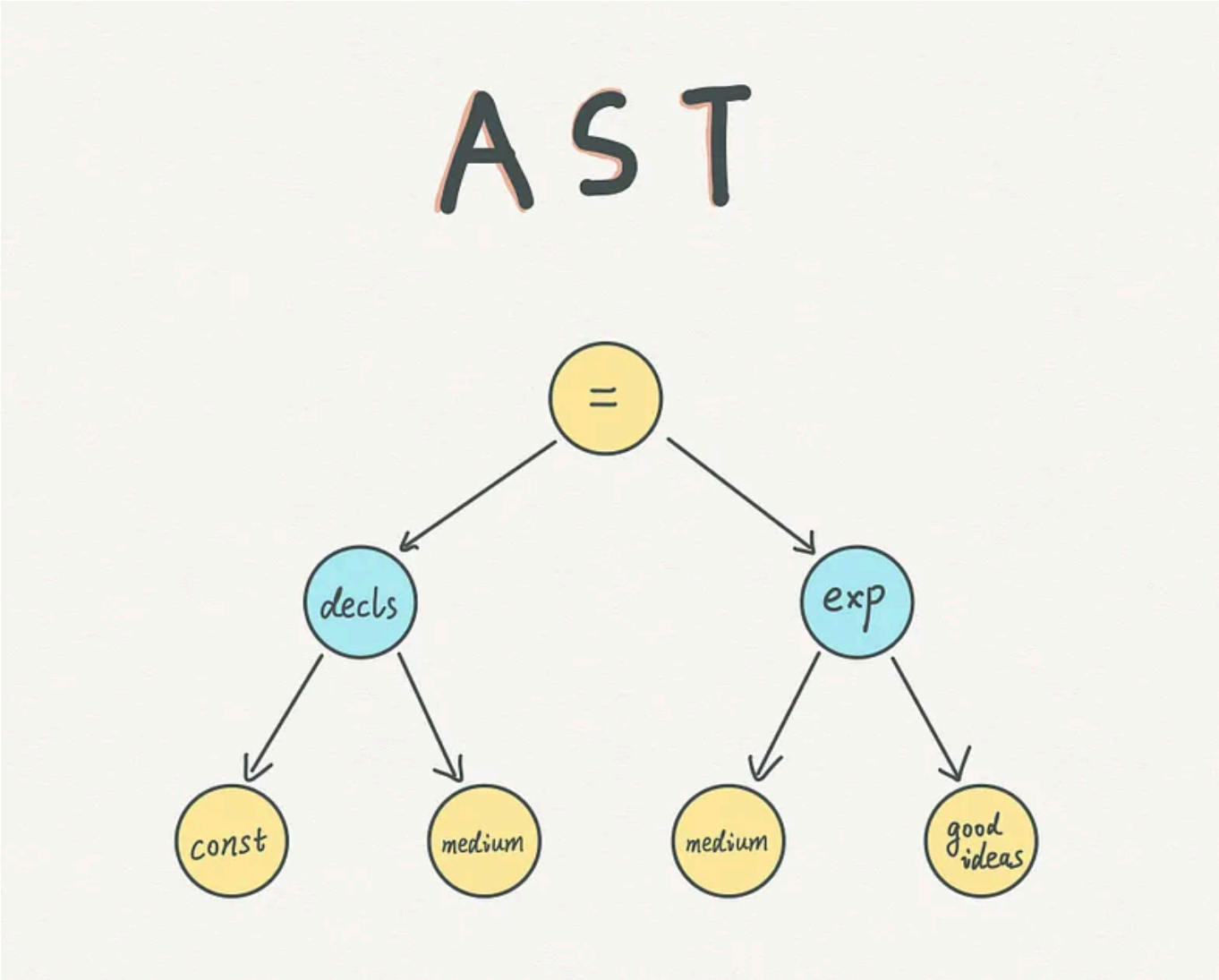
AST is a tree structure, easy for V8 to digest.

Meanwhile, scopes are generated at this step, including the global scope and more scopes at the top of it stored in the host environment's call stack. The scope itself worths another post to explain. You can safely skip it here.

How an AST looks like?

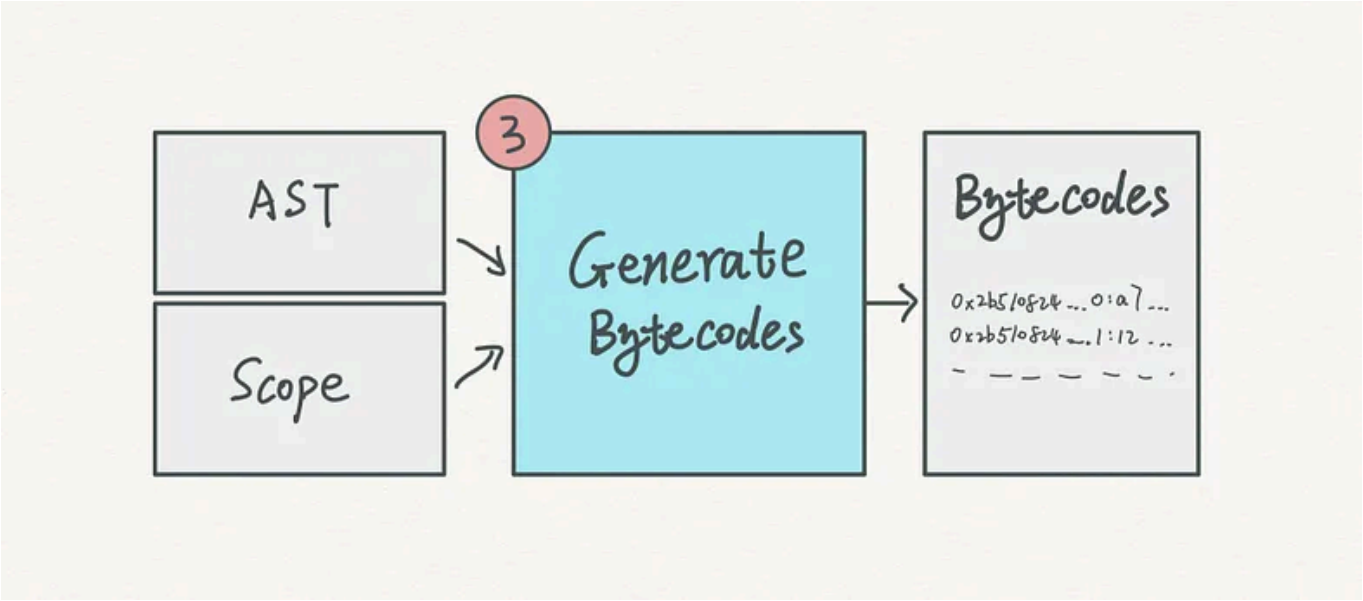
Let’s check a simple example by showing the following JavaScript in an AST format.

```
const medium = 'good ideas';
```



Each line of your JavaScript codes will be converted into AST, like the example at this step.

3. Generate bytecodes



At this step, the V8 engine takes the AST and scopes and outputs bytecodes.

How do the bytecodes look like?

Let's use the same example, this time, with D8, the developer shell of Chrome V8.

To install the D8 in macOS, run the following command in the terminal.

```
brew install v8
```

Save our example codes in a javascript file `v8.js`, and run the following command in the terminal.

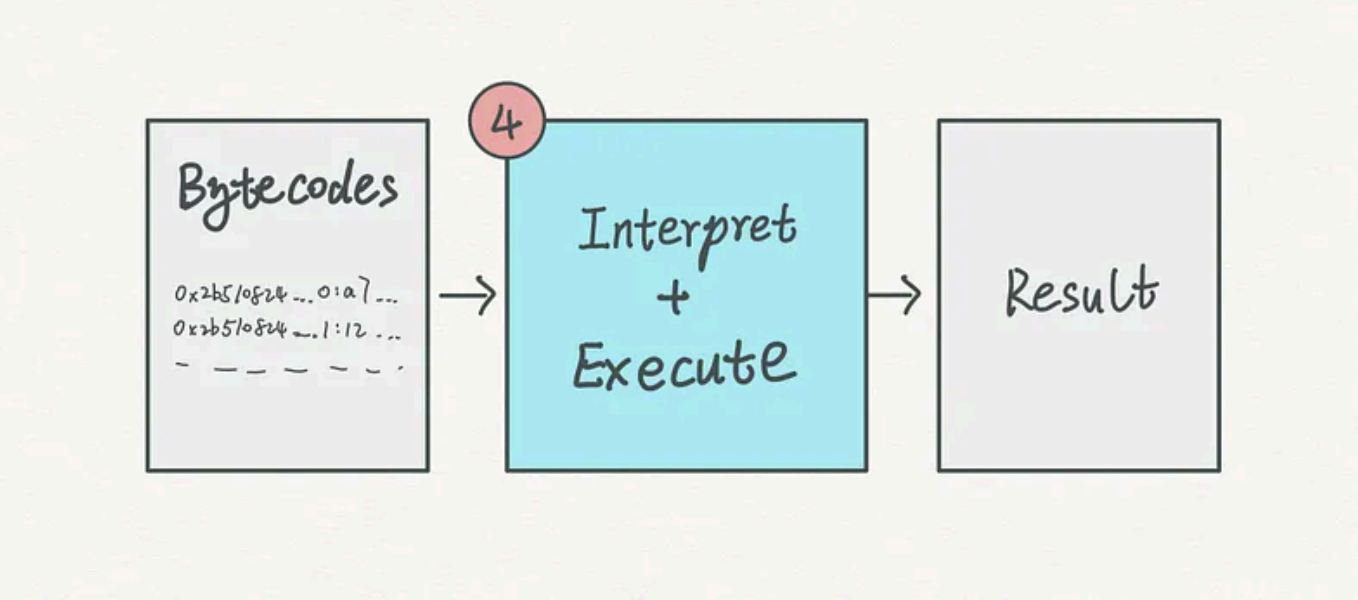
```
d8 --print-bytecode v8.js
```

D8 prints the bytecodes generated based on the AST and scopes from the last step.

```
[generated bytecode for function:  (0x0ee70820ffed
<SharedFunctionInfo>)]
Parameter count 1
Register count 1
Frame size 8
  0xee708210076 @    0 : 12 00             LdaConstant [0]
  0xee708210078 @    2 : 1d 02             StaCurrentContextSlot [2]
  0xee70821007a @    4 : 0d                LdaUndefined
  0xee70821007b @    5 : aa                Return
Constant pool (size = 1)
Handler Table (size = 0)
Source Position Table (size = 0)
```

`Parameter count 1` means there is one parameter, which is the `medium` in our case. Then, there are 4 lines of bytecodes for the interpreter to execute.

4. Interpret and execute bytecodes



The bytecodes are a collection of instructions. At this step, the interpreter will execute each line of bytecodes from top to bottom.

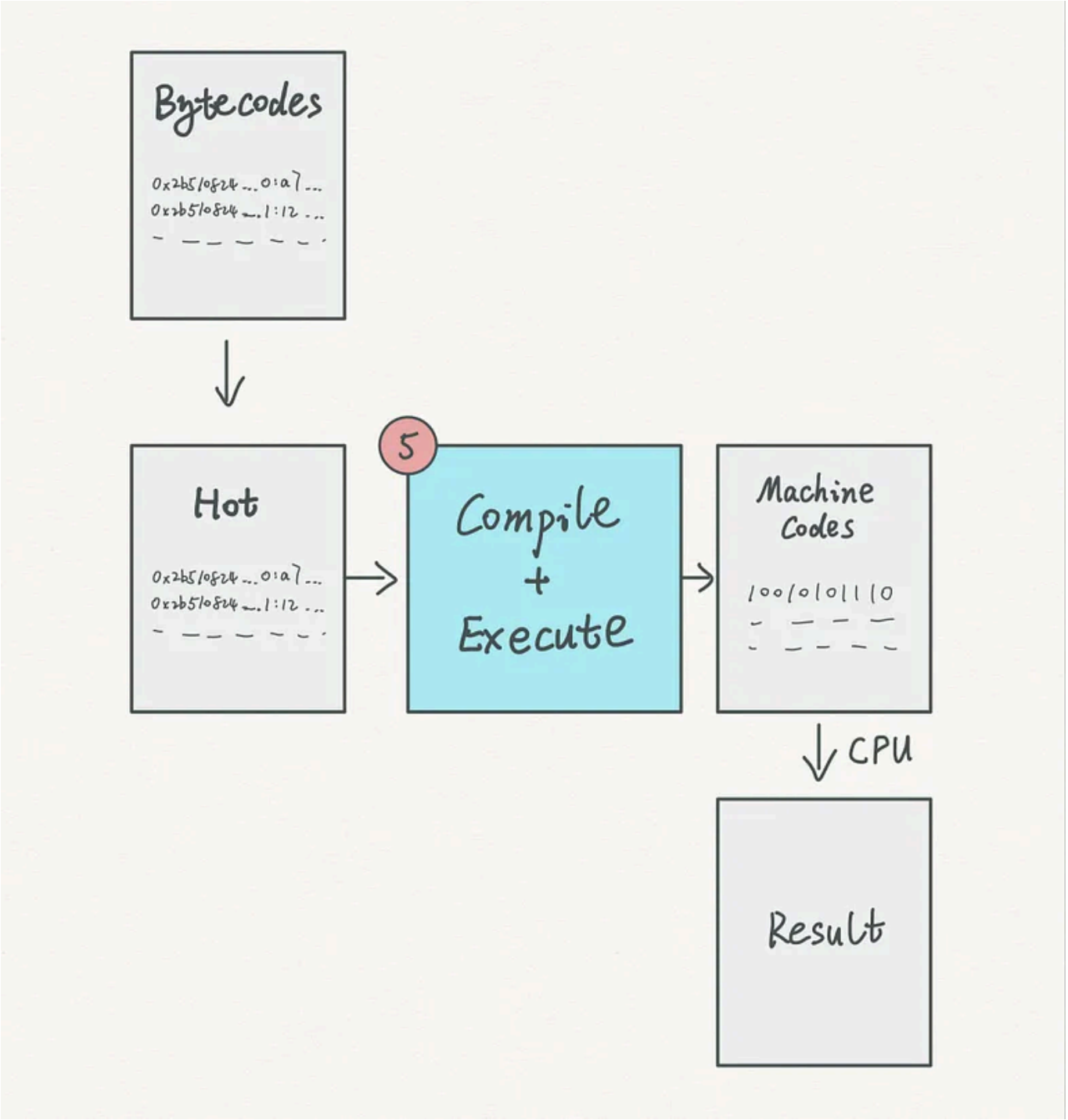
In the previous example, we see the following 4 bytecodes.

```
LdaConstant [0]
StaCurrentContextSlot [2]
LdaUndefined
Return
```

Each line of the bytecodes is like a prefabricated block of Lego. No matter how fancy your codes are, all are built with these basic blocks behind the scene.

The details of each bytecode are out of the scope of this post. If you are interested in it, here is a full list of the V8 bytecodes.

5. Compile and execute machine codes

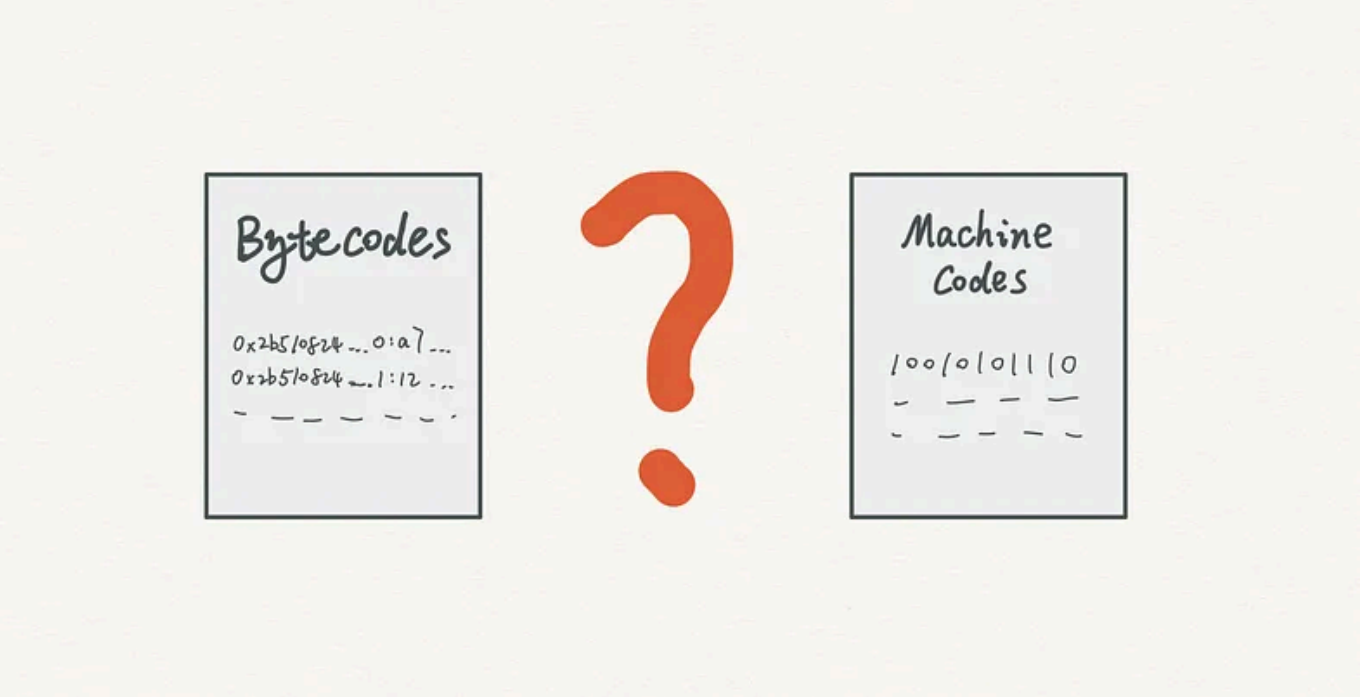


This step is parallel to the previous one. When executing the bytecodes, V8 keeps monitoring the codes and looking for opportunities to optimize them.

When some frequently used bytecodes are detected, V8 marks them as “hot.” Hot codes are then converted to efficient machine codes and consumed by the CPU.

What if the optimization fails? The compiler de-optimizes codes and let the interpreter executes the original bytecodes.

Bytecodes vs. machine codes



But why not V8 uses faster machine codes directly? Wouldn't introduce intermediate bytecodes slow down the entire process?

Theoretically, yes. But that's not the whole story.

Interestingly, that's precisely how the V8 team initially designed the JavaScript engine. At the early age of V8, the steps are the following:

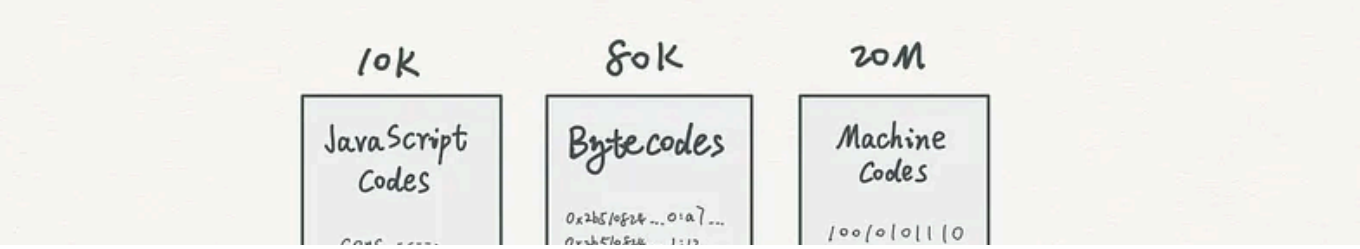
1. V8 compiles scripts to AST and scopes.
2. A compiler compiles the AST and scopes to machine codes.

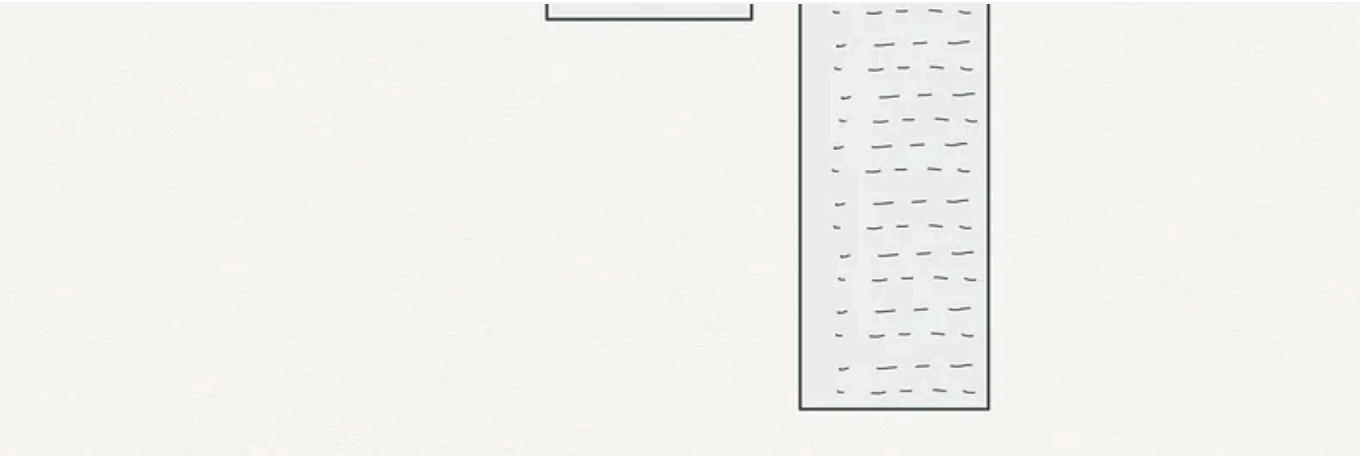
3. V8 detects some frequently used machine codes and marks them as “hot.”
4. Another compiler optimizes the “hot” codes to optimized machine codes.
5. If the optimization fails, the compiler runs the de-optimization process.

Though today’s V8 structure is more complicated, the basic idea remains the same.

However, the V8 team introduces bytecodes when the engine evolves. Why? Because using machine codes along brings some troubles.

1. Machine codes requires a great amount of memory





The V8 engine stored complied machine codes in the memory to reuse them when the page loads.

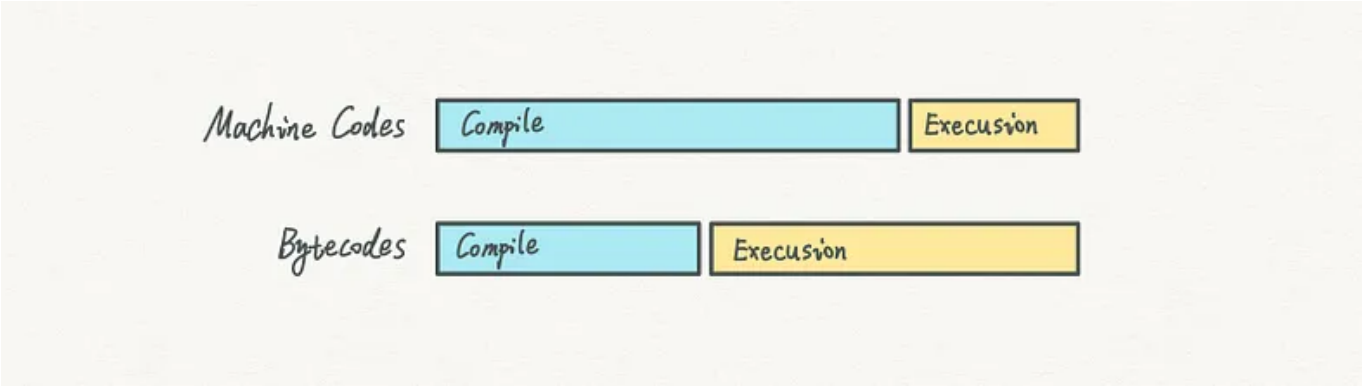
When compiled to machine codes, a 10K JavaScript could inflate into 20M machine codes. That's about 2,000 times larger memory space.

How about the size of the bytecodes in the same case? It is about 80K. Bytecodes are still more massive than the original JavaScript one, but it is way smaller than its corresponding machine codes.

Today, it is common to see JavaScript files over 1M. A 2G memory consumption for machine codes is not a good idea.

Thanks to the size reduction, a browser can cache all compiled bytecodes, skip all previous steps, and execute them directly.

2. Machine codes are not always faster than bytecodes



Machine codes take a longer time to compile, although it is lighting-fast in terms of execution.

Bytecodes need a shorter time to compile, but the tradeoff is a slower execution step. An interpreter needs to interpret bytecodes before executing them.

When we measure both options from end to end, which one is faster?

It depends.

The art is finding a balance between these two options while developing a powerful interpreter and a smart optimizing compiler for the bytecodes.

Ignition, the interpreter V8 using, is the fastest one on the market.

The optimizing compiler in V8 is the famous TurboFan, compiling highly-optimized machine codes from bytecodes.

3. Machine codes increases complexity in development

Different CPUs could have various structures. Each can only understand a kind of machine code. There are a lot of processor structure designs on the market. To name a few:

- ARM
- ARM64
- X64
- S397
- And more...

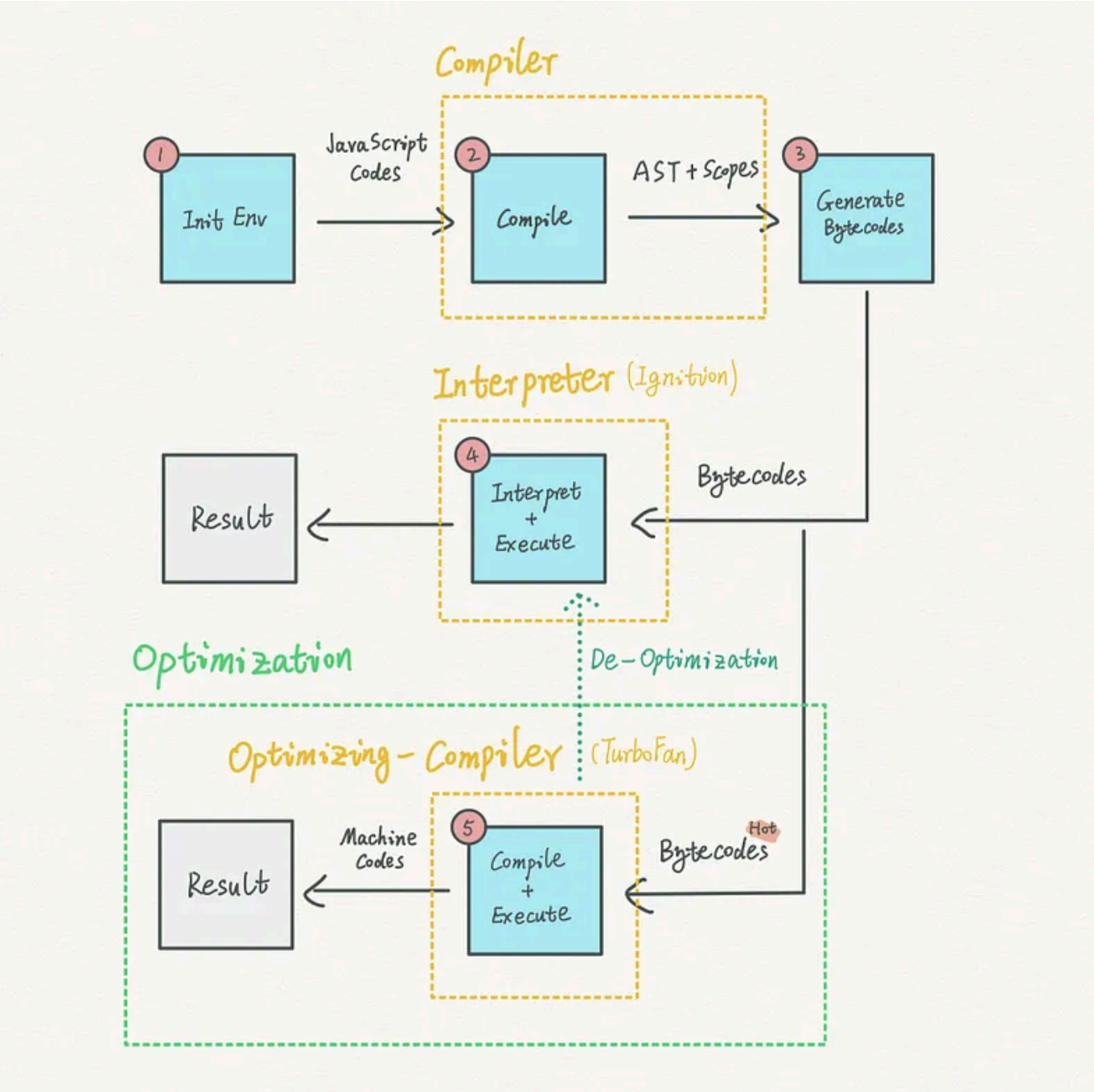
If the browser only uses machine codes, it needs to take care of so many cases separately. As a developer, we know that's not a good practice intuitively.

We need an abstract.

Bytecodes are the abstract between JavaScript and CPUs. By introducing the intermediate bytecodes, the V8 team reduces the workload to compile the machine codes. Meanwhile, it helps V8 to migrate to new platforms easily.

Takeaways

Putting everything together, now we can see a completed version of how Chrome V8 works from a high-level view.



Resources and references

- How JavaScript works: inside the V8 engine + 5 tips on how to write optimized code
- JavaScript V8 Engine Explained | Hacker Noon
- V8 docs
- V8 bytecodes list

- Crankshafting from the ground up
- V8 Resources

Browsers JavaScript Chrome V8 Javascript Engine

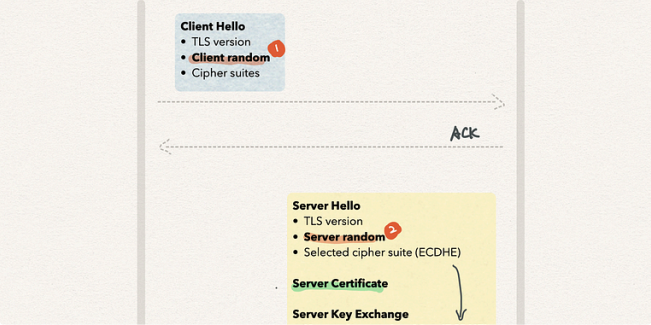
Written by Carson

471 Followers

a coder

Follow

More from Carson



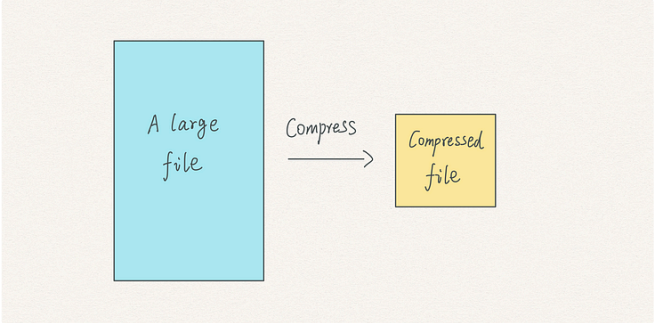
Carson

TLS 1.2 andTLS 1.3 Handshake Walkthrough

The ultimate goal of the TLS handshake is safely exchanging the master secret for futu...

7 min read · Mar 19, 2021

47 1



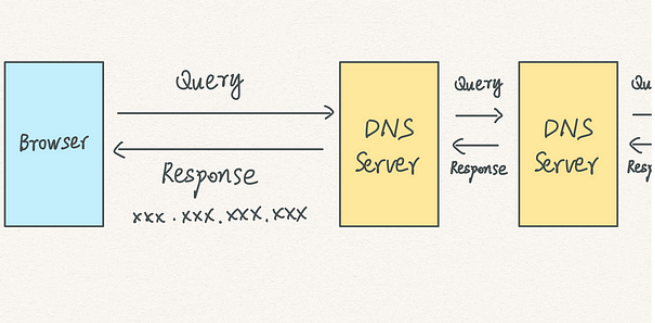
Carson

How HTTP Delivers a Large File?

In the early era of the network, people send files in single-digit KB size. In 2021, we enjoy...

6 min read · Feb 26, 2021

148 2

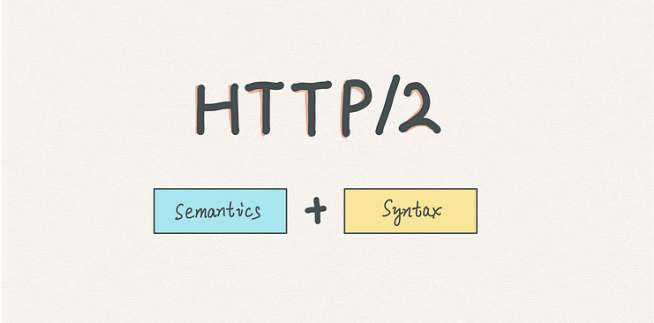


Carson

DNS Message—How to Read Query and Response Message

5 min read · May 5, 2021

28



Carson

HTTP/2 and How it Works

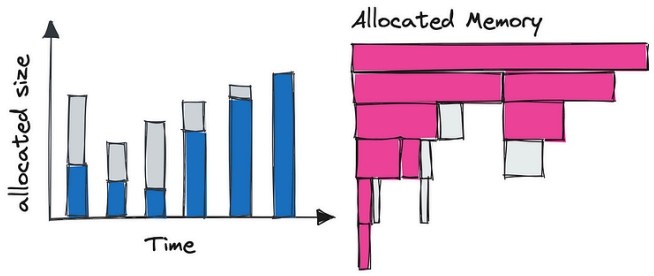
TLS helps improve security. Now it is time for performance enhancement, the focus of...

8 min read · Mar 24, 2021

116 1

See all from Carson

Recommended from Medium



Lily C... in Performance engineering for the ordin...

Intro to Memory Profiling & Chrome DevTools Memory Tab explained

I think the Homer hiding gif best captures my immediate reaction when I saw the Memory...

8 min read · Oct 11, 2023

165 1

Vit Prajzler

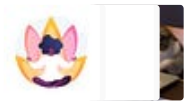
Using LoRa without LoRaWAN

What are the IoT problems that are better solved with “raw” LoRa compared to using th...

6 min read · Nov 10, 2023

9

Lists



Stories to Help You Grow as a Software Developer

19 stories · 858 saves

General Coding Knowledge

20 stories · 962 saves

Living Well as a Neurodivergent Person

10 stories · 610 saves

Generative AI Recommended Reading

52 stories · 761 saves

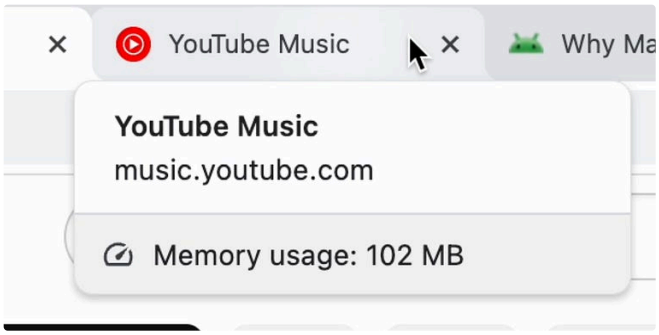
Satyendra Jaiswal

Caching Strategies for APIs: Improving Performance and...

In the dynamic landscape of web development, where responsiveness and...

9 min read · Nov 25, 2023

31



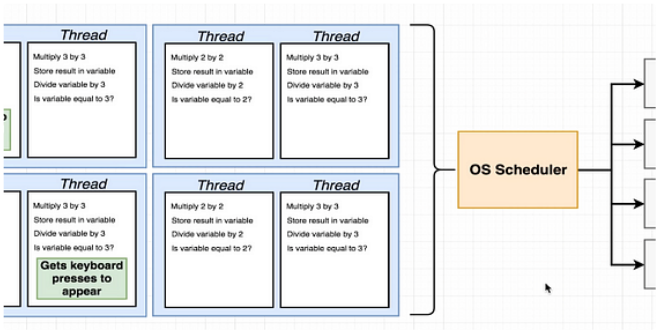
Addy Osmani

Chrome now shows each active tab's memory usage!

In Chrome Canary, we recently introduced hover cards that show the memory usage for...

5 min read · Oct 4, 2023

212 2



Mohammad Abbas

Understanding Processes, Threads, and Scheduling in Node.js

When it comes to understanding the internal workings of any programming environment,...

Saeid

Running Complex Long-Running Tasks in the Browser Using...

Discover how to execute complex, long-running tasks in the browser using JavaScrip...

3 min read · Sep 25, 2023

11

4 min read · Sep 4, 2023

1

See more recommendations