Guida HTML5: Web Worker

In questo articolo della Guida HTML5 parleremo di una nuova funzionalità che fornisce ai programmatori sia più potenza di calcolo che una soluzione ad alcuni problemi causati dall'esecuzione di codice JavaScript asincrono.

Si tratta dei Web Worker, che consentono di eseguire il codice contenuto in un file javascript in background. Il linguaggio JavaScript, nato su computer equipaggiati con processori single-core, veniva (e viene tuttora) eseguito in un unico thread (flusso). Al giorno d'oggi però i computer montano processori multi-core, e grazie ai Web Worker possiamo finalmente sfruttare i thread forniti da questi core aggiuntivi.

Ma perché è importante usare thread paralleli? Immaginate di avere due parti di codice che si trovano a dover essere eseguite contemporaneamente perché richiamate in maniera asincrona (tramite ad esempio *XMLHttpRequest, setTimeout, Promise, FileReader*, ecc). Se si dispone di un unico thread, queste due parti si troveranno in **concorrenza**: si tratta in pratica di un collo di bottiglia dove una delle due parti deve aspettare che l'esecuzione dell'altra venga completata prima di poter essere a sua volta eseguita. **Un Web Worker offre invece un thread aggiuntivo, in modo che nessuna parte di codice debba aspettare**.

Si può pensare ai Web Worker anche come ad una "forza lavoro" aggiuntiva, da usare quando si tratta di fare calcoli pesanti: pensate alle immagini, dove ogni pixel di ogni immagine o frame corrisponde a quattro elementi di un array: ecco che manipolare un'immagine 640x360 anche se pesa pochi KB significa gestire un array di quasi un milione di elementi. Ovviamente si possono creare più web worker, ma è bene non crearne di più del numero di core logici della CPU del visitatore, pena il rischio di ricadere in una situazione di concorrenza. Questo numero è ottenibile tramite navigator.hardwareConcurrency, valore che potrebbe variare ad ogni interrogazione ed essere più basso del reale in base ad una stima delle effettive risorse disponibili effettuata dal browser. Tale indeterminazione, oltre a portare ad un'implementazione più realistica, è quantomai opportuna in ambito privacy, in quanto evita che questo parametro possa venire usato per il device fingerprinting. Qui trovate il polyfill più valido esistente, ma che purtroppo per forza di cose non è molto preciso.

I motivi più comuni per la quale si decide di ricorrere ai Web Worker sono evitare il rallentamento o blocco momentaneo dell'interfaccia utente, oppure l'esecuzione di task in background, o ancora l'esecuzione periodica di un polling. Problemi di performance possono derivare anche da casi meno comuni dove è richiesta una grossa potenza di calcolo, come la generazione in tempo reale di file PDF o ZIP, la manipolazione di contenuti multimediali, il pathfinding su una mappa stradale, il voler rendere fruibile l'applicazione su device poco potenti, eccetera. Infine, c'è la semplice ottimizzazione per far guadagnare velocità e reattività alle nostre pagine.

In questo articolo, ogni esempio assume che main.js sia un file javascript richiamato da una pagina html, mentre worker.js sia un file javascript usato per creare un web worker.

Limitazioni e potenza

Il codice eseguito in un Web Worker ha alcune limitazioni, come il fatto che si trova in un contesto diverso da window, più potente in fatto di calcolo in quanto non implementa certe funzionalità. Non si può, ad esempio, accedere a *document* (bye bye jQuery) o dichiarare una variabile *myVar* per accedervi poi con *window.myVar* (ma si possono usare sia *self.myVar* che *this.myVar*). Inoltre, se si vuole sviluppare nel proprio computer locale usando i Web Worker, sarà bene installare un server HTTP come Apache, presente anche in distribuzioni “AMP” (Apache+MariaDB+PHP) all-in-one come XAMPP.

La compatibilità cross-browser di questa funzionalità di HTML5 è buona: Chrome, Firefox e Safari la implementano già da moltissimo tempo, Internet Explorer la supporta dalla versione 10, e Android Browser dalla versione 4.4. Per verificare se il browser la supporta basta un if:

Anche per sapere se il codice che si sta eseguendo si trova in un web worker basta un if:

Dedicated Worker

Per comunicare, i thread usano l’istruzione *postMessage()* e l’evento *onmessage()* per inviare e ricevere certi tipi di dati, come ad esempio un oggetto JSON. Il codice del worker deve risiedere in un file javascript osservante la Same Origin Policy.

main.js

worker.js

All’interno di un web worker è possibile caricare in maniera asincrona (ma verranno eseguiti in maniera sincrona) file javascript grazie all’istruzione speciale *importScripts()*.

Per terminare un web worker, si usa la funzione *myWorker.terminate()* oppure, dall’interno del web worker, il suo metodo *close()*;

Shared Worker

Per questo tipo di web worker vale tutto quanto è stato detto finora. In più, **è accessibile da qualsiasi finestra, iframe o altro web worker** che osservi la Same Origin Policy. Di conseguenza, grazie ad esso è possibile aggiornare contemporaneamente i dati contenuti in diverse tab quando arriva un input da una sola di esse.

Il codice è un po' più complesso perché fa uso dell'evento *onconnect* e della proprietà *port*. Il codice del worker non viene eseguito ogni volta che viene aperta una nuova finestra: invece, alla creazione di ogni istanza del worker successiva alla prima viene scatenato l'evento *onconnect*. Ecco, quindi, che possiamo dichiarare un array "clients" che conterrà le connessioni a tutte le tab.

main.js

worker.js

NOTA: Nel caso in cui l'evento "message" venga gestito come *port.onmessage()* anziché come *port.addEventListener('message')*, l'istruzione *port.start()* diventa facoltativa. Quando, però, è obbligatoria, per avere una comunicazione bidirezionale dev'essere richiamata sia nel main thread che nel worker thread.

NOTA: Mentre state sviluppando o debuggando con più tab del browser aperte che usano lo stesso shared worker, se modificate il worker.js e poi aggiornate le tab non vedrete i cambiamenti: questi si vedranno soltanto quando ci sarà al massimo una sola tab aperta che usa quello shared worker.

Creare un Worker da una stringa

Un "Inline Worker" non è un tipo di Web Worker: è un web worker creato con una tecnica di programmazione che consente di usare codice che si trova in una stringa anziché in un file.

main.js

NOTA: usando questa tecnica, i file caricati con *importScripts()* andranno indicati con URL assoluta, completa di protocollo.

Trasferire, non copiare

Il messaggio inviato con *postMessage()* viene **copiato** nel web worker. Questo significa che vengono impiegate più risorse del necessario: un bel problema, se dobbiamo elaborare decine di MB, come ad esempio nel caso di un'immagine ad alta risoluzione. Fortunatamente è

possibile **trasferire** oggetti di tipo ArrayBuffer e MessagePort indicandoli come elementi di un array nel secondo argomento di *postMessage()*.

main.js

worker.js

Conclusioni

Con l'introduzione dei Web Worker il linguaggio JavaScript è diventato multithread ed ha ottenuto la capacità di eseguire calcolo parallelo e processi in background. Oltre a fornire maggiore potenza di calcolo, l'utilizzo di questa nuova funzionalità di HTML5 può risolvere problemi di scarsa reattività delle nostre pagine, evitando che gli utenti facciano clic a vuoto e di conseguenza si infastidiscano, abbandonando la nostra applicazione.

GUIDA HTML5: GLI ARTICOLI
1) Guida HTML5:Introduzione
2) Guida HTML5: la prima pagina
3) Guida HTML5: la struttura
4) Guida HTML5: Immagini e outlines
5) Guida HTML5: nuovi elementi semantici
6) Guida HTML5: i form – Parte 1
7) Guida HTML5: i form – Parte 2
8) Guida HTML5: i form – Parte 3
9) Guida HTML5: i form – Parte 4
10) Guida HTML5:i tag audio e video – parte 1
11) Guida HTML5:i tag audio e video – parte 2
12) Guida HTML5: I player video
13) Guida HTML5: Il Canvas – Parte 1
14) Guida HTML5: Il Canvas - Parte 2
15) Guida HTML5: Il Canvas - Parte 3
16) Guida HTML5: Il Canvas - Parte 4
17) HTML5: Web storage
18) Guida HTML5: Web Worker