JavaScript: How is callback execution strategy for promises different than DOM events callback?

jitendra kasaudhan
May 5, 2018 · 10 min read ★

As a JavaScript developer, first of all, I thought, why the heck do I need to understand the internals of browsers with respect to JavaScript? Sometimes I have been through a situation like “WHAT?? THIS IS STRANGE!!” but later on after diving deeper into the subject matter, I realized that it’s important to understand a bit of how browser and JS engine work together. I hope this article will provide a bit of guidance to predict the correct behavior of your JS code and minimize strange situations.

Basically, this article covers following sub topics

1. Quiz to predict the output of the sample JS code.
2. An overview of browser, event loop, JS engine, event loop queues.
3. How browser executes DOM events(click, http requests) and its callback?
4. How is callback execution strategy for promises different than DOM events callback?
5. Different execution strategy of tasks queued in Task queue and Micro task queue.

```
{
 console.log("Start");
 // First settimeout
 setTimeout(function CB1() {
    console.log("Settimeout 1");
 }, 0);
 // Second settimeout
 setTimeout(function CB2() {
    console.log("Settimeout 2");
 }, 0);
 // First promise
 Promise.resolve().then(function CB3() {
    for(let i=0; i<100000; i++) {}
    console.log("Promise 1");
 });
 // Second promise
 Promise.resolve().then(function CB4() {
    console.log("Promise 2");
 });

 console.log("End");
}
```

taskQueueMicroTaskQueueTest1 hosted with ❤️ by GitHub view raw

Test 2: We have a button with two click event listeners on the same button as shown in the code sample below.

```
var button = document.querySelector(".button");

// First click listener
button.addEventListener("click", function CB1() {
  console.log("Listener 1");

  setTimeout(function ST1() {
    console.log("Settimeout 1");
```

```
    setTimeout(function ST2() {
      console.log("Settimeout 2");
    }, 0);

    Promise.resolve().then(function P2() {
          console.log("Promise 2");
    });
  });
```

taskQueueMicroTaskQueueTest2 hosted with ❤️ by GitHub view raw

Running example can be found here . Please try to predict the order of log messages when a button is clicked?

If we have correct versions of browsers, output of the above code samples are as follow

Test 1: Start, End, Promise 1, Promise 2, Settimeout 1, Settimeout 2

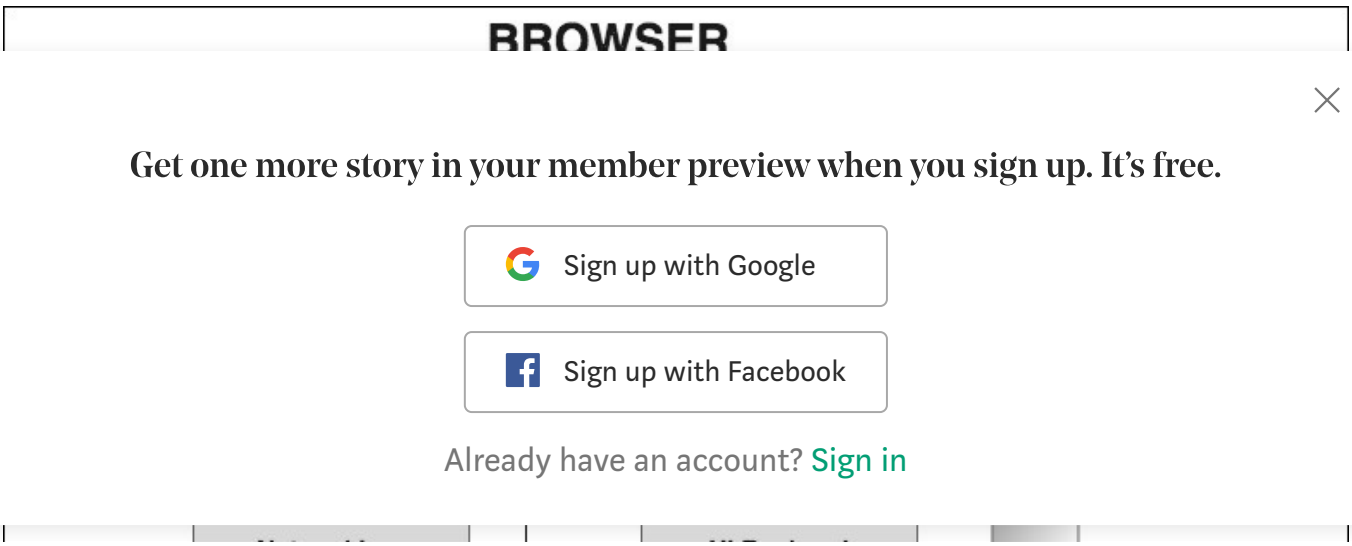
Test 2: Listener 1, Promise 1, Listener 2, Promise 2,Settimeout 1,Settimeout 2

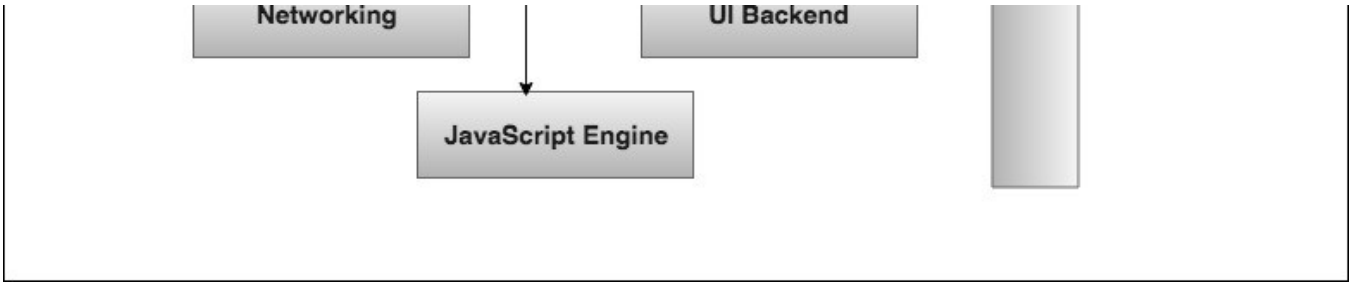
I have tested on following versions of browsers(Chrome 65, Firefox 60, Safari 8.0.2)

If you have predicted it right, AWESOME!!! but if it surprises you, then you are welcome to read further :)

It is important to understand this behavior because popular browsers have implemented this execution strategy for performance reason(eg. to avoid race conditions). In order to understand the execution strategy of callback functions, let's try to understand basic overview of how internal of browsers work together.

2. An overview of browser





2.1. User interface: It includes every part of the browser which is visible to the user except the window. For eg. the address bar, back/forward button, bookmarking menu, etc.

2.2. The browser engine: It acts as a bridge between UI and the rendering engine and provide several methods to interact with a web page such as reloading a page, back, forward etc.

2.3. The rendering engine: It is responsible for displaying requested content. For example, if the requested content is HTML, the rendering engine parses HTML and CSS and displays the parsed content on the screen.

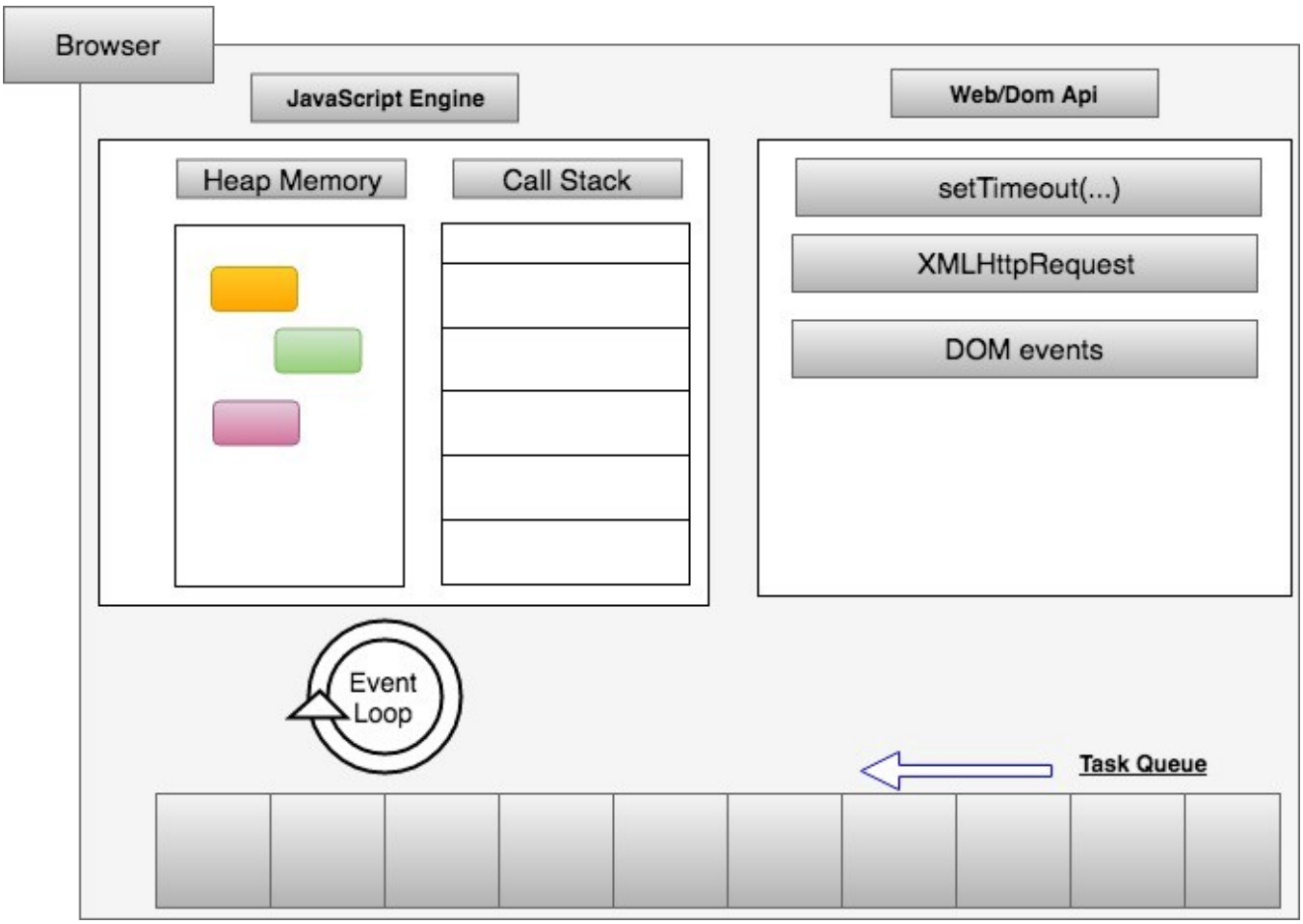
2.4. Networking: It is responsible for network calls such as HTTP requests and get actual content to render.

2.4. UI backend: It is used for drawing basic widgets like combo boxes and windows. This backend exposes a generic interface that is not platform specific. Underneath it uses an operating system user interface method.

2.5. JavaScript Engine: Executes actual JavaScript code.

2.6. Data storage. This is a persistence layer. The browser may need to save all sorts of data locally, such as cookies. Browsers also support storage mechanisms such as localStorage, IndexedDB, WebSQL, and FileSystem.

A runtime environment is the execution environment provided to an application by the operating system. In a runtime environment, the application can send instructions or commands to the processor and access other system resources such as RAM, DISK etc. JS engine, Event queues, Event loop and Web/Dom apis forms the Runtime Environment.

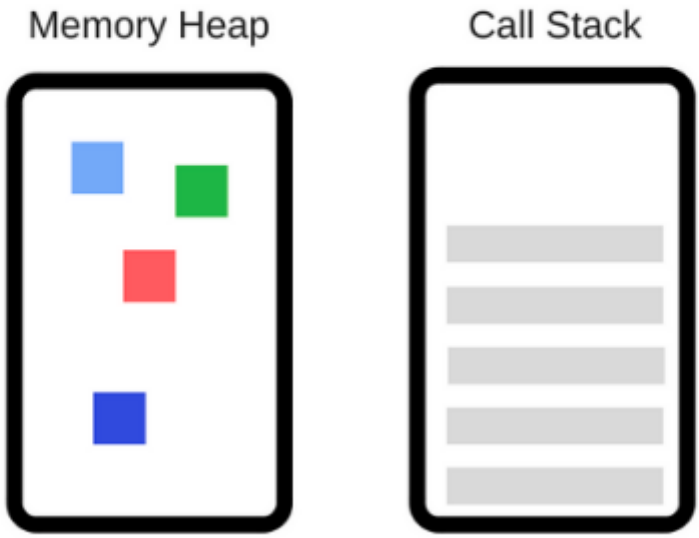


Browser's Runtime Environment

References: What the heck is event loop? JavaScript runtime simulator

JavaScript Engine

It consists of two main components, **Heap Memory** and **Call Stack.** It does not handle



Similarly, call stack load our main JS code and starts executing it. Whenever a function is encountered in our JS code, JS engine creates a new stack and piles it on top and starts executing that function.

Task Queue
Task queue is a data structure that holds callback functions to be executed. Task which is queued first is processed first (first-in-first-out behavior).

Event Loop
The event loop is the mastermind that orchestrates:

- What JavaScript code gets executed?
- When does it run?

new task from the task queue until the queue is emptied. Following piece of a pseudocode illustrates basics of how event loop looks like

```
// `eventLoop` is an array that acts as a queue (first-in, first-out)
var eventLoop = [ ];
var event;

// keep going "forever"
while (true) {
        // perform a "tick"
        if (eventLoop.length > 0) {
                // get the next event in the queue
                event = eventLoop.shift();

                // now, execute the next event
                try {
                        event();
                }
                catch (err) {
                        reportError(err);
                }
        }
}
```

basicEventLoopPseudoCode hosted with ❤ by GitHub view raw

Basic event loop pseudo code

As you can see, there is a continuously running loop represented by the `while` loop, and each iteration of this loop is called a "tick." For each tick, if an event is waiting on the queue, it's taken off and executed. These events are your function callbacks. (Source: You Don't Know JS — Async and Performance series)

Following code shows what standard event loop specification says

```
    ],

    nextTask: function() {
        // Spec says:
        // "Select the oldest task on one of the event loop's task   queues"
        // Which gives browser implementers lots of freedom
        // Queues can have different priorities, etc.
        for (let q of taskQueues)
            if (q.length > 0)
                return q.shift();
        return null;
    },

    executeMicrotasks: function() {
        if (scriptExecuting)
            return;
        let microtasks = this.microtaskQueue;
        this.microtaskQueue = [];
        for (let t of microtasks)
            t.execute();
    },

    needsRendering: function() {
        return vSyncTime() && (needsDomRerender() || hasEventLoopEventsToDispatch());
    },

    render: function() {
        dispatchPendingUIEvents();
        resizeSteps();
        scrollSteps();
        mediaQuerySteps();
        cssAnimationSteps();
        fullscreenRenderingSteps();

        animationFrameCallbackSteps();


```

```
59      if (task) {
60          task.execute();
61      }
62      eventLoop.executeMicrotasks();
63      if (eventLoop.needsRendering())
64          eventLoop.render();
65  }
```

detailEventLoopPseudoCode hosted with ❤ by GitHub view raw

References: Event loop explainer , Standard event loop specification

3. How browser executes DOM events(click, http requests) and its callback?

There are different types of events supported by browser such as
- Keyboard events (keydown, keyup etc)
- Mouse events (click, mouseup, mousedown etc)
- Network events (online, offline)
- Drag and drop events (dragstart, dragend)etc

These events can have a callback handler which should be executed whenever event is fired. Whenever event is fired, it's callback (aka task) is queued in the task queue. As shown in **Test 2**, when a button is clicked, it's callback handler *CB1()* is queued in a task queue and event loop is responsible to pick it up and execute it in a call stack. There are several rules event loop applies, before picking up a task from a task queue and executing it in a call stack.

Event loop checks, if call stack is empty or not. **If call stack is empty** and there is **nothing to execute in a micro task queue**, than it picks up a task from a Task queue and execute it.

4. How is callback execution strategy for promises different than DOM events callback?

queue but not in Task queue.

As mentioned before, event loop will pick up a new task from a Task queue only when call stack is empty and there is nothing to execute in a micro task queue. Let's assume, there are 3 tasks in Task queue, T1, T2 and T3. Task T1 has one **task**(say — setTimeout(T4, 0)) and two **micro tasks**(say promises — M1, M2). When task T1 is executed in the call stack, it will encounter setTimeout(…) and delegates it to runtime to handle its callback. Runtime will queue T4 in aTask queue. When engine encounters promise 1, it will queue its callback (M1) to micro task queue. Likewise, when it encounters another promise 2 object, it will queue it in a micro task queue. Now call stack becomes clear, so before picking up task T2 from the Task queue, event loop will execute all the callbacks (M1, M2) queued in micro task queue. Once micro tasks are executed in a call stack and stack is cleared, it is ready for Task T2.

NOTE (**Exception**): Even though window.requestAnimationFrame(…) is a function of DOM object *window,* it's callback is queued in a micro task queue but it's execution strategy is different. Execution strategy of window.requestAnimationFrame(…) has not been covered in this article.

5. What kind of tasks are queued in Task queue and Micro task queue?
Tasks are basically callback functions of promises or DOM/web api events. Because tasks in Micro task queue and Task queue are processed in a different way, browser should decide types of tasks which should be queued in Task queue or Micro task queue. According to the standard specification, callback handlers of following events are **queued in Task queue**
- DOM/Web events (onclick, onkeydown, XMLHttpRequest etc)
- Timer events (setTimeout(…), setInterval(…))

Similarly, callback handlers following objects are **queued in Micro task** queue
- Promises (resolve(), reject())

6. Different execution strategy of tasks queued in Task queue and Micro task queue

Callbacks queued in task queue are executed in first-come-first-service order and browser may render between them (Eg. DOM manipulation, changing html styles etc).

Callbacks queued in Micro task queue are executed in first-come-first-service order, but at the end of every task from the task queue (only if call stack is empty). As mentioned above in the event loop's pseudo code, Micro tasks are processed at the end of each task.

```
// Popular JS engines(Eg. google chrome's V8 engine, Mozilla's SpiderMonkey etc)
// implements event loop using C++, which means code snippet below is executed synchron
while(true) {
    // Each iteration of this loop is called an event loop 'tick'
    task = eventLoop.nextTask();

    if (task) {
     // First: A task from the Task queue is executed
        task.execute();
    }

    // Second: All the tasks in the Micro task queue are executed
    eventLoop.executeMicrotasks();

    // Third: It will check if there is someting to render, eg. DOM changes, request ar
    // and renders in browser if required
    if (eventLoop.needsRendering())
        eventLoop.render();
}
```

actualEventLoopPseudoCode hosted with ❤ by GitHub view raw

References: Tasks, microtasks, queues and schedules , In The Loop — JSConf.Asia 2018 — By JakeArchibald

console.log("End") statement. According to standard specification, once a call stack is emptied, it will check Micro task queue and finds CB3 and CB4. Call stack will execute CB3 (logs Promise 1) and than CB4 (logs Promise 2). Once again, call stack is emptied after processing callbacks in Micro task queue. Finally, event loop picks up a new task from the Task queue i.e CB1 (logs setTimeout 1) execute it. Likewise, event loop will pick up other task from Task queue and execute it in the call stack until Task queue is emptied.

Placement of callbacks in Task and Micro task queue

For animated simulation of the above code sample, I would recommend you to read the blog post by Jack Archibald.

Test 2

When call stack is empty, event loop picks up CB1 to execute. First console.log(“Listener 1 ”) is executed, then callback of setTimeout(…) is queued in Task queue as ST1 and callback of promise 1 is queued in Micro task queue as shown in the diagram below

for setTimeout(…) ST2 is queued in Task queue and promise (P2) is queued in micro task queue as shown in the diagram below

Finally, P2 , ST1 and ST2 are executed sequentially which logs Promise 2, Settimeout 1 and Settimeout 2.

NOTE ON EVENT LOOP: Until now (April, 2018), event loop is a part of browser's runtime environment but not JavaScript engine. At some point in future, it might be a part of JS engine as mentioned in a book You Don't Know JavaScript — Async and performance. One main reason for this change is the introduction of ES6 Promises, because they require the ability to have direct, fine-grained control over scheduling operations on the event loop queue.

Originally published on zeolearn.com