Web Worker-Grundlagen

By Eric Bidelman

Veröffentlicht: Juli 26th, 2010
Comments: 64

Das Problem: Nebenläufigkeit von JavaScript

Es gibt mehrere Hindernisse, die eine Übertragung interessanter Anwendungen von serverlastigen Implementierungen in clientseitige JavaScript-Umgebungen verhindern. Dazu gehören unter anderem Browserkompatibilität, statische Typisierung, Zugriff und Leistung. Glücklicherweise gehört Letzteres zunehmend der Vergangenheit an, da die Browser-Anbieter die Geschwindigkeit ihrer JavaScript-Engines konstant verbessern.

Ein Aspekt, der weiterhin eine Hürde für JavaScript darstellt, ist die Sprache selbst. JavaScript ist eine Single-Threaded-Umgebung, es können also nicht gleichzeitig mehrere Skripts ausgeführt werden. Stellen Sie sich zum Beispiel eine Website vor, die Benutzeroberflächenereignisse abwickeln, große Mengen von API-Daten abfragen und verarbeiten und das DOM manipulieren muss. Nicht ungewöhnlich, oder? Leider lassen sich diese Prozesse aufgrund von Beschränkungen bei der JavaScript-Laufzeit von Browsern nicht parallel ausführen. Die Skript-Ausführung erfolgt in einem einzigen Thread.

Mithilfe von Techniken wie `setTimeout()`, `setInterval()`, `XMLHttpRequest` und Ereignis-Handlern ahmen Entwickler eine Nebenläufigkeit nach. Ja, all diese Funktionen werden asynchron ausgeführt, aber blockierungsfrei bedeutet nicht gleichzeitig parallel. Asynchrone Ereignisse werden verarbeitet, nachdem das aktuelle Ausführungsskript ausgesetzt wurde. Die gute Nachricht ist, dass HTML5 bessere Möglichkeiten bietet!

Web Worker-Einführung: Threading für JavaScript

Die Web Worker-Spezifikation legt ein API fest, das Hintergrund-Skripts in Ihrer

Webanwendung erzeugt. Mit Web Workern können Sie zum Beispiel lange Skripts auslösen, die rechenintensive Aufgaben abwickeln, ohne dabei die Benutzeroberfläche oder andere Skripts daran zu hindern, Interaktionen von Nutzern zu verarbeiten. Sie schaffen Abhilfe und setzen diesem lästigen Dialogfeld, das auf ein nicht reagierendes Skript hinweist und das wir alle lieben, ein Ende:

Häufig angezeigtes Dialogfeld mit Hinweis auf nicht reagierendes Skript

Web Worker machen sich eine threadähnliche Nachrichtenweitergabe zunutze, um Parallelität zu erzielen. So profitieren Ihre Nutzer jederzeit von einer aktuellen, leistungsstarken und reaktionsschnellen Benutzeroberfläche.

Web Worker-Arten

Es ist erwähnenswert, dass die Spezifikation zwei Web Worker-Arten behandelt: Dedicated Worker und Shared Worker. In diesem Artikel befassen wir uns lediglich mit Dedicated Workern, wobei ich jeweils von "Web Worker" oder "Worker" spreche.

Erste Schritte

Web Worker werden in einem isolierten Thread ausgeführt. Aus diesem Grund muss der Code, den sie ausführen, in einer separaten Datei enthalten sein. Bevor wir jedoch dazu kommen, müssen Sie zunächst ein neues `Worker`-Objekt auf Ihrer Hauptseite erstellen. Der Konstruktor trägt den Namen des Worker-Skripts:

```
var worker = new Worker('task.js');
```

Ist die angegebene Datei vorhanden, erzeugt der Browser einen neuen Worker-Thread, der asynchron heruntergeladen wird. Der Worker beginnt erst, wenn die Datei vollständig heruntergeladen und ausgeführt wurde. Falls der Pfad zu Ihrem Worker den Fehler 404 zurückgibt, schlägt der Worker ohne weitere

Fehlermeldung fehl.

Nach der Erstellung des Workers starten Sie ihn mithilfe der `postMessage()`-Methode:

```
worker.postMessage(); // Start the worker.
```

Mit einem Worker via Nachrichtenweitergabe kommunizieren

Die Kommunikation zwischen Worker und übergeordneter Seite erfolgt auf der Grundlage eines Ereignismodells und der `postMessage()`-Methode. Je nach Browser bzw. Version akzeptiert die `postMessage()`-Methode entweder eine Zeichenfolge oder ein JSON-Objekt als einfaches Argument. Die neuesten Versionen der modernen Browser unterstützen die Weitergabe eines JSON-Objekts.

Im Folgenden finden Sie ein Beispiel, bei dem mithilfe einer Zeichenfolge "Hello World" an einen Worker in doWork.js weitergegeben wird. Der Worker gibt einfach die Nachricht zurück, die er erhalten hat.

Hauptskript:

```
var worker = new Worker('doWork.js');

worker.addEventListener('message', function(e) {
  console.log('Worker said: ', e.data);
}, false);

worker.postMessage('Hello World'); // Send data to our worker.
```

doWork.js (der Worker):

```
self.addEventListener('message', function(e) {
  self.postMessage(e.data);
}, false);
```

Wenn `postMessage()` über die Hauptseite aufgerufen wird, verarbeitet unser Worker diese Nachricht durch Festlegung eines `onmessage`-Handlers für das `message`-Ereignis. Die Nutzdaten der Nachricht, in diesem Fall "Hello World", sind in `Event.data` verfügbar. Obwohl das genannte Beispiel nicht besonders

spannend ist, so zeigt es dennoch, dass `postMessage()` auch eine Möglichkeit darstellt, um Daten zurück an den Haupt-Thread zu übermitteln. Praktisch!

Nachrichten, die zwischen der Hauptseite und den Workern ausgetauscht werden, werden kopiert, nicht gemeinsam genutzt. Zum Beispiel ist die Eigenschaft "msg" der JSON-Nachricht im nächsten Beispiel an beiden Orten einsehbar. Es scheint, als würde das Objekt direkt an den Worker weitergegeben, obwohl es in einem separaten, dafür vorgesehenen Bereich ausgeführt wird. Tatsächlich wird das Objekt serialisiert, wenn es an den Worker übergeben wird. Anschließend wird die Serialisierung am anderen Ende wieder aufgehoben. Seite und Worker teilen sich nicht die gleiche Instanz, sodass bei jeder Weitergabe ein Duplikat erstellt wird. Die meisten Browser implementieren diese Funktion, indem Sie an beiden Enden automatisch eine JSON-Codierung/-Decodierung des Werts vornehmen.

Das folgende Beispiel ist etwas komplexer. Darin werden Nachrichten mithilfe von JSON-Objekten weitergegeben.

Hauptskript:

```
<button onclick="sayHI()">Say HI</button>
<button onclick="unknownCmd()">Send unknown command</button>
<button onclick="stop()">Stop worker</button>
<output id="result"></output>

<script>
  function sayHI() {
    worker.postMessage({'cmd': 'start', 'msg': 'Hi'});
  }

  function stop() {
    // Calling worker.terminate() from this script would also
stop the worker.
    worker.postMessage({'cmd': 'stop', 'msg': 'Bye'});
  }

  function unknownCmd() {
    worker.postMessage({'cmd': 'foobard', 'msg': '???'});
  }

  var worker = new Worker('doWork2.js');

  worker.addEventListener('message', function(e) {
    document.getElementById('result').textContent = e.data;
  }, false);
</script>
```

doWork2.js:

```
self.addEventListener('message', function(e) {
  var data = e.data;
  switch (data.cmd) {
    case 'start':
      self.postMessage('WORKER STARTED: ' + data.msg);
      break;
    case 'stop':
      self.postMessage('WORKER STOPPED: ' + data.msg + '.
(buttons will no longer work)');
      self.close(); // Terminates the worker.
      break;
    default:
      self.postMessage('Unknown command: ' + data.msg);
  };
}, false);
```

Hinweis: Ein Worker kann auf zwei Arten beendet werden: durch Aufrufen von `worker.terminate()` über die Hauptseite oder durch Aufrufen von `self.close()` im Worker selbst.

Beispiel: Führen Sie diesen Worker aus!

Hallo sagen | Unbekannten Befehl senden | Worker beenden

Die Worker-Umgebung

Geltungsbereich des Workers

Im Zusammenhang mit Workern weisen `self` und `this` auf den globalen Geltungsbereich des Workers hin. Daher könnte das vorherige Beispiel auch wie folgt geschrieben werden:

```
addEventListener('message', function(e) {
  var data = e.data;
  switch (data.cmd) {
    case 'start':
      postMessage('WORKER STARTED: ' + data.msg);
      break;
    case 'stop':
  ...
}, false);
```

Alternativ könnten Sie den Ereignis-Handler `onmessage` direkt festlegen, obwohl `addEventListener` von JavaScript-Ninjas empfohlen wird.

```
onmessage = function(e) {
  var data = e.data;
  ...
};
```

Für Worker verfügbare Funktionen

Aufgrund ihres Multithread-Verhaltens können Web Worker nur auf einen Teil der Funktionen von JavaScript zugreifen:

- Das `navigator`-Objekt
- Das `location`-Objekt (schreibgeschützt)
- `XMLHttpRequest`
- `setTimeout()/clearTimeout()` und `setInterval()/clearInterval()`
- Den Anwendungscache
- Import externer Skripts mithilfe der `importScripts()`-Methode
- Erzeugen weiterer Web Worker

Worker haben KEINEN Zugriff auf:

- Das DOM (nicht threadsicher)
- Das `window`-Objekt
- Das `document`-Objekt
- Das `parent`-Objekt

Externe Skripts laden

Mithilfe der `importScripts()`-Funktion können Sie externe Skriptdateien oder -bibliotheken in einen Worker laden. Bei dieser Methode können null oder mehr Zeichenfolgen als Dateinamen der zu importierenden Ressourcen angegeben werden.

Bei diesem Beispiel werden `script1.js` und `script2.js` in den Worker geladen:

worker.js:

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

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

Dies lässt sich auch in einer Importanweisung zusammenfassen:

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

Untergeordnete Worker

Worker sind in der Lage, untergeordnete Worker zu erzeugen. So lassen sich große Aufgaben während der Laufzeit noch weiter unterteilen. Untergeordnete Worker haben jedoch auch einige Nachteile:

- Beim Hosten untergeordneter Worker müssen Sie sich an der übergeordneten Seite orientieren.
- URIs in untergeordneten Workern werden in Abhängigkeit vom Speicherort des übergeordneten Workers aufgelöst (im Gegensatz zur Hauptseite).

Beachten Sie, dass die meisten Browser für jeden Worker separate Prozesse erzeugen. Bevor Sie eine Worker-Farm erzeugen, vergewissern Sie sich, dass Sie nicht zu viele Systemressourcen des Nutzers beanspruchen. Dies könnte unter anderem der Fall sein, weil Nachrichten, die zwischen der Hauptseite und den Workern ausgetauscht werden, kopiert und nicht gemeinsam genutzt werden. Weitere Informationen finden Sie unter Mit einem Worker via Nachrichtenweitergabe kommunizieren.

Ein Beispiel für die Erstellung eines untergeordneten Workers finden Sie in der Spezifikation.

Inline-Worker

Nehmen wir an, Sie möchten Ihr Worker-Skript spontan erstellen oder eine eigenständige Seite erzeugen, ohne separate Worker-Dateien erstellen zu müssen. Mit der neuen `BlobBuilder`-Oberfläche können Sie Ihren Worker in die gleiche HTML-Datei wir Ihre Hauptlogik "einbetten". Dazu erstellen Sie einen `BlobBuilder` und hängen den Worker-Code als Zeichenfolge an:

```
// Prefixed in Webkit, Chrome 12, and FF6:
window.WebKitBlobBuilder, window.MozBlobBuilder
var bb = new BlobBuilder();
bb.append("onmessage = function(e) { postMessage('msg from
```

```
worker'); }");

// Obtain a blob URL reference to our worker 'file'.
// Note: window.webkitURL.createObjectURL() in Chrome 10+.
var blobURL = window.URL.createObjectURL(bb.getBlob());

var worker = new Worker(blobURL);
worker.onmessage = function(e) {
  // e.data == 'msg from worker'
};
worker.postMessage(); // Start the worker.
```

Blob-URLs

Der eigentliche Zauber beginnt mit dem Aufruf an `window.URL.createObjectURL()`. Bei dieser Methode wird eine einfache URL-Zeichenfolge erstellt, mit der auf Daten verwiesen werden kann, die in einem DOM `File`- oder `Blob`-Objekt gespeichert sind. Zum Beispiel:

```
blob:http://localhost/c745ef73-ece9-46da-8f66-ebes574789b1
```

Blob-URLs sind eindeutig und behalten für die Lebensdauer Ihrer Anwendung ihre Gültigkeit, also bis `document` entladen wird. Wenn Sie viele Blob-URLs erstellen, ist es ratsam, sich von nicht mehr benötigten Referenzen zu trennen. Um sich von einer Blob-URL zu trennen, geben Sie sie an `window.URL.revokeObjectURL()` weiter:

```
window.URL.revokeObjectURL(blobURL); //
window.webkitURL.createObjectURL() in Chrome 10+.
```

In Chrome finden Sie eine Übersicht mit allen erstellten Blob-URLs: `chrome://blob-internals/`.

Vollständiges Beispiel

Wenn wir einen Schritt weitergehen, können wir sehen, wie der JS-Code des Workers in unsere Seite eingebettet ist. Bei dieser Technik wird der Worker mithilfe des `<script>`-Tags definiert:

```
<!DOCTYPE html>
<html>
```

```
<head>
  <meta charset="utf-8" />
</head>
<body>

  <div id="log"></div>

  <script id="worker1" type="javascript/worker">
    // This script won't be parsed by JS engines because its
type is javascript/worker.
    self.onmessage = function(e) {
      self.postMessage('msg from worker');
    };
    // Rest of your worker code goes here.
  </script>

  <script>
    function log(msg) {
      // Use a fragment: browser will only render/reflow once.
      var fragment = document.createDocumentFragment();
      fragment.appendChild(document.createTextNode(msg));
      fragment.appendChild(document.createElement('br'));

      document.querySelector("#log").appendChild(fragment);
    }

    var bb = new BlobBuilder();
    bb.append(document.querySelector('#worker1').textContent);

    // Note: window.webkitURL.createObjectURL() in Chrome 10+.
    var worker = new
Worker(window.URL.createObjectURL(bb.getBlob()));
    worker.onmessage = function(e) {
      log("Received: " + e.data);
    }
    worker.postMessage(); // Start the worker.
  </script>
</body>
</html>
```

Meiner Ansicht nach ist dieser neue Ansatz etwas geradliniger und leserlicher. Dabei wird ein Skript-Tag mit *id="worker1"* und *type='javascript/worker'* definiert, sodass der Browser nicht das JS parst. Dieser Code wird mithilfe von `document.querySelector('#worker1').textContent` als Zeichenfolge extrahiert und an `BlobBuilder.append()` weitergegeben.

Externe Skripts laden

Bei Verwendung dieser Techniken zum Einbetten Ihres Worker-Codes funktioniert `importScripts()` nur, wenn Sie eine absolute URI angeben. Bei der versuchten Weitergabe einer relativen URI gibt der Browser einen Sicherheitsfehler zurück. Der Grund: Der Worker, der nun aus einer Blob-URL erstellt wird, wird mit einem `blob:`-Prefix aufgelöst, während Ihre App über ein anderes Schema ausgeführt wird, vermutlich `http://`. Der Fehler wird also durch ursprungsüberschreitende Beschränkungen hervorgerufen.

Eine Möglichkeit zur Verwendung von `importScripts()` in einem Inline-Worker besteht darin, die aktuelle URL, über die Ihr Hauptskript ausgeführt wird, einzuspeisen, indem Sie sie an den Inline-Worker weitergeben und die absolute URL manuell erzeugen. So stellen Sie sicher, dass das externe Skript von gleicher Stelle importiert wird. Nehmen wir an, Ihre Haupt-App wird von http://example.com/index.html ausgeführt:

```
...
<script id="worker2" type="javascript/worker">
self.onmessage = function(e) {
  var data = e.data;

  if (data.url) {
    var url = data.url.href;
    var index = url.indexOf('index.html');
    if (index != -1) {
      url = url.substring(0, index);
    }
    importScripts(url + 'engine.js');
  }
  ...
};
</script>
<script>
  var worker = new
Worker(window.URL.createObjectURL(bb.getBlob()));
  worker.postMessage({url: document.location});
</script>
```

Fehlerbehebung

Wie bei jeder JavaScript-Logik möchten Sie sicherlich zurückgegebene Fehler in Ihren Web Workern behandeln. Wenn beim Ausführen eines Workers ein Fehler auftritt, wird ein `ErrorEvent` ausgelöst. Die Oberfläche liefert drei nützliche Eigenschaften, mit deren Hilfe Sie der Sache auf den Grund gehen können: `filename` - der Name des Worker-Skripts, das den Fehler verursacht hat,

`lineno` - die Nummer der Zeile, in der der Fehler aufgetreten ist, und `message` - eine nützliche Beschreibung des Fehlers. In diesem Beispiel zeige ich Ihnen, wie Sie einen `onerror`-Ereignis-Handler einrichten, um die Fehlereigenschaften auszudrucken:

```
<output id="error" style="color: red;"></output>
<output id="result"></output>

<script>
  function onError(e) {
    document.getElementById('error').textContent = [
      'ERROR: Line ', e.lineno, ' in ', e.filename, ': ',
e.message].join('');
  }

  function onMsg(e) {
    document.getElementById('result').textContent = e.data;
  }

  var worker = new Worker('workerWithError.js');
  worker.addEventListener('message', onMsg, false);
  worker.addEventListener('error', onError, false);
  worker.postMessage(); // Start worker without a message.
</script>
```

Beispiel: workerWithError.js versucht, 1/x durchzuführen, wobei x nicht definiert ist.

Ausführen

workerWithError.js:

```
self.addEventListener('message', function(e) {
  postMessage(1/x); // Intentional error.
};
```

Ein Wort zum Thema Sicherheit

Lokale Zugriffsbeschränkungen

Aufgrund der Sicherheitsbeschränkungen von Google Chrome werden Worker in den neuesten Versionen des Browsers nicht lokal ausgeführt, zum Beispiel über `file://`. Stattdessen schlagen Sie ohne weitere Fehlermeldung fehl! Um

Ihre App über das `file://`-Schema auszuführen, führen Sie Chrome unter Angabe von `--allow-file-access-from-files` aus. **Hinweis:** Es wird nicht empfohlen, Ihren primären Browser mit dieser Einstellung auszuführen. Diese sollte nur zu Testzwecken und nicht während des regulären Surfvorgangs verwendet werden.

Bei anderen Browsern ist eine solche Beschränkung nicht vorgesehen.

Überlegungen zum gleichen Ursprung

Bei Worker-Skripts muss es sich um externe Dateien handeln, die das gleiche Schema wie die aufrufende Seite aufweisen. Daher können Sie ein Skript nicht von einer `data:`- oder `javascript:`-URL laden und eine `https:`-Seite kann keine Worker-Skripts starten, die mit `http:`-URLs beginnen.

Anwendungsfälle

Welche Arten von Apps würden also Web Worker verwenden? Leider sind Web Worker relativ neu und die Mehrzahl der vorhandenen Beispiele/Anleitungen beinhaltet die Berechnung von Primzahlen. Obwohl dies eher weniger interessant ist, lassen sich die Konzepte von Web Workern damit besser verstehen. Im Folgenden habe ich einige weitere Ideen zusammengestellt, mit denen Sie Ihre grauen Zellen ein wenig anstrengen können:

- Daten für die spätere Verwendung im Voraus abrufen und/oder im Zwischenspeicher ablegen
- Syntaxhervorhebung oder andere Möglichkeiten zur Textformatierung in Echtzeit
- Rechtschreibprüfung
- Analyse von Video- oder Audiodaten
- Hintergrund-E/A oder Abrufen von Webdiensten
- Verarbeitung großer Arrays oder riesiger JSON-Antworten
- Filtern von Bildern in <Canvas>
- Aktualisierung einer großen Anzahl von Zeilen in einer lokalen Webdatenbank

Demos

- Beispiele aus HTML5Rocks-Präsentationen
- Motion-Tracker

- Simulierte Abkühlung
- HTML5demos-Beispiel

Referenzen

- Web Worker-Spezifikation
- "Using web workers" aus dem Mozilla Developer Network
- "Web Workers rise up!" aus Dev.Opera