[Tự học Javascript] Giải thích chi tiết về Event Loop

Bởi

[**Sơn Dương**](https://vntalking.com/author/smobile)

 -

Tháng Mười Một 8, 2021

[0](https://vntalking.com/tu-hoc-javascript-giai-thich-event-loop.html#respond)

Khi học lập trình Javascript, việc hiểu được khái niệm Event Loop và cơ chế hoạt động của nó là điều bắt buộc.

Bởi vì hiểu được khái niệm này sẽ giúp bạn hiểu được bản chất của những dòng code mà bạn đang viết nó chạy như thế nào.

Bài viết này mình sẽ cố gắng giải thích thật chi tiết và dễ hiểu về chủ đề này nhé

Bắt đầu thôi nhỉ!

**Nội dung chính của bài viết**

* [Một số thành phần trong kiến trúc của Javascript](https://vntalking.com/tu-hoc-javascript-giai-thich-event-loop.html#mot-so-thanh-phan-trong-kien-truc-cua-javascript)
* [Message queue](https://vntalking.com/tu-hoc-javascript-giai-thich-event-loop.html#message-queue)
* [Giải thích dễ hiểu về cách hoạt động của Message Queue](https://vntalking.com/tu-hoc-javascript-giai-thich-event-loop.html#giai-thich-de-hieu-ve-cach-hoat-dong-cua-message-queue)
* [Call stack, message queue and priorities](https://vntalking.com/tu-hoc-javascript-giai-thich-event-loop.html#call-stack-message-queue-and-priorities)
* [Cách xử lý của message queue](https://vntalking.com/tu-hoc-javascript-giai-thich-event-loop.html#cach-xu-ly-cua-message-queue)
* [Javascript Event loop](https://vntalking.com/tu-hoc-javascript-giai-thich-event-loop.html#javascript-event-loop)

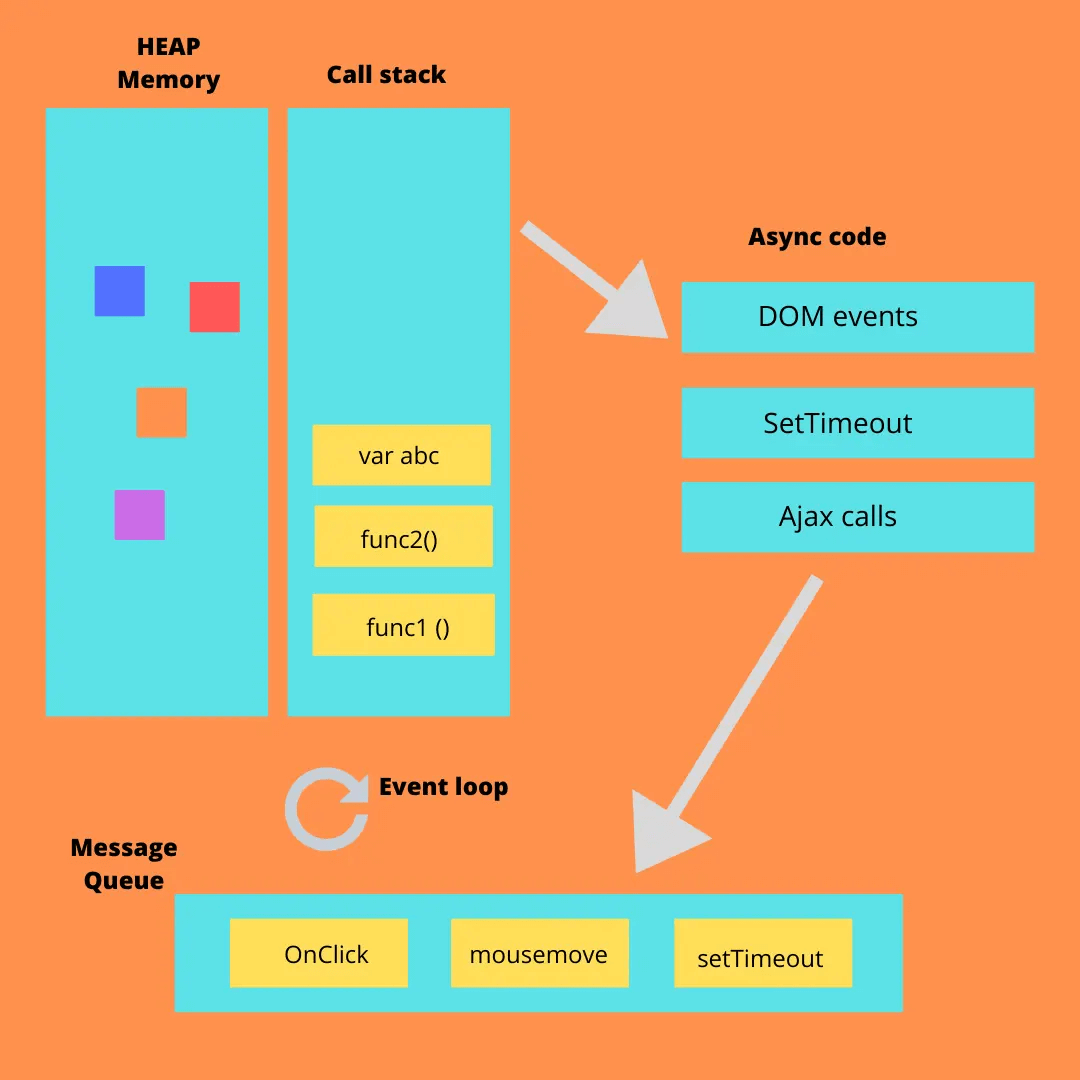
**Một số thành phần trong kiến trúc của Javascript**

Trong Javascript, bạn sẽ bắt gặp một số thành phần cơ bản như memory heap, stack, call stack, web APIs, message queue và event loop.

Mỗi thành phần sẽ đảm nhiệm một nhiệm vụ riêng. Memory Head là nơi mà Javascript sẽ lưu trữ các đối tượng, function được tạo. Stack là để lưu trữ static data như là các giá trị kiểu nguyên thủy (number, string…)

Còn call stack là một cơ chế để Javascript theo dõi các function được thực thi. Nó chính là một phần trong cơ chế Event Loop mà chúng ta sẽ tìm hiểu trong bài viết này.

Web API là những API được tích hợp sẵn vào trong trình duyệt. Những API này cho phép chương trình của bạn tương tác với hệ thống thông qua trình duyệt, ví dụ: [geolocation API](https://developer.mozilla.org/en-US/docs/Web/API/Geolocation_API), WebGL API… Vì những API được tích hợp vào trình duyệt nên không phải trình duyệt nào cũng giống hỗ trợ hết tất cả.



**Message queue**

Về cơ bản, message queue là một storage. Nó là nơi để Javascript lưu trữ các “message” cần xử lý. Mỗi message này là các callback được sử dụng trong các hàm bất đồng bộ, ví dụ như hàm setTimeout(…) và các events do người dùng kích hoạt, ví dụ như các nhấp chuột, sự kiện nhập liệu từ bàn phím…

Khi có bất kỳ [hàm bất đồng bộ](https://vntalking.com/toan-tap-ve-javascript-async-await-tai-sao-lai-nen-dung.html) nào được thực thi hoặc các events xảy ra, Javascript sẽ gửi chúng tới call stack. Từ đây, Javascript sẽ tiến hành gửi xem xét và gửi tới các web API phù hợp để xử lý, sau khi hoàn thành thì nó sẽ gửi một message được liên kết với callback function tới message queue.

Những message được lưu trong message queue cho tới khi call stack không còn gì cả. Khi call stack rỗng, message đầu tiên trong queue sẽ được đẩy vào trong call stack. Call stack sẽ gọi call back và thực hiện những đoạn mã trong đó.

Có một điểm quan trọng về message queue mà bạn cần phải lưu ý. Đó là, call stack hoạt động theo cơ chế LIFO (vào trước – ra sau). Còn queue thì ngược lại, queue theo nguyên táce FIFO (vào trước – ra trước). Nên với message queue, message nào vào trước thì sẽ được gọi thực hiện trước.

**Giải thích dễ hiểu về cách hoạt động của Message Queue**

Chúng là sẽ lấy hàm setTimeout là ví dụ minh họa nhé.

Khi chúng ta sử dụng hàm setTimeout(), Javascript sẽ gửi nó tới call stack để thực thi. Quá trình thực thi nó, JS sẽ tạo mới một timer. Timer này sẽ được gửi tới Web API thích hợp. và API này sẽ bắt đầu đếm ngược.

Khi bộ đếm ngược về 0, Web API sẽ gửi call back của hàm setTimeout() tới message queue. Callback sẽ cứ được lưu và chờ đợi trong message queue cho tới khi call stack rỗng. Khi call stack rỗng, Javascript sẽ lấy call back trong message queue và đẩy nó vào call stack, khi vào call stack thì call back của setTimeout sẽ được thực thi.

* // Sử dụng hàm setTimeout để tạo delay
* //trước khi thực hiện một hàm nào đó
* setTimeout(function cb() {
* console.log('Hello.')
* }, 500)
* // Bước 1: thêm vào call stack etTimeout(function cb() { console.log('Hello.') }, 500)
* // Bước 2:
* // gửi cb() tới web API và xóa setTimeout khỏi call stack, đồng thời tạo bộ đếm: 500
* // Bước 3: Khi bộ đếm về 0, gửi cb() tới message queue và xóa nó khỏi Web API
* // Bước 4: Khi call stack rỗng, gửi lấy cb() từ message queue và gửi tới call stack
* // xóa cb() trong message queue

**Call stack, message queue and priorities**

Trong Javascript, call stack có độ ưu tiên cao hơn message queue. Tức là, message queue phải đợi khi call stack rỗng trước rồi mời có thể đẩy bất cứ thứ gì từ message queue sang call stack.

Vậy khi nào thì call stack rỗng?

Call stack rỗng khi tất cả các functions bên trong nó được thực thi xong.

**Cách xử lý của message queue**

Tại một thời điểm, Message queue chỉ có thể xử lý được một message. Nếu một message mà tốn nhiều thời gian xử lý thì những message sau phải chờ thôi.

Nguyên lý này gọi là run-to-complition hay còn gọi là zero-delay.

Quay trở lại hàm setTimeout(), nếu bạn thiết lập giá trị delay = 0 thì callback sẽ được thực thi ngay lập tức khi và chỉ khi nó là message đầu tiên trong message queue, nếu khi thì vẫn phải đợi như bình thường.

**Javascript Event loop**

Đọc đến đây, chắc hẳn bạn đã hiểu phần nào cách Javascript xử lý các hàm bất đồng bộ rồi đúng không? Thực chất nó là sự tương tác nhịp nhàng, chuyển message giữa call stack, message queue và web API. Mặc dù, bản thân Javascript là đơn luồng nhưng JS có thể làm được điều này là do các Web API chạy trên luồng riêng biệt.

Vậy Event Loop là gì? Đóng vai trò gì trong quá trình xử lý các hàm bất đồng bộ?

Thực ra, nếu bạn để ý kỹ quy trình hoạt động giữa call stack và message queue bạn sẽ tìm ra câu trả lời. Làm sao Javascript biết được call stack lúc nào là rỗng?

Câu trả lời chính là Event loop. Chính cơ chế event loop sẽ liên tục kiểm tra xem call stack có rỗng hay không? Nếu nó rỗng, nó sẽ lấy message đầu tiên trong message queue và đẩy vào call stack.

Vậy là rõ rùi đúng không?