14th February 2020

Event loop: microtasks and macrotasks

Browser JavaScript execution flow, as well as in Node.js, is based on an *event loop*.

Understanding how event loop works is important for optimizations, and sometimes for the right architecture.

In this chapter we first cover theoretical details about how things work, and then see practical applications of that knowledge.

Event Loop

The concept of *event loop* is very simple. There's an endless loop, when JavaScript engine waits for tasks, executes them and then sleeps waiting for more tasks.

The general algorithm of the engine:

1. While there are tasks:
   - execute them, starting with the oldest task.
2. Sleep until a task appears, then go to 1.

That's a formalization for what we see when browsing a page. JavaScript engine does nothing most of the time, only runs if a script/handler/event activates.

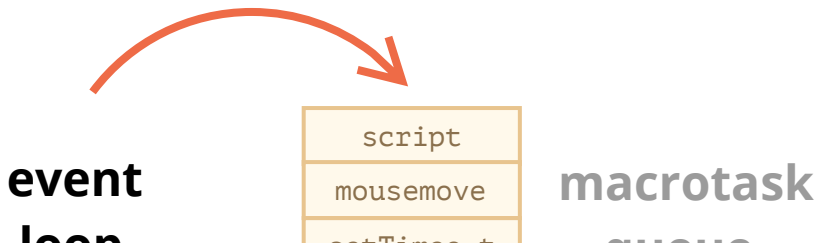
Examples of tasks:

- When an external script `<script src="...">` loads, the task is to execute it.
- When a user moves their mouse, the task is to dispatch `mousemove` event and execute handlers.
- When the time is due for a scheduled `setTimeout`, the task is to run its callback.
- …and so on.

Tasks are set – the engine handles them – then waits for more tasks (while sleeping and consuming close to zero CPU).

It may happen that a task comes while the engine is busy, then it's enqueued.

The tasks form a queue, so-called "macrotask queue" (v8 term):



For instance, while the engine is busy executing a `script`, a user may move their mouse causing `mousemove`, and `setTimeout` may be due and so on, these tasks form a queue, as illustrated on the picture above.

Tasks from the queue are processed on "first come – first served" basis. When the engine browser is done with the `script`, it handles `mousemove` event, then `setTimeout` handler, and so on.

So far, quite simple, right?

Two more details:

1. Rendering never happens while the engine executes a task. Doesn't matter if the task takes a long time. Changes to DOM are painted only after the task is complete.
2. If a task takes too long, the browser can't do other tasks, process user events, so after a time it raises an alert like "Page Unresponsive" suggesting to kill the task with the whole page. That happens when there are a lot of complex calculations or a programming error leading to infinite loop.

That was a theory. Now let's see how we can apply that knowledge.

Use-case 1: splitting CPU-hungry tasks

Let's say we have a CPU-hungry task.

For example, syntax-highlighting (used to colorize code examples on this page) is quite CPU-heavy. To highlight the code, it performs the analysis, creates many colored elements, adds them to the document – for a large amount of text that takes a lot of time.

While the engine is busy with syntax highlighting, it can't do other DOM-related stuff, process user events, etc. It may even cause the browser to "hiccup" or even "hang" for a bit, which is unacceptable.

We can avoid problems by splitting the big task into pieces. Highlight first 100 lines, then schedule `setTimeout` (with zero-delay) for the next 100 lines, and so on.

To demonstrate this approach, for the sake of simplicity, instead of text-highlighting, let's take a function that counts from `1` to `1000000000`.

If you run the code below, the engine will "hang" for some time. For server-side JS that's clearly noticeable, and if you are running it in-browser, then try to click other buttons on the page – you'll see that no other events get handled until the counting finishes.

```
let i = 0;

let start = Date.now();

function count() {

  // do a heavy job
  for (let j = 0; j < 1e9; j++) {
```

```
    i++;
  }

  alert("Done in " + (Date.now() - start) + 'ms');
}

count();
```

The browser may even show a “the script takes too long” warning.

Let’s split the job using nested `setTimeout` calls:

```
let i = 0;

let start = Date.now();

function count() {

  // do a piece of the heavy job (*)
  do {
    i++;
  } while (i % 1e6 != 0);

  if (i == 1e9) {
    alert("Done in " + (Date.now() - start) + 'ms');
  } else {
    setTimeout(count); // schedule the new call (**)
  }

}

count();
```

Now the browser interface is fully functional during the “counting” process.

A single run of `count` does a part of the job `(*)`, and then re-schedules itself `(**)` if needed:

1. First run counts: `i=1...1000000`.
2. Second run counts: `i=1000001..2000000`.
3. …and so on.

Now, if a new side task (e.g. `onclick` event) appears while the engine is busy executing part 1, it gets queued and then executes when part 1 finished, before the next part. Periodic returns to the event loop between `count` executions provide just enough “air” for the JavaScript engine to do something else, to react to other user actions.

The notable thing is that both variants – with and without splitting the job by `setTimeout` – are comparable in speed. There’s not much difference in the overall counting time.

To make them closer, let’s make an improvement.

We’ll move the scheduling to the beginning of the `count()`:

```
let i = 0;

```

```
let start = Date.now();

function count() {

  // move the scheduling to the beginning
  if (i < 1e9 - 1e6) {
    setTimeout(count); // schedule the new call
  }

  do {
    i++;
  } while (i % 1e6 != 0);

  if (i == 1e9) {
    alert("Done in " + (Date.now() - start) + 'ms');
  }

}

count();
```

Now when we start to `count()` and see that we'll need to `count()` more, we schedule that immediately, before doing the job.

If you run it, it's easy to notice that it takes significantly less time.

Why?

That's simple: as you remember, there's the in-browser minimal delay of 4ms for many nested `setTimeout` calls. Even if we set `0`, it's `4ms` (or a bit more). So the earlier we schedule it – the faster it runs.

Finally, we've split a CPU-hungry task into parts – now it doesn't block the user interface. And its overall execution time isn't much longer.

Use case 2: progress indication

Another benefit of splitting heavy tasks for browser scripts is that we can show progress indication.

Usually the browser renders after the currently running code is complete. Doesn't matter if the task takes a long time. Changes to DOM are painted only after the task is finished.

On one hand, that's great, because our function may create many elements, add them one-by-one to the document and change their styles – the visitor won't see any "intermediate", unfinished state. An important thing, right?

Here's the demo, the changes to `i` won't show up until the function finishes, so we'll see only the last value:

```
<div id="progress"></div>

<script>

  function count() {
    for (let i = 0; i < 1e6; i++) {
      i++;
      progress.innerHTML = i;
    }
  }
```

```
11
12    count();
13  </script>
```

…But we also may want to show something during the task, e.g. a progress bar.

If we split the heavy task into pieces using `setTimeout`, then changes are painted out in-between them.

This looks prettier:

```
1  <div id="progress"></div>
2
3  <script>
4    let i = 0;
5
6    function count() {
7
8      // do a piece of the heavy job (*)
9      do {
10       i++;
11       progress.innerHTML = i;
12     } while (i % 1e3 != 0);
13
14     if (i < 1e7) {
15       setTimeout(count);
16     }
17
18   }
19
20   count();
21 </script>
```

Now the `<div>` shows increasing values of `i`, a kind of a progress bar.

Use case 3: doing something after the event

In an event handler we may decide to postpone some actions until the event bubbled up and was handled on all levels. We can do that by wrapping the code in zero delay `setTimeout`.

In the chapter Dispatching custom events we saw an example: custom event `menu-open` is dispatched in `setTimeout`, so that it happens after the “click” event is fully handled.

```
1  menu.onclick = function() {
2    // ...
3
4    // create a custom event with the clicked menu item data
5    let customEvent = new CustomEvent("menu-open", {
6      bubbles: true
7    });
8
9    // dispatch the custom event asynchronously
10   setTimeout(() => menu.dispatchEvent(customEvent));
11 };
```

Macrotasks and Microtasks

Along with *macrotasks*, described in this chapter, there exist *microtasks*, mentioned in the chapter Microtasks.

Microtasks come solely from our code. They are usually created by promises: an execution of `.then/catch/finally` handler becomes a microtask. Microtasks are used "under the cover" of `await` as well, as it's another form of promise handling.

There's also a special function `queueMicrotask(func)` that queues `func` for execution in the microtask queue.

Immediately after every *macrotask*, the engine executes all tasks from *microtask* queue, prior to running any other macrotasks or rendering or anything else.

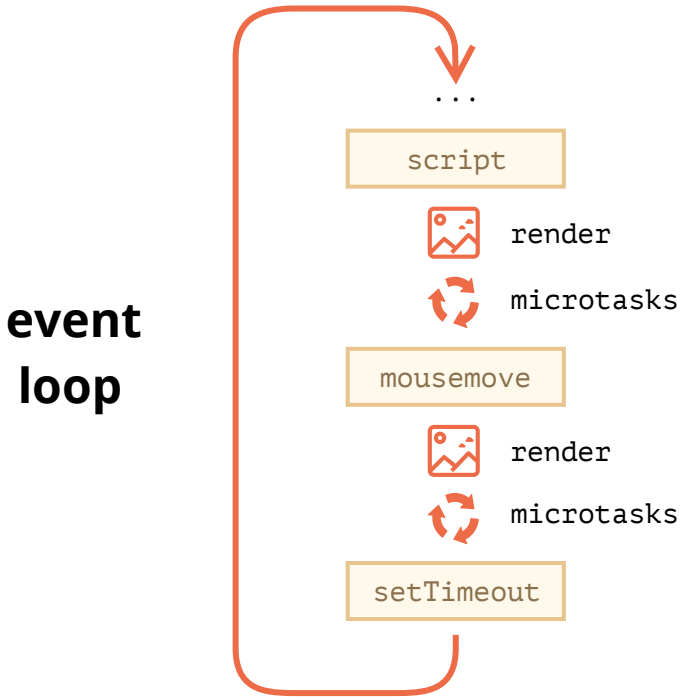
For instance, take a look:

```
1  setTimeout(() => alert("timeout"));
2
3  Promise.resolve()
4    .then(() => alert("promise"));
5
6  alert("code");
```

What's going to be the order here?

1. `code` shows first, because it's a regular synchronous call.
2. `promise` shows second, because `.then` passes through the microtask queue, and runs after the current code.
3. `timeout` shows last, because it's a macrotask.

The richer event loop picture looks like this (order is from bottom to top, that is: `setTimeout` first, then microtasks and so on):



All microtasks are completed before any other event handling or rendering or any other macrotask takes place.

That's important, as it guarantees that the application environment is basically the same (no mouse coordinate changes, no new network data, etc) between microtasks.

If we'd like to execute a function asynchronously (after the current code), but before changes are rendered or new events handled, we can schedule it with `queueMicrotask`.

Here's an example with "counting progress bar", similar to the one shown previously, but `queueMicrotask` is used instead of `setTimeout`. You can see that it renders at the very end. Just like the synchronous code:

```
<div id="progress"></div>

<script>
  let i = 0;

  function count() {

    // do a piece of the heavy job (*)
    do {
      i++;
      progress.innerHTML = i;
    } while (i % 1e3 != 0);

    if (i < 1e6) {
      queueMicrotask(count);
    }

  }

  count();
</script>
```

Summary

The more detailed algorithm of the event loop (though still simplified compare to the specification):

1. Dequeue and run the oldest task from the *macrotask* queue (e.g. "script").
2. Execute all *microtasks*:
   - While the microtask queue is not empty:
     - Dequeue and run the oldest microtask.
3. Render changes if any.
4. If the macrotask queue is empty, wait till a macrotask appears.
5. Go to step 1.

To schedule a new *macrotask*:

- Use zero delayed `setTimeout(f)`.

That may be used to split a big calculation-heavy task into pieces, for the browser to be able to react on user events and show progress between them.

Also, used in event handlers to schedule an action after the event is fully handled (bubbling done).

To schedule a new *microtask*

- Use `queueMicrotask(f)`.
- Also promise handlers go through the microtask queue.

There's no UI or network event handling between microtasks: they run immediately one after another.

So one may want to `queueMicrotask` to execute a function asynchronously, but within the environment state.

Web Workers

For long heavy calculations that shouldn't block the event loop, we can use Web Workers.

That's a way to run code in another, parallel thread.

Web Workers can exchange messages with the main process, but they have their own variables, and their own event loop.

Web Workers do not have access to DOM, so they are useful, mainly, for calculations, to use multiple CPU cores simultaneously.

Previous lesson Next lesson

Share

Tutorial map

Comments

- If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.
- If you can't understand something in the article – please elaborate.
- To insert a few words of code, use the `<code>` tag, for several lines – use `<pre>`, for more than 10 lines – use a sandbox (plnkr, JSBin, codepen…)