What Can Web Workers Do?

2016年3月9日
14:35

Contemplate tasks like the following:

•Encoding/decoding a large string
• Complex mathematical calculations (e.g., prime numbers, encryption, simulated annealing, etc.)
• Sorting a large array
• Network requests and resulting data processing
• Calculations and data manipulation on local storage
• Prefetching and/or caching data
• Code syntax highlighting or other real-time text analysis (e.g., spell checking)
• Image manipulation
• Analyzing or processing video or audio data (including face and voice recognition)
• Background I/O
• Polling web services
• Processing large arrays or huge JSON responses

What Web Workers Can and Can't Do

Can't do:
• The window object
• The document object
• The parent object
• And, last but not least, they can't use JavaScript libraries that depend on these objects to work, like jQuery.

Can Do:
• The navigator object
• The location object (read-only)
• The XMLHttpRequest function
• The atob() and btoa() functions for converting Base 64 ASCII to and from binary data
• setTimeout() / clearTimeout() and setInterval() / clearInterval()
• dump()
• The application cache
• External scripts using the importScripts() method
• Spawning other Web Workers4

Worker Execution

2016年3月9日
14:45

Web Workers threads run their code synchronously from top to bottom, and then enter
an asynchronous phase in which they respond to events and timers. This allows roughly
two types of Web Workers:

- Web Workers that register an onmessage event handler, for long-running tasks that need to run in the background. This Web Worker won't exit, as it keeps listening for new messages.
- Web Workers that never register for onmessage events, handling single tasks that need to be offset from the main web app thread, like fetching and parsing a massive JSON object. This Web Worker will exit once the operation is over.

How and where can we use web worker

2016年3月9日
14:55

Check for support

```
isWorkersAvailable() {
return !!window.Worker;
}

if (Modernizr.webworkers) {
// window.Worker is available!
} else {
// no native support for Web Workers
}
```

Loading External Scripts

```
importScripts('script1.js');
importScripts('script1.js', 'script2.js');


importScripts('http://twitter.com/statuses/user_timeline/' + user + '.json?count=10&callback=processTweets');
.
.
function processTweets(data) {
// parse the json object that holds the tweets and build a html block from
// their content.
.
.
}
```

Dedicated Web Workers

2016年3月9日
15:02

Dedicated Web Workers let you run scripts in background threads. Once the Web Worker is running, it can communicate with its web app by posting messages to an event handler registered with the web app that spawned it.

A dedicated Web Worker supports two events:

onmessage
Triggered when a message is received. An event object with a data member will be provided with the message.

onerror
Triggered when an error occurs in the Worker thread. The event provides a data member with the error information.

Index.html for fetching tweets and putting them in localStorage

```
<!DOCTYPE HTML>
<html>

<head>
    <meta charset="utf-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge,chrome=1">
</head>

<body>
    <article>
        <div id="result"></div>
        <div id="tweets"></div>
    </article>
    <script>
    console.log("WebWorker: Starting");
    var worker = new Worker("Example-3-2-tweet.js");
    worker.addEventListener("message", function(e) {
        var curTime = new Date();
        // here we will show the messages between our page and the Worker
        $('#result').append(curTime + " ) " + e.data + "<br/>");
        var source = e.data[0].source;
        // in case we have some data from Twitter - let's show it to the user
        if (typeof source != 'undefined') {
            var tweets = document.createElement("ul");
            for (var i = 0; i < 10; i++) {
                if (typeof e.data[i] != 'undefined' &&
                    e.data[i].text != 'undefined') {
                    var tweetTextItem = document.createElement("li");
                    var tweetText = document.createTextNode(e.data[i].text + " | " +
                        e.data[i].source + " (" +
                        e.data[i].created_at + ")");
                    tweetTextItem.appendChild(tweetText);
                    tweets.appendChild(tweetTextItem);
                    saveTweet(e.data[i]);
                }
            }
            // update the DOM outside our loop so it will be efficient
            console.log("WebWorker: Updated the DOM with Tweets");
            $("#tweets").append(tweets);
        }
    }, false);
    worker.onerror = function(e) {
        throw new Error(e.message + " (" + e.filename + ":" + e.lineno + ")");
    };
    // Key - tweet ID
    // Val - Time tweet created and the text of the tweet.
    function saveTweet(tweet) {
```

```
        localStorage.setItem(tweet.id_str, "{" +
            "'created': '" + tweet.created_at + "'," +
            "'tweet-text': '" + tweet.text + "'}");
    }
    // Get a tweet from our localStorage. We could use sessionStorage if we
    // wish to have this data just for our session
    function getTweet(tweetID) {
        return localStorage.getItem(tweetID);
    }
    </script>
</body>

</html>

// Example-3-2-tweet.js
// Pull Tweets and send them so the parent page could save them in the localStorage
var connections = 0; // count active connections
var updateDelay = 30000; // = 30sec delay
var user = "greenido";

function getURL(user) {
    return 'http://twitter.com/statuses/user_timeline/' + user + '.json?count=' + 12 + '&callback=processTweets';
}

function readTweets() {
    try {
        var url = getURL(user);
        postMessage("Worker Status: Attempting To Read Tweets for user - " + user +
            " from: " + url);
        importScripts(url);
    } catch (e) {
        postMessage("Worker Status: Error - " + e.message);
        setTimeout(readTweets, updateDelay);
    }
}

function processTweets(data) {
    var numTweets = data.length;
    if (numTweets > 0) {
        postMessage("Worker Status: New Tweets - " + numTweets);
        postMessage(data);
    } else {
        postMessage("Worker Status: New Tweets - 0");
    }
    setTimeout(readTweets, updateDelay);
}
//
// start the party in the Worker
//
readTweets();
```

Control Your Web Workers

2016年3月9日
15:15

// Index.html

```
<!DOCTYPE HTML>
<html>

<head>
    <title>Web Worker: The highest prime number</title>
    <!-- Get the latest jQuery code -->
    <script src="http://ajax.googleapis.com/ajax/libs/jquery/1/jquery.min.js">
</script>
    <meta charset=utf-8 />
</head>

<body>
    <h1>Web Worker: The highest prime number</h1>
    <article>The prime numbers:
        <output id="result"></output>
        <div id="status"></div>
    </article>
    <div id="actions">
        <input type="text" name="upto" id="upto" />
        <button onclick="start()" title="Start the work">Start</button>
        <button onclick="stop()" title="Stop the work and go have a drink">
Stop</button>
    </div>
    <script>
    var myWorker;

    function start() {
        console.log("WebWorker: Starting");
        myWorker = new Worker("highPrime2.js");
        myWorker.addEventListener("message", primeHandler, false);
        var maxNum = $('#upto').val();
        myWorker.postMessage({
            'cmd': 'start',
            'upto': maxNum
        });
    }

    function stop() {
        if (myWorker) {
            var msg = "<br/>WebWorker: Terminating " + new Date();
            console.log(msg);
            $('#status').append(msg);
            myWorker.terminate();
            myWorker = null;
        }
    }

    function primeHandler(event) {
        console.log('got e:' + event.data);
        if (is_numeric(event.data)) {
            $('#result').append(event.data);
        } else {
            $('#status').append(JSON.stringify(event.data));
        }
    }

    function is_numeric(input) {
        return typeof(input) == 'number';
    }
    </script>
</body>

</html>
```

// highPrime2.js

```
//
// A simple way to find prime numbers
// Please note the self refers to the Worker context inside the Worker.
self.addEventListener('message', function(e) {
    var data = e.data;
    var shouldRun = true;
    switch (data.cmd) {
        case 'stop':
            postMessage('Worker stopped the prime calculation (Al Gore is
happy now) ' +
                data.msg);
            shouldRun = false;
            self.close(); // Terminates the Worker.
            break;
        case 'start':
            postMessage("Worker start working upto: " + data.upto + " (" +
new Date() + ")<br/>");
            var numbers = isPrime(data.upto);
            postMessage("Got back these numbers: " + numbers + "<br/>");
            Control Your Web Workers | 19
            www.it - ebooks.info
            break;
        default:
            postMessage('Dude, unknown cmd: ' + data.msg);
    };
}, false);
// simple calculation of primes (not the most efficient - but works)
function isPrime(number) {
    var numArray = "";
    var thisNumber;
    var divisor;
    var notPrime;
    var thisNumber = 3;
    while (thisNumber < number) {
        var divisor = parseInt(thisNumber / 2);
        var notPrime = 0;
        while (divisor > 1) {
            if (thisNumber % divisor == 0) {
                notPrime = 1;
                divisor = 0;
            } else {
                divisor = divisor - 1;
            }
        }
        if (notPrime == 0) {
            numArray += (thisNumber + " ");
        }
        thisNumber = thisNumber + 1;
    }
    return numArray;
}
```

Parsing Data with Workers

2016年3月9日
15:23

Web Workers are great for handling long-running tasks. In modern web applications,there are many cases in which we need to handle large amounts of data. If you have a large JSON string you wish to parse and it will take ~250 milliseconds (or more), you should use Web Workers. This way, your users will love you and won't hate the fact that the web app doesn't feel responsive.

The following example shows how you can use a simple Web Worker to parse this string and get a nice JSON object you can work with in return.

Code in Main-web-app-page.html that works with a returned JSON object:

```
var worker = new Worker("worker-parser.js");
//when the data is fetched (e.g. in our xhr) -> this event handler
//is called to action
worker.onmessage = function(event) {
    //let's get our JSON structure back
    var jsonObj = event.data;
    //work with the JSON object
    showData(jsonObj);
};
//send the 'huge' JSON string to parse
worker.postMessage(jsonText);
```

worker-parser.js, a Web Worker that handles the actual JSON processing

```
self.onmessage = function(event) {
    //the JSON string comes in as event.data
    var jsonText = event.data;
    //parse the structure
    var jsonObj = JSON.parse(jsonText);
    //send back to the JSON obj.
    self.postMessage(jsonObj);
};
```

Transferable Objects

2016年3月9日
15:32

The option to use postMessage() not just for strings, but complex types like File, Blob, ArrayBuffer, and JSON objects, makes this an important enhancement. Structured cloning is a powerful algorithm for any web developer, but it's still a copy operation that can take hundreds of milliseconds

Chrome 17+ offers another performance boost through a new message-passing approach called Transferable Objects. This implementation makes sure that the data is transferred and not copied from one context to another. It is a "move" operation and not a copy, which vastly improves the performance of sending data to a Worker. It's similar to a pass-by-reference operation that we have in other languages. In a "normal" pass-by-reference we will have the same pointer to the data; however, here the "version" from the calling context is no longer available once the object is transferred to the new context. In other words, when we transfer an ArrayBuffer from our main web app page to the Web Worker, the original ArrayBuffer is cleared and we can no longer access it. Instead, its contents are transferred to the Worker context and are accessible only in the Web Worker's scope. There is a new (prefixed) version of postMessage() in Chrome 17+ that supports transferable objects. It takes two arguments, the ArrayBuffer message and a list of items that should be transferred:

```
worker.webkitPostMessage(arrayBuffer, [arrayBuffer]);
```

You can also send messages through the window object. This approach requires adding the targetOrigin because we can post this message to different workers.

```
window.webkitPostMessage(arrayBuffer, targetOrigin, [arrayBuffer]);
```

These approaches allow massive data manipulation, image processing, WebGL textures, etc., to be passed between the Web Worker and the main app with less impact on memory footprint and speed.

Inline Workers

2016年3月9日
15:35

There are cases in which you will want to create your Worker script “on the fly” in response to some event that your web app has fired. In other cases, you might want to have a self-contained page without having to create separate Worker files. Sometimes, you might wish to have your entire web app encapsulated in one page: you want to be able to fetch the app with one Ajax call, or bundle it as a Chrome extension. Inline Workers support these use cases. The example below shows how we can use the new BlobBuilder1 interface to inline your Worker code in the same HTML file.

```
//Creating an inline Worker with a javascript/worker type

<script id="worker1" type="javascript/worker">
    // This script won't be parsed by JS engines because its type is JavaScript/worker.
    // Simple code to calculate prime number and send it back to the parent page.
    self.onmessage = function(e) {
        self.postMessage("<h3>Worker: Started the calculation</h3><ul>");

        var n = 1;
        search: while (n < 500) {
            n += 1;
            for (var i = 2; i <= Math.sqrt(n); i += 1)
            if (n % i == 0)
            continue search;
            // found a prime!
            postMessage("<li>Worker: Found another prime: " + n + "</li>");
        }

        postMessage("</ul><h3>Worker: Done</h3>");
    }
</script>

//Creating a Worker using BlobBuilder

<script>
    // Creating the BlobBuilder and adding our Web Worker code to it.
    var bb = new(window.BlobBuilder || window.WebKitBlobBuilder ||
        window.MozBlobBuilder)();
    bb.append(document.querySelector('#worker1').textContent);

    // Creates a simple URL string which can be used to reference
    // data stored in a DOM File / Blob object.
    // In Chrome, there's a nice page to view all of the created
    // blob URLs: chrome://blob-internals/
    // OurUrl enable our code to run in Chrome and Firefox.
    var ourUrl = window.webkitURL || window.URL;
    var worker = new Worker(ourUrl.createObjectURL(bb.getBlob()));

    worker.onmessage = function(e) {
        status(e.data);
    }

    worker.postMessage();
</script>
```

Shared Workers

2016年3月9日
15:44

Other good uses for shared Workers include the following:

• Providing a single source of truth for any type of logic that your app needs in more
then one place (e.g., user identification, connection status, etc.).
• Ensuring data consistency between windows of the same web app.
• Reducing the memory consumption of multiple web app tabs/windows, by allowing some code (e.g., server communications) to be centralized in one place.

// SharedWorker1.html

```
<!DOCTYPE HTML>
<html>

<head>
    <meta charset="utf-8" />
    <title>Shared Web Workers: Basic Example</title>
</head>

<body>
    <h1>Shared Web Workers: Basic Example</h1>
    <article>
        To create a shared Web Worker, you pass a JavaScript file name to a new instance of the SharedWorker object:
        <br/>var worker = new SharedWorker("jsworker.js");
        <br/> Our web shared Web Worker will count the connection and return the data back to our listener in this page. You might want to
open the Chrome DevTools to see the process.
        <output id="result"></output>
    </article>
    <script>
    var worker = new SharedWorker('sharedWorker1.js');
    worker.port.addEventListener("message", function(e) {
        document.getElementById('result').textContent += " | " + e.data;
    }, false);
    worker.port.start();
    // post a message to the shared Web Worker
    console.log("Calling the worker from script section 1");
    worker.port.postMessage("script-1");
    </script>

    <script>
    // This new script block might be found on a separate tab/window
    // of our web app. Here it's just for the example on the same page.
    console.log("Calling the worker from script section 2");
    worker.port.postMessage("script-2");
    </script>
</body>

</html>
```

// sharedWorker1.js

```
// Shared workers that handle the connections and Welcome each new script
var connections = 0; // count active connections
self.addEventListener("connect", function(e) {
    var port = e.ports[0];
    connections++;
    port.addEventListener("message", function(e) {
        port.postMessage("Welcome to " + e.data +
            " (On port #" + connections + ")");
    }, false);
    //
    port.start();
}, false);
```

Shared Workers - Demo

2016年3月9日
16:03

// SharedWorkers-1.html

```
<!DOCTYPE HTML>
<html>

<head>
    <meta charset="utf-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge,chrome=1">
    <title>Shared Web Workers: Twitter Example</title>
    <meta name="author" content="Ido Green">
    <script src="http://ajax.googleapis.com/ajax/libs/jquery/1/jquery.min.js"></script>
    <style>
    #result {
        background: lightblue;
        padding: 20px;
        border-radius: 18px;
    }

    #tweets {
        background: yellow;
        border-radius: 28px;
        padding: 20px;
    }
    </style>
</head>

<body>
    <nav>
        <button id="start-button">Start The Shared Worker</button>
        <button id="stop-button">Stop The Shared Worker</button>
    </nav>
    <article>
        <div id="result"></div>
        <div id="tweets"></div>
    </article>

    <script>
    var worker;

    function startWorker() {
        console.log("WebWorker: Starting");
        worker = new SharedWorker("sharedWorker2.js");
        worker.port.addEventListener("message", function(e) {
                var curTime = new Date();
                // here we will show the messages between our page and the shared Worker
                $('#result').append(curTime + " ) " + e.data + "<br/>");
                var source = e.data[0].source;
                // in case we have some data from Twitter - let's show it to the user
                if (typeof source != 'undefined') {
                    var tweets = document.createElement("ul");
                    for (var i = 0; i < 10; i++) {
                        if (typeof e.data[i] != 'undefined' &&
                            e.data[i].text != 'undefined') {
                            var tweetTextItem = document.createElement("li");
                            var tweetText = document.createTextNode(e.data[i].text + " | " +
                                e.data[i].source + " (" +
                                e.data[i].created_at + ")");
```

```
                        tweetTextItem.appendChild(tweetText);
                        tweets.appendChild(tweetTextItem);
                    }
                    // update the DOM outside our loop so it will be efficient action
                    console.log("WebWorker: Updated the DOM with Tweets");
                    $("#tweets").append(tweets);
                }
                // just to help us analyze what we got as data form the shared Worker
                console.log("msg we got back: " + JSON.stringify(e));
            }, false); worker.onerror = function(e) {
            throw new Error(e.message + " (" + e.filename + ":" + e.lineno + ")");
        }; worker.port.start();
        // post a message to the shared Web Worker
        console.log("Calling the worker with @greenido as user"); worker.port.postMessage({
            cmd: "start",
            user: "greenido"
        });
    }

    function stopWorker() {
        if (worker != undefined) {
            worker.port.postMessage({
                cmd: "stop"
            });
            console.log("WebWorker: Stop the party");
            // You might use worker = null if you wish not to use the Worker from now
        }
    }
    // when the DOM is ready - attached our 2 actions to the buttons
    $(function() {
        $('#start-button').click(function() {
            startWorker();
        });
        $('#stop-button').click(function() {
            stopWorker();
        });
    });
</script>
</body>

</html>
```

//SharedWorkers2.html

```
<!DOCTYPE HTML>
<html>

<head>
    <meta charset="utf-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge,chrome=1">
    <title>Shared Web Workers: Twitter Example</title>
    <meta name="author" content="Ido Green">
    <script src="http://ajax.googleapis.com/ajax/libs/jquery/1/jquery.min.js"></script>
</head>

<body>
    <nav>
        <button id="start-button">Start The Shared Worker</button>
        <button id="stop-button">Stop The Shared Worker</button>
    </nav>
    <article>
```

```
    <div id="result"></div>
    <div id="tweets"></div>
</article>
<script>
var worker;

function startWorker() {
    console.log("WebWorker: Starting");
    worker = new SharedWorker("Example-5-4-sharedWorkerTweet.js");
    worker.port.addEventListener("message", function(e) {
        var curTime = new Date();
        // here we will show the messages between our page and the shared Worker
        $('#result').append(curTime + " ) " + e.data + "<br/>");
        var source = e.data[0].source;
        // in case we have some data from Twitter - let's show it to the user
        if (typeof source != 'undefined') {
            var tweets = document.createElement("ul");
            for (var i = 0; i < 10; i++) {
                if (typeof e.data[i] != 'undefined' &&
                    e.data[i].text != 'undefined') {
                    var tweetTextItem = document.createElement("li");
                    var tweetText = document.createTextNode(e.data[i].text + " | " +
                        e.data[i].source + " (" +
                        e.data[i].created_at + ")");
                    tweetTextItem.appendChild(tweetText);
                    tweets.appendChild(tweetTextItem);
                }
            }
            // update the DOM outside our loop so it will be efficient action
            console.log("WebWorker: Updated the DOM with Tweets");
            $("#tweets").append(tweets);
        }
    }, false);
    worker.onerror = function(e) {
        throw new Error(e.message + " (" + e.filename + ":" + e.lineno + ")");
    };
    worker.port.start();
    // post a message to the shared Web Worker
    console.log("Calling the worker with @greenido as user");
    worker.port.postMessage({
        cmd: "start",
        user: "greenido"
    });
}

function stopWorker() {
    if (worker != undefined) {
        worker.port.postMessage({
            cmd: "stop"
        });
        console.log("WebWorker: Stop the party");
        // You might use worker = null if you wish not to use the Worker from now
    }
}
// when the DOM is ready - attached our 2 actions to the buttons
$(function() {
    $('#start-button').click(function() {
        startWorker();
    });
    $('#stop-button').click(function() {
        stopWorker();
```

```
        });
    });
    </script>
</body>

</html>
```

// sharedWorker2.js

```
//
// Shared workers that handle the connections and Welcome each new script
// @author Ido Green
// @date 11/11/2011
var connections = 0; // count active connections
var updateDelay = 60000; // = 1min delay
var port;
var user;

function getURL(user) {
    return 'http://twitter.com/statuses/user_timeline/' + user + '.json?count=' + 12 + '&callback=processTweets';
}

function readTweets() {
    try {
        var url = getURL(user);
        port.postMessage("Worker: Attempting To Read Tweets for user - " + user +
            " from: " + url);
        importScripts(url);
    } catch (e) {
        port.postMessage("Worker: Error - " + e.message);
        setTimeout(readTweets, updateDelay); // lets do it every 2min
    }
}

function processTweets(data) {
    if (data.length > 0) {
        port.postMessage("Worker: New Tweets - " + data.length);
        port.postMessage(data);
    } else {
        port.postMessage("Worker: New Tweets - 0");
    }
    setTimeout(readTweets, updateDelay);
}
//
// The controller that manage the actions/commands/connections
//
self.addEventListener("connect", function(e) {
    port = e.ports[0];
    connections++;
    port.addEventListener("message", function(e) {
        var data = e.data;
        switch (data.cmd) {
            case 'start':
                port.postMessage("Worker: Starting You are connection number:" + connections);
                user = data.user;
                readTweets();
                break;
            case 'stop':
                port.postMessage("Worker: Stopping");
                self.close();
```

```
                break;
            default:
                port.postMessage("Worker: Error - Unknown Command");
        };
    }, false);
    port.start();
}, false);
```

Debug Your Workers

2016年3月9日
16:11

If you aren’t using Chrome, there is an option to gain information when an error occurs.The Web Workers specification1 shows us that an error event (onerror handler) shouldbe fired when a runtime script error occurs in a Worker. The main properties in the onerror handler are the following:

message
    The error message itself.
lineno
    The number of the line inside our Web Worker that caused the error.
filename
    The name of the file inside the Worker in which the error occurred.

You can override the onerror function with a version that will throw an error withenough information to help us see what’s happening inside the Worker

//This is a simple way to understandwhere your code is broken.

```
var worker = new Worker("worker.js");
worker.onerror = function(e){
    throw new Error(e.message + " (" + e.filename + ":" + e.lineno + ")" );
};
```

//Debugging in Chrome Dev Tools

chrome://inspect/#workers

Web workers beyond the browser: node

2016年3月9日
16:28

node-webworker module

// main.js

```
var sys = require('sys');

// fetching node-webworker
var Worker = require('webworker');

// create a new worker to calculate routes
var w = new Worker('routes-worker.js');

// listen to messages from the Worker and in our case kill it when we get the first message
(with or without the calculated route w.onmessage = function(e) {
        sys.debug('* Got mesage: ' + sys.inspect(e));
        w.terminate();
    };

 // ask the Worker to run on a 'test' route from L.A. to San Francisco
 w.postMessage({ route: 'lax-sfo'
});
```

// routes-worker.js

```
onmessage = function(data) {
    // calculating the route here
    // ...
    postMessage({ route: 'json obj with the route details' });
};
onclose = function() {
    sys.debug('route-worker shutting down.');
};
```