JavaScript: Timeout, Interval e Date

Timing in JavaScript

Il timing in JavaScript permette di eseguire delle funzioni non immediatamente, ma dopo un certo periodo di tempo. Questa funzionalità è chiamata "scheduling a call" (programmazione di una chiamata). JavaScript offre due metodi principali per gestire l'esecuzione temporizzata:

setTimeout

La funzione `setTimeout` permette di eseguire una funzione una sola volta dopo un determinato intervallo di tempo.

Sintassi:

```javascript
setTimeout(funzione, millisecondi, parametro1, parametro2, ...)
```

Dove:

- `funzione`: la funzione che verrà eseguita (obbligatoria)
- `millisecondi`: il numero di millisecondi di attesa prima dell'esecuzione (opzionale, se omesso viene usato 0)
- `parametro1, parametro2, ...`: parametri aggiuntivi da passare alla funzione (opzionali)

`setTimeout` può essere utilizzato in diversi modi:

1. Con una funzione nominata:

```javascript
function saluta() {
  console.log('Ciao');
}
setTimeout(saluta, 1000); // Esegue dopo 1 secondo
```

2. Con una funzione anonima:

```javascript
setTimeout(function() {
  console.log('Ciao');
}, 3000); // Esegue dopo 3 secondi
```

3. Con una funzione freccia:

```javascript
setTimeout(() => {
  console.log('Ciao');
}, 3000);
```

4. Con parametri:

```javascript
function saluta(frase, chi) {
  console.log(frase + ', ' + chi);
}
setTimeout(saluta, 1000, 'Ciao', 'Giovanni');
```

clearTimeout

Una chiamata a `setTimeout` restituisce un "identificatore del timer" (`timerId`) che possiamo usare per annullare l'esecuzione della funzione programmata:

```javascript
let timerId = setTimeout(function() {
  console.log('questo non avverrà mai');
}, 1000);
clearTimeout(timerId); // Annulla l'esecuzione programmata
```

setInterval

La funzione `setInterval` permette di eseguire una funzione regolarmente con un intervallo specificato, ripetendo l'esecuzione finché non viene fermata.

Sintassi:

```javascript
setInterval(funzione, millisecondi, parametro1, parametro2, ...)
```

I parametri sono identici a quelli di `setTimeout`. La differenza è che `setInterval` continuerà a eseguire la funzione a intervalli regolari anziché una sola volta.

Esempi di utilizzo:

```javascript
// Con funzione nominata
function saluta() {
  console.log('Ciao');
}
setInterval(saluta, 1000);

// Con funzione anonima
setInterval(function() {
  console.log('Ciao');
}, 3000);

// Con funzione freccia
setInterval(() => {
  console.log('Ciao');
}, 3000);

// Con parametri
function saluta(frase, chi) {
  console.log(frase + ', ' + chi);
}
setInterval(saluta, 1000, 'Ciao', 'Giovanni');
```

clearInterval

Analogamente a `clearTimeout`, è possibile interrompere l'esecuzione regolare impostata con `setInterval` usando `clearInterval`:

```javascript
// Ripete con un intervallo di 2 secondi
let timerId = setInterval(function() {
  console.log('tick');
}, 2000);

// Dopo 5 secondi, interrompe l'esecuzione
setTimeout(function() {
  clearInterval(timerId);
  console.log('stop');
}, 5000);
```

JavaScript è single-threaded: Event Loop e funzionamento interno dei timer

JavaScript è un linguaggio di programmazione single-threaded, il che significa che può eseguire una sola operazione alla volta all'interno di un singolo thread di esecuzione. Questo concetto è

fondamentale per comprendere come funzionano realmente i timer in JavaScript.

Come funziona l'Event Loop

Per capire come `setTimeout` e `setInterval` operano, dobbiamo prima comprendere l'Event Loop di JavaScript, che costituisce il cuore del modello di concorrenza del linguaggio:

1. **Call Stack**: È una struttura dati che registra il punto di esecuzione del programma. Quando si chiama una funzione, viene inserita (push) nello stack. Quando una funzione termina, viene rimossa (pop) dallo stack.
2. **WebAPIs**: Sono funzionalità fornite dal browser (non dal motore JavaScript stesso) che possono operare in modo asincrono. Qui vengono elaborati timer, richieste HTTP, eventi DOM, ecc.
3. **Callback Queue (o Task Queue)**: È una coda dove vengono inserite le funzioni callback quando le API asincrone completano le loro operazioni.
4. **Event Loop**: È un processo continuo che controlla:
   - Se la Call Stack è vuota
   - Se ci sono callback nella Callback Queue
   - In tal caso, prende la prima callback dalla coda e la inserisce nella Call Stack per l'esecuzione

Funzionamento di setTimeout e setInterval

Quando chiamiamo `setTimeout`:

1. JavaScript registra il timer presso le WebAPIs
2. La funzione `setTimeout` completa la sua esecuzione e viene rimossa dalla Call Stack
3. L'esecuzione del programma continua con il codice successivo
4. Nel frattempo, la WebAPI conta il tempo richiesto
5. Quando il timer scade, la callback viene inserita nella Callback Queue
6. L'Event Loop, quando la Call Stack è vuota, trasferisce la callback nella Call Stack
7. La callback viene eseguita

Questo spiega perché, nonostante l'intervallo specificato, il tempo di esecuzione effettivo può essere maggiore del previsto: se la Call Stack è occupata con altre operazioni lunghe, l'Event Loop non può trasferire la callback finché lo stack non si libera.

```javascript
console.log('Inizio');
setTimeout(() => console.log('Timer di 0ms'), 0); // Verrà comunque eseguito dopo
console.log('Fine');

// Output:
// Inizio
// Fine
// Timer di 0ms
```

Nel caso di un ciclo infinito o di un'operazione che blocca il thread principale, le funzioni programmate con `setTimeout` o `setInterval` non verranno mai eseguite perché l'Event Loop non avrà mai l'opportunità di spostarle dalla Callback Queue alla Call Stack:

```javascript
function logMessaggiRitardati() {
  setTimeout(() => console.log('1 secondo'), 1000);
  setTimeout(() => console.log('2 secondi'), 2000);
  while (true) {} // ciclo infinito che blocca il thread principale
}
logMessaggiRitardati(); // I messaggi non verranno mai visualizzati
```

Precisione dei timer

È importante notare che JavaScript non garantisce l'esecuzione esatta del timer al millisecondo specificato, per diverse ragioni:

1. **Risoluzione del timer**: I browser limitano la precisione dei timer (spesso a circa 4ms) per ragioni di sicurezza e performance.
2. **Priorità del browser**: Le operazioni ad alta priorità possono ritardare l'esecuzione dei timer.
3. **Throttling nelle schede in background**: Per risparmiare risorse, i browser limitano l'esecuzione di timer nelle schede non attive (spesso a 1000ms).
4. **Occupazione della Call Stack**: Se ci sono operazioni prolungate nella Call Stack, l'Event Loop non può processare le callback nel momento esatto.

Implementazione di setInterval tramite setTimeout

Una peculiarità interessante è che è possibile implementare un comportamento simile a `setInterval` utilizzando `setTimeout` in modo ricorsivo:

javascript

```javascript
function customSetInterval(callback, delay) {
  function repeat() {
    callback();
    timeoutId = setTimeout(repeat, delay);
  }

  let timeoutId = setTimeout(repeat, delay);

  // Funzione per fermare l'intervallo
  return {
    clear: function() { clearTimeout(timeoutId); }
  };
}

// Utilizzo
const myInterval = customSetInterval(() => console.log('Tick'), 1000);
// Per fermare: myInterval.clear();
```

Questo approccio offre un vantaggio rispetto al `setInterval` nativo: garantisce che non ci siano sovrapposizioni di esecuzioni della callback, poiché ogni nuova esecuzione viene programmata solo dopo il completamento della precedente.

Data e ora in JavaScript

JavaScript fornisce l'oggetto `Date` per lavorare con date e orari.

L'oggetto Date

Una data in JavaScript consiste di anno, mese, giorno, ora, minuto, secondo e millisecondi.

Sintassi per creare un oggetto Date:

javascript

```javascript
new Date() // Data e ora correnti
new Date(millisecondi) // Millisecondi dalla "epoch" (1 gennaio 1970)
new Date(stringaData) // Data specificata come stringa
new Date(anno, mese, giorno, ore, minuti, secondi, millisecondi) // Data e ora specifici
```

Esempi:

```javascript
let d = new Date(); // Data e ora correnti
let d = new Date("13 Ottobre 2024 11:13:00");
let d = new Date(86400000); // Rappresenta il 2 gennaio 1970 (un giorno dopo l'epoch)
let d = new Date(2024, 3, 18, 11, 33, 30, 0); // 18 aprile 2024, 11:33:30
```

Note importanti sui parametri:

- Il valore 0 rappresenta il 1° gennaio 1970, 00:00:00 UTC (l'epoch di UNIX)
- I valori dell'anno tra 0 e 99 vengono mappati agli anni dal 1900 al 1999:

```javascript
let d = new Date(99, 5, 24); // 24 giugno 1999
let d = new Date(24, 5, 24); // 24 giugno 1924
let d = new Date(2024, 5, 24); // 24 giugno 2024
```

Visualizzazione delle date

JavaScript converte automaticamente gli oggetti Date in stringhe quando necessario:

```javascript
let d = new Date();
console.log(d); // Mostra la data
console.log(d.toString()); // Lo stesso output, ma esplicito
console.log(d.toUTCString()); // Data in formato UTC
console.log(d.toDateString()); // Solo la parte della data
```

Formati di data

JavaScript supporta diversi formati di input per le date:

- **Formato ISO**: "2024-03-25" (standard internazionale)
- **Data breve**: "03/25/2024"
- **Data lunga**: "Mar 25 2024" o "25 Mar 2024"
- **Data completa**: "Wednesday March 25 2024"

Metodi get dell'oggetto Date

L'oggetto Date fornisce numerosi metodi per ottenere informazioni sulla data:

```javascript
let d = new Date();
console.log(d.getMonth()); // Restituisce il mese (0-11, dove 0 è gennaio)
console.log(d.getMinutes()); // Restituisce i minuti (0-59)
console.log(d.getFullYear()); // Restituisce l'anno (4 cifre)

// Per ottenere il timestamp (millisecondi dall'epoch):
new Date().valueOf();
new Date().getTime();
Date.now(); // Metodo statico, non richiede creazione di oggetto
```

Altri metodi get disponibili includono:

- `getDate()`: giorno del mese (1-31)
- `getDay()`: giorno della settimana (0-6, dove 0 è domenica)
- `getHours()`: ore (0-23)
- `getSeconds()`: secondi (0-59)
- `getMilliseconds()`: millisecondi (0-999)

Metodi set dell'oggetto Date

Esistono anche metodi per modificare una data:

```javascript
let d = new Date();
d.setFullYear(2024, 0, 14); // Imposta la data al 14 gennaio 2024
d.setDate(d.getDate() + 50); // Aggiunge 50 giorni alla data

// Analisi di una stringa di data
let msec = Date.parse('21 Marzo 2024');
let date = new Date(msec);
```

Altri metodi set disponibili includono:

- `setMonth(mese, [giorno])`: imposta il mese (0-11)
- `setDate(giorno)`: imposta il giorno del mese (1-31)
- `setHours(ore, [minuti], [secondi], [millisecondi])`: imposta l'ora
- `setMinutes(minuti, [secondi], [millisecondi])`: imposta i minuti
- `setSeconds(secondi, [millisecondi])`: imposta i secondi
- `setMilliseconds(millisecondi)`: imposta i millisecondi

- `setTime(millisecondi)`: imposta la data in millisecondi dall'epoch

Confronto tra date

Gli operatori di confronto funzionano anche con le date:

```javascript
let data1 = new Date();
let data2 = new Date("24 Febbraio 2024 15:50:00");
if (data1 > data2) {
   console.log('pausa');
} else {
   console.log('resta in classe');
}
```

Considerazioni finali

Il timing e la gestione delle date sono componenti essenziali della programmazione JavaScript. I metodi di timing come `setTimeout` e `setInterval` permettono di eseguire codice in modo asincrono, rendendo possibile creare animazioni, aggiornamenti periodici e ritardi controllati.

L'oggetto `Date` offre un insieme completo di strumenti per creare, manipolare e confrontare date, essenziale per qualsiasi applicazione che richieda funzionalità legate al tempo, come calendari, orologi, pianificatori e sistemi di prenotazione.

È importante comprendere che JavaScript è single-threaded, quindi le operazioni che bloccano il thread principale impediranno l'esecuzione delle funzioni temporizzate. In scenari complessi, potrebbero essere necessarie tecniche più avanzate come Web Workers per gestire operazioni intensive senza bloccare l'interfaccia utente.

Risorse utili per approfondire

- MDN Web Docs su setTimeout e setInterval
- JavaScript Date Objects su W3Schools
- MDN Web Docs sugli oggetti Date
- Documentazione JavaScript per i metodi delle date