2020 International Conference on Intelligent Computing and Human-Computer Interaction (ICHCI) | 978-1-6654-2316-8/20/$31.00 ©2020 IEEE | DOI: 10.1109/ICHCI51889.2020.00008

Research and Application of Node.js Core Technology

Xiaoping Huang
Department of network technology
South China Institute of Software Engineering
Guangzhou University,
Guangzhou, 510990, China
Guangzhou, China
Hxp@sise.com.cn

Abstract. **Web development companies and developers can choose a variety of technology stacks to build Web applications. In the early days of network development, different technologies were used for front-end and back-end development. With the release of node.js, the construction of the website has undergone tremendous changes. Unlike single-threaded PHP and multi-threaded JAVA, a server programming platform based on the Chrome V8 engine JavaScript runtime environment-node.js came into being. Node.js uses its own built and defined attributes to make up for the shortcomings of the background development language in the traditional sense. It is a server-side JavaScript interpreter, which is used to conveniently build web applications with fast response speed and easy expansion. Node.js, with its event-driven, time loop mechanism, and non-blocking I/O model, can realize functions that Core JavaScript does not have or are not perfect, such as file systems, modules, packages, operating system APIs, and network communications. Historically, there has been more than one plan to port JavaScript outside the browser, but Node.js is the best one.**

Keywords: Technology stack; node.js; event-driven; event loop

I. INTRODUCTION

Node.js is a non-blocking (non-blocking), event-driven (event-based) I/O platform built on Google Chrome's v8 engine, with event-driven as its core. Its biggest feature is the use of asynchronous I/O and event-driven architecture design. For high-concurrency solutions, the traditional architecture is a multi-threaded model, that is, a system thread is provided for each business logic, and system thread switching is used to make up for the time overhead of synchronous I/O calls. Node.js uses a single-threaded model and uses asynchronous request methods for all I/O, avoiding frequent context switching. Its unique event loop mechanism uses the processing advantages of callback functions to maximize the efficiency of task access. These unique construction advantages can be that node.js can handle more than 40,000 user connections at the same time on a server with 8GB memory. In traditional server-side languages such as Php, Java or .NET, a new thread will be used for each client in a conventional way. According to the estimate that each thread will consume 2MB of memory, the maximum number of simultaneous connections to a server with 8GB of memory is the number of users is only about 4000.

II. EVENT DRIVEN

The event-driven mechanism is realized by Node.js through the internal single-threaded efficient maintenance of the event loop queue, which is different from the resource occupation of multi-threaded and the up and down switching between multiple tasks. In the face of large-scale data requests, Node.js. The asynchronous mechanism of the callback function in the event mechanism can be used to complete high-concurrency data processing.

Event-driven programming is to write corresponding event handlers for events that need to be handled. The code is executed when the event occurs. Write corresponding event handlers for events that need to be handled. Event-driven programs have the opportunity to release the CPU to enter the sleep state, and are awakened by the operating system when the event is triggered, so that the CPU can be used more effectively.

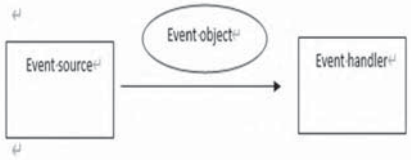

Figure 1 Event-driven model (uml) diagram

The event-driven model mainly includes three objects: event source, event and event handler.

Objects in Node.js will be triggered to achieve event division. When the client implements clicks or mouse operations and keyboard operations, the event object will call the data of the event source to generate an event object. The event object is the pre-encapsulated event information, which is triggered by the client when the event occurs. The callback function implements data processing according to the stack ordering of the event loop mechanism. For example: The fs.readStream object will trigger an event when the file is opened. All the objects that generate events are instances of events.EventEmitter.

For example: demo1

978-1-6654-2316-8/20/$31.00 ©2020 IEEE
DOI 10.1109/ICHCI51889.2020.00008

```
var events=require('events');
var eventsEmitter=new events.EventEmitter();
eventsEmitter.on('listen2',function (data) {
    console.log('Accept the second trigger')
})
eventsEmitter.on('listen',function (data) {
    console.log('Accept the first trigger');
    eventsEmitter.emit('listen2','Second trigger')
})
setTimeout(function () {
    eventsEmitter.emit('listen','First trigger')
},2000)
```

In this code, there are two listening events and two triggering events. When the trigger of the "listen" listener is triggered for the first time, the second listener "listen2" of its trigger event is also triggered. The results are as follows:

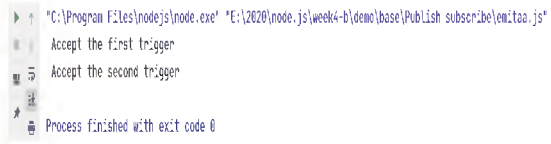

Figure 2 Event-driven running results of demo1

III. THE PRINCIPLE OF NODE.JS ARCHITECTURE

Node.js is compiled by JavaScript and C++.

C++ mainly completes the basic construction of compiling Node.js, and Node bindings is the middle layer, which is the key to the communication between JavaScript and C++. The former calls the latter through buildings to exchange data. The top-level Node.js standard library is written by JavaScript, which is an API that we can call directly during use.

Analyze node.js from the architecture level of Libuv, which is divided into v8 engine, libuv, builtin modules, native modules and other modules.

Your code: The edited code. Usually used to write logic operations and reference and use of third-party modules.

Node.js is the code of the core module. It is loaded as the node.js software is loaded, and you can just reference it when you use it.

Host environment: host environment. Provide various services, such as file management, multi-threading, multi-process, use of IO, etc. For the specific functions of other modules, please refer to the official website documents, so I won't repeat them here.

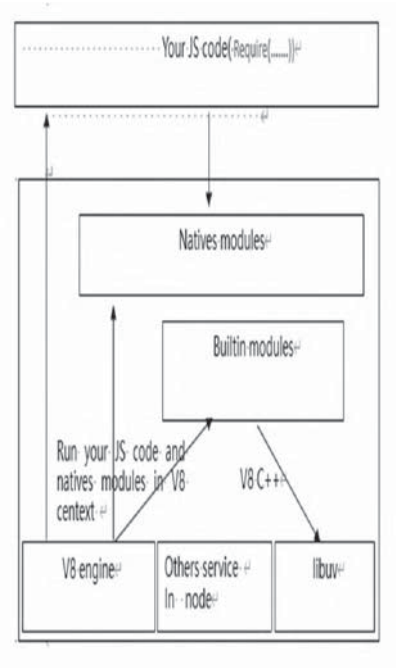

Figure 3 The architecture of node.js based on module dependencies

According to the characteristics of this architecture, it can interact with the operating system.

For example: demo2

```
var fs = require('fs');
fs.readFile('./song.txt', function(err, data) {
    if (err) {
        return console.log(' Failed to read lyrics file ');
    }
    data = data.toString();
    var lines = data.split('\n');
    var reg = /\[(\d{2})\:(\d{2})\.(\d{2})\]\s*(.+)/;

    for (var i = 0; i < lines.length; i++) {
        (function(index) {
            var line = lines[index];
            var matches = reg.exec(line);
            if (matches) {
                var m = parseFloat(matches[1]);
                var s = parseFloat(matches[2]);
                var ms = parseFloat(matches[3]);
                var content = matches[4];
                var time = m * 60 * 1000 + s * 1000 + ms;
                setTimeout(function() {
                    console.log(content);
                }, time);
            }
        })(i);
    }
});
```

Through this case, the local lyrics file song.txt is read through data transfer between modules. The specific module calling process is:

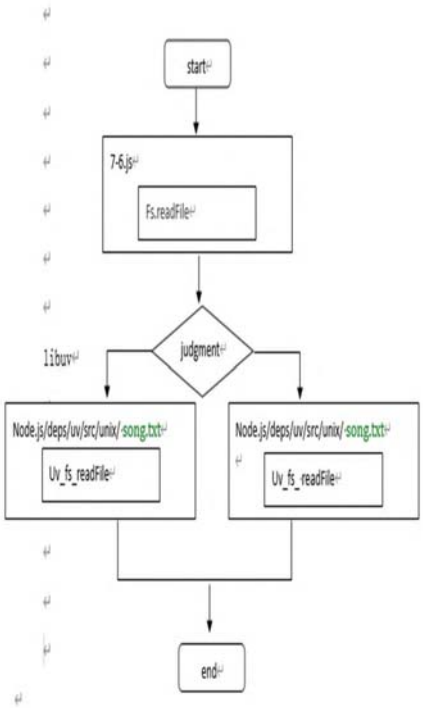

Figure 4 Flow chart of demo2 data calling module

The results of its operation are as follows:

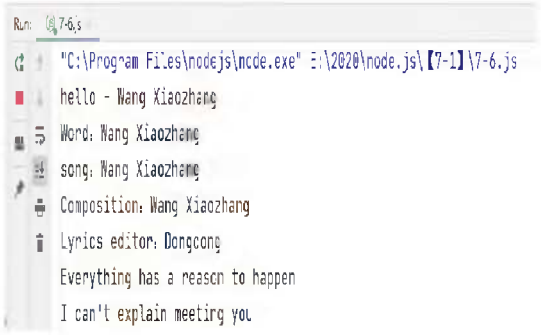

Figure 5 The result of calling the demo2 module

When we call Fs.readFile, Node.js calls the C/C++ level Read function through process.binding, and then calls the specific method uv_fs_readFile in Libuv through it, and the final execution result is passed back through a callback to complete the process.

IV. Event Loop

Node.js maintains an event queue in the main thread. When receiving a request, it puts the request into this queue as an event, and then continues to receive other requests. When the main thread is idle (when there is no request for access), it starts to circulate the event queue to check whether there are events to be processed in the queue. At this time, there are two situations: if it is a non-I/O task, it will be handled by itself and the callback will be passed The function returns to the upper call: if it is an I/O task, a thread is taken from the thread pool to handle the event, and the callback function is specified, and then the other events in the loop are continued.

Its operating principle is shown in the figure below:

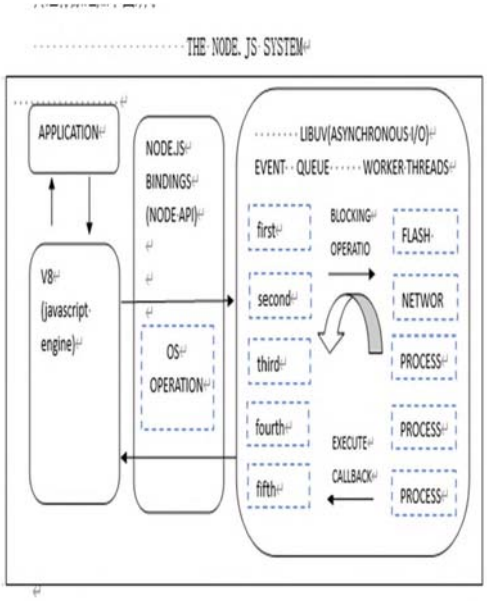

Figure 6 Node.js event loop mechanism principle

In the past, video playback required the Flash plug-in. But the instability of the Flash plug-in often causes the browser to crash, and now many browsers have eliminated flash. What replaces it is the HTML5 video element.

If you embed a video in the page, it can look like this:

According to the principle of the event loop mechanism, anonymous and functions can be encapsulated to form a "non-blocking asynchronous call" to complete multi-task event processing.

For example: demo3

```
const http=require('http');
console.log(1);
setTimeout(function () {
    console.log(2)
},1000)
console.log(3);
http.createServer(function (req,res) {

res.writeHead(200,{'content-type':'text/html;charset=utf-8'});
    console.log(req.url);
    res.write("你好")
    res.end();
}).listen(3005);
console.log('http://127.0.0.1:3005');
```

This case completed the non-blocking data processing of asynchronous data and synchronous data based on the principle of the event loop mechanism. The processing of synchronous events is generally faster than the processing of asynchronous events. The result of the operation should be read in order from

top to bottom. At the same time, when the asynchronous event is parsed, the asynchronous event will be put into the stack of the event loop. Here, it is sorted according to the processing time of the callback function. Therefore, the running results are as follows:

Figure 7 demo3 running results

V. CONCLUSION

In summary, Node.js, based on its own unique architectural features, can not only efficiently complete large-scale concurrent processing, but also complete the construction of high-quality servers. With the development of the information age, Web sites are no longer limited to the presentation of content. Many interactive and collaborative environments are gradually being added to the site, and this demand is still growing. This is the so-called data-intensive real applications, such as online collaborative whiteboards, multiplayer online games, etc. This kind of web application requires a platform that can respond to a large number of concurrent user requests in real time. This is where Node.js excels. . Of course, Node.js also has some shortcomings of its own, such as CUP-intensive applications and template rendering, compression/decompression, encryption/decryption, etc. are all the weaknesses of Node.js. But we will look forward to the further development and application of Node.js, and expect it to bring us more convenience and applications.

REFERENCES

[1] Zhang Zhaoyuan. A preliminary study on the back-end technology of web Node.js [J]. Small and Medium Enterprise Management and Technology 2020, Issue 22: 193-194.

[2] Wang Jijie. Design and implementation of a highly concurrent network application architecture based on Node.JS technology [J]. Journal of Tonghua Teachers College, 2020, Issue 7: 106-109

[3] Zhu Xiaoyang, Design and Implementation of Backend System of Learning Platform Based on Node.js[J]. Computer Knowledge and Technology: Academic Edition, 2019, Issue 5: 116-118.

[4] Zhou Anhui. Discussion on Node.js asynchronous programming mode [J]. Journal of Sichuan Vocational and Technical College, 2018 Issue 4: 149-154.