เอกสารแจกฟรี

เสียดายไม่ได้อ่าน

จาวาสคริปต์ฝั่งเซิร์ฟเวอร์

(Node.js ฉบับย่อ)

เล่ม 1

โดย แอดมินโฮ โอน้อยออก

ประวัติการแก้ไข

ครั้งที่	วันที่	รายละเอียดการแก้ไข
1	30 ธ.ค. 2558	เริ่มสร้าง และเผยแพร่ผลงาน
2	1 ม.ค. 2559	แก้เนื้อหาที่ผิด
3	8 ม.ค. 2559	เพิ่มเนื้อหา
4	27 ม.ค. 2559	แก้เนื้อหาที่ผิดตามคำแนะนำของคุณ Sarin Achawaranont

ถ้าท่านดาวน์โหลดทิ้งไว้นาน แล้วเพิ่งมาเปิดอ่าน ก็ขอรบกวนให้โหลดใหม่อีกครั้งที่

http://www.patanasongsivilai.com/itebook_form.html

เผื่อผมอัพเดตแก้ไข pdf ตัวใหม่เข้าไป หรือใครไปดาวน์โหลดมาจากที่อื่น ก็อาจพลาดเวอร์ชั่นใหม่ล่าสุดได้ครับ

และรบกวนช่วยกรอกแบบสอบถาม **ตามลิงค์ข้างบนด้วยนะครับ**

แอดมินโฮ โอน้อยออก

(จตุรพัชร์ พัฒนทรงศิวิไล)

30 ธันวาคม 2558

ถ้าสนใจเกี่ยวกับเพจด้านไอที ก็ติดตามได้ที่ https://www.facebook.com/programmerthai/

คำนำ

JAVASCRIPT

(ECMAScript)

สืบเนื่องมาจากผมได้แต่งหนังสือ**จาวาสคริปต์** (JavaScript) มาตรฐานตัวใหม่ ECMAScript 2015 หรือเรียกสั้น ๆ ว่า **“ES6” หรือ “ES6 Harmony”** (ประกาศออกมาล่าสุด เมื่อกลางเดือนมิถุนายน พ.ศ. 2558) **โดยเล่มนี้จะตีพิมพ์จำหน่ายทั่วประเทศ** (กำลังอยู่ในกระบวนการผลิต)

สำหรับหนังสือเล่มนี้ที่ท่านเปิดอ่าน ผมตั้งใจจะแจกจ่าย**ฟรี** เพื่ออธิบายจาวาสคริปต์ในอีกมุมมองหนึ่ง ซึ่งจะขยายเนื้อหามาจากหนังสือที่ผมเขียนไว้ขาย โดยจะแสดงให้เห็นว่า จาวาสคริปต์ไม่ได้จำกัดแค่การทำงานอยู่บนเว็บบราวเซอร์ (Web browser) เท่านั้น แต่มันยังทำงานอยู่ฝั่งเซิร์ฟเวอร์ได้ ด้วยการใช้ Node.js ไม่ต่างอะไรกับภาษาสคริปต์ดัง ๆ เช่น PHP, ASP หรือ JSP เป็นต้น ...แจ่วป่าว

และถ้ามีเวลา ผมก็อยากเขียนในแง่มุมอื่น เพื่อแสดงให้เห็นถึงความสามารถต่าง ๆ ที่ซ่อนเร้นอยู่เยอะมากในจาวาสคริปต์ เสมือนเป็น “การเปิดโลกจาวาสคริปต์ เดอะซีรีย์” (ถ้าเป็นไปได้นะครับ)

โดยผมเองก็ยอมรับตรง ๆ ว่า เพิ่งเริ่มศึกษามันอยู่เช่นกัน ดังนั้นหากเนื้อหามีอะไรผิดพลาดไป เช่น ให้ข้อมูลผิด สะกดอะไรผิดไป มุมแป๊กบ้าง ขำบ้าง หรืออ่านแล้วมึนงงไป 7 วัน เป็นต้น ผมก็ขออภัยมา ณ โอกาสนี้ด้วย และถ้าคุณเข้าใจ ไม่เข้าใจยังไง ก็สามารถชี้แนะผมได้ตลอดเวลา

ที่สำคัญผมลองเขียนเป็นน้ำจิ้มเล็กน้อยก่อน เป็นแค่พื้นฐานเบื้องต้นเท่านั้น ทฤษฎีคงไม่ได้เจาะลึกอะไรมาก และก็ยังเขียนไม่เสร็จดีด้วย 555+

...แต่ก็ตั้งใจจะทยอยเขียนเรื่อย ๆ พร้อมทั้งหมั่นอัพเดตเนื้อหา ขึ้นอยู่กับเวลาโอกาส และความสามารถจะอำนวย

หนังสือเล่มนี้เหมาะกับใคร

ก่อนจะอ่านหนังสือเล่มนี้ ผมต้องถามว่า ...คุณสนใจสิ่งเหล่านี้หรือไม่ ?

- ต้องการสร้างเว็บแอพพลิเคชั่น (Web application) โดยใช้แค่ 3 ภาษาเท่านั้น ได้แก่ HTLM, CSS และใช้จาวาสคริปต์
- ไม่ใช้ภาษาสคริปต์ดัง ๆ เช่น PHP, ASP และ JSP เป็นต้น
- เวลาจะรันสคริปต์ คุณไม่ต้องติดตั้งซอฟต์แวร์ซึ่งทำหน้าที่เป็นเว็บเซิร์ฟเวอร์ (Web server) เช่น XAMPP, IIS และ Apache Tomcat เป็นต้น เพื่อใช้รันไฟล์สคริปต์ PHP, ASP และ JSP ตามลำดับ
- ต้องการเขียนเว็บแอพพลิเคชั่นแบบเรียลไทม์ พร้อมทั้งรองรับโหลดเยอะ ๆ ไม่ว่าจะเป็นการติดต่อเข้ามาของไคลเอนต์จำนวนมาก รวมทั้งยุ่งเกี่ยวกับข้อมูลมหาศาล (Big Data)
- คุณไม่อยากปวดหัว เวลาเขียนโปรแกรมเพื่อแตกเทรด (thread)
- และคุณอยากรู้ว่าทำไม Node.js ถึงทำให้วงการ Back-end **สั้นสะเทือนถึงดวงดาว?**

“ถ้าคุณสนใจสิ่งเหล่านี้ ก็อ่านต่อได้เลยครับ”

ก่อนจะอ่านหนังสือเล่มนี้

1) คุณต้องมีพื้นฐานของคอมพิวเตอร์มาก่อน ...ก็แงละซิ เพราะเนื้อหาเล่มนี้ฮาร์ดคอคอมล้วน ๆ ถ้าไปให้นักกฎหมาย หมอ บัญชี สถาปนิก นักบิน พวกไม่ใช่สายคอมอ่าน คงงงเป็นไก่ตาแตกแหละน๊า
2) ควรรู้ภาษาโปรแกรมที่ใช้เขียนหน้าเว็บ ได้แก่ CSS กับ HTML เพราะผมกล่าวถึงมันด้วย
3) ต้องมีพื้นฐาน JavaScript มาก่อน หรืออย่างน้อยก็น่าจะรู้จักภาษา C, C++, C# และ Java เป็นต้น ก็น่าจะเข้าใจโค้ดในหนังสือได้ ไม่ยากเย็นอะไรนัก
4) ควรศึกษา ES6 (จาวาสคริปต์ตัวใหม่) มาบ้าง เพราะอย่างน้อยผมก็พูดถึงนิดหน่อย
5) ไปที่ลิงค์ http://www.patanasongsivilai.com/itebook_form.html เพื่อทำการดาวน์โหลด “วิธีติดตั้ง Node.js และ npm เบื้องต้น” แล้วขอร้องเถอะครับ กรุณาอ่านมันก่อน มิฉะนั้นเดี๋ยวคุณจะงงแล้วมาแอบนินทาผมในใจว่า ...เอ็งเขียนอะไรของมันฟะ ไม่รู้เรื่อง

อธิบายเกี่ยวกับ ES6

ในปัจจุบันองค์กร Ecma International (องค์กรจัดการมาตรฐานแห่งยุโรป) จะเป็นผู้กำหนดมาตรฐานจาวาสคริปต์ ซึ่งมาตรฐานของมันจะมีชื่อเรียกว่า “ECMA-262” ส่วนตัวภาษาจาวาสคริปต์นั้น จะมีชื่อเรียกเต็มยศอย่างเป็นทางการว่า “ภาษา ECMAScript“

สำหรับจาวาสคริปต์ไม่ได้ออกเวอร์ชั่นล่าสุดมานานเกือบ 6 ปี และล่าสุดเมื่อกลางเดือนมิถุนายน พ.ศ. 2558 องค์กร Ecma International ได้ออกมาตรฐานจาวาสคริปต์ตัวใหม่เป็น ECMAScript รุ่นที่ 6 ซึ่งชื่อเต็ม ๆ ของมันคือ ECMAScript 2015 แต่ส่วนใหญ่จะเรียกสั้น ๆ ไปเลยว่า ES6 (มันยังมีชื่อเล่นอีกชื่อคือ ECMAScript Harmony หรือจะเรียกว่า ES6 Harmony ก็ได้เช่นกัน)

และแน่นอนครับ Node.js ก็ต้องรองรับ ES6 ได้ด้วยเช่นกัน แต่ผมคงไม่ได้ลงลึกเกี่ยวกับมันมาก เพราะ ES6 เนื้อหามันค่อนข้างเยอะ และผมก็แยกเขียนอีกเล่มหนึ่ง โดยเน้นทฤษฎีโดยเฉพาะ มันจึงนอกประเด็นของหนังสือเล่มนี้พอควร

แนะนำ Node.js

ประวัติ Node.js

ขอสรุปมาจาก https://en.wikipedia.org/wiki/Node.js (ถ้าข้อมูลผิดพลาดโทษ wiki นะ อิ ๆ ๆ)

Node.js ถูกสร้างขึ้น และเผยแพร่ให้ใช้งานบน Linux ในปี ค.ศ. 2552 โดยคุณ “Ryan Dahl” พร้อมเพื่อนทำงานของเขาที่บริษัท Joyent

Ryan Dahl ได้แรงบัลดาลใจหลังจากเห็นแถบสถานะ progress bar บน Flickr เวลาอัพโหลดไฟล์ ซึ่งเว็บบราวเซอร์จะไม่รู้ว่าไฟล์กำลังถูกอัพโหลด และติดต่อกับเซิร์ฟเวอร์ แต่ทว่าเขาต้องการวิธีที่ง่ายกว่านั้น จึงเกิดเป็น Node.js ขึ้นมา

และเขาได้นำ Node.js ไปพูดครั้งแรก ที่การประชุม JSConf เมื่อ 8 พ.ย. 2552 อีกด้วย ลองดูคลิปที่เข้าบรรยายได้เลยครับ

https://www.youtube.com/watch?v=ztspvPYybIY

รูปภาพคุณ Ryan Dahl ผู้สร้าง Node.js
(ที่มาภาพ https://en.wikipedia.org/wiki/Node.js#/media/File:Ryan_Dahl.jpg)

Nodes.js คืออะไร

ให้คุณลืมประวัติความเป็นมาไปก่อน
มันไม่ค่อยสัมพันธ์กับความหมายของ Node.js เท่าไร

สำหรับเว็บบราวเซอร์ต่าง ๆ ไม่ว่าจะเป็น Internet Explorer (IE), Firefox และ Google Chrome เป็นต้น พวกมันจะมีจาวาสคริปต์เอนจิน (JavaScript engine) ติดตั้งอยู่ภายใน และใช้เป็นตัวแปลภาษา และประมวลผลโค้ดที่เขียนด้วยจาวาสคริปต์ให้ทำงาน

ส่วน Node.js ก็สามารถรันจาวาสคริปต์ได้เช่นกัน ไม่ต่างอะไรกับเว็บบราวเซอร์
พูดอีกนัยหนึ่งเรารันจาวาสคริปต์ นอกเว็บบราวเซอร์ได้ด้วย Node.js (อายย่ะ)

นับตั้งแต่ Node.js อุแว้ อุแว้ เกิดขึ้นมาบนโลกแห่งนี้ มันจึงเหมือนติดปีกให้จาวาสคริปต์**บินได้** จริง ๆ เลยนะ

นิยาม

คราวนี้ผมจะให้ดูนิยาม Node.js อย่างเป็นทางการ จากเว็บไซต์ของมัน (https://nodejs.org/en/)

> Node.js® is a JavaScript runtime built on Chrome's V8 JavaScript engine. Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient.

ซึ่งผมได้พยายามแปล ให้มันสละสลวยเท่าที่จะทำได้ดังนี้

ไม่ใช่รายการ AF หรือ “ทรู อะคาเดมี่ แฟนเทเซีย”

> **Node.js** คือตัวรันไทม์ (runtime) ของภาษาจาวาสคริปต์ โดยมันถูกสร้างขึ้นมาบน V8 ซึ่งเป็นจาวาคริปต์เอนจินของ Google Chrome
>
> สำหรับ Node.js จะทำงานแบบ event-driven และเป็น non-blocking I/O ด้วยเหตุนี้จึงทำให้มันเบาวิว และเต็มเปี่ยมไปด้วยประสิทธิภาพ (lightweight and efficient)

จากนิยามที่ผมแปลมาให้ ไม่รู้ว่าคุณจะสงสัยคำศัพท์เหล่านี้หรือไม่ ?

- event-driven
- non-blocking I/O

???

- lightweight

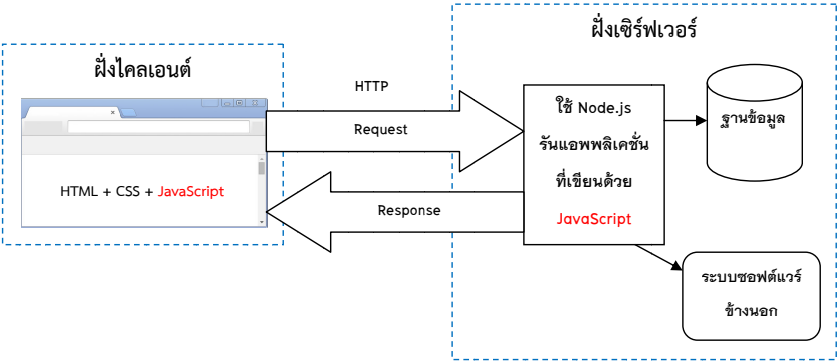
ชื่อเหมือนรุ่นของนักมวย

ถ้าคุณเข้าใจคำศัพท์ที่ผมยกมาให้ดู ก็จะเข้าใจ Node.js ซึ่งผมจะทยอยอธิบายคำพวกนี้ไปเรื่อย ๆ แล้วกัน ...ก็แบบว่า ไม่รู้จะอธิบายยังไง ให้เข้าใจในตอนนี้ 555+

หมายเหตุ V8 จะเป็นจาวาสคริปต์เอนจินที่เป็นโอเพ่นซอร์ส และถูกเขียนขึ้นด้วย C++ พร้อมทั้งถูกติดตั้งไว้ใน Google Chrome ....สำหรับ Node.js ก็ถูกสร้างขึ้นบน V8 ที่ว่านี้แหละ

แล้ว Node.js มันเอาไปใช้ทำงานอะไร

จริง ๆ แล้ว Node.js มันประยุกต์ใช้งานได้หลากหลายนะครับ แต่ที่ผมเห็นส่วนใหญ่จะนิยมนำไปพัฒนาเว็บแอพพลิเคชั่น (Web applications) โดยให้มันทำงานอยู่ฝั่งเซิร์ฟเวอร์ ตามภาพต่อไปนี้



ตามภาพในฝั่งไคลเอนต์ ซึ่งก็คือเว็บบราวเซอร์ จะต้องใช้ 3 ภาษาไฟต์บังคับ ในการพัฒนาหน้าเว็บไซต์ ซึ่ง 3 ภาษานี้ มันเปรียบเสมือนเสาหลักค้ำฟ้าฝั่งหน้าเว็บ โดยจะประกอบไปด้วย

1. ภาษา HTML (ปัจจุบันเป็นเวอร์ชั่น HTML5) เอาไว้แสดงหน้าเว็บไซต์
2. ภาษา CSS (ปัจจุบันเป็นเวอร์ชั่น CSS3) เอาไว้ตบแต่งหน้าเว็บให้สวยงาม
3. จาวาสคริปต์ (ปัจจุบันเป็น ES6) จะทำให้เว็บมันไดนามิก (Dynamic) แลดูยืดหยุ่น และมีชีวิตชีวา

ถ้าคุณมองดูฝั่งเซิร์ฟเวอร์ในภาพที่ผมวาดให้ดู ก็จะเห็นว่าแอพพลิเคชั่นถูกพัฒนาด้วยจาวาสคริปต์ ที่ถูกรันด้วย Node.js โดยปราศจากภาษาสคริปต์ดัง ๆ เช่น PHP, ASP และ JSP เป็นต้น

สำหรับการพัฒนาเว็บแอพพลิเคชั่นด้วยจาวาสคริปต์ล้วน ๆ เขาจะเรียกว่า “Full Stack JavaScript” และในเว็บฝรั่ง ผมก็เห็นเขาเรียกอีกคำหนึ่งว่า “Isomorphic JavaScript” ด้วยนะ (Isomorphic ที่แปลว่า “ซึ่งมีรูปร่างสัณฐานเหมือนกัน”)

...แต่ถึงกระนั้นก็ดี การสร้างหน้าตาเว็บ (UI) ยังต้องพึ่งพาภาษา HTML กับ CSS อยู่ดี

หมายเหตุ Node.js สามารถทำงานตามลำพัง (Stand alone) เป็นแอพพลิเคชั่นโดด ๆ โดยไม่ต้องทำเป็นเว็บแอพพลิเคชั่นก็ได้นะ

ปุจฉา จำเป็นต้องใช้ Node.js ในฝั่งเซิร์ฟเวอร์หรือไม่?

คำตอบ ไม่จำเป็น เพราะมันมีภาษาทางเลือกเยอะ นอกจากจาวาสคริปต์ 

อีกทั้งตัว Node.js ไม่ใช่แฟนทำแทนทุกเรื่องไม่ได้ เพราะมันไม่ได้แก้ปัญหาการเขียนโปรแกรมฝั่งเซิร์ฟเวอร์ได้ครอบจักรวาลหรอกนะ แต่ Node.js จะตอบโจทย์กรณีที่งานหลังบ้านมีโหลดเยอะ ๆ เวลาติดต่อกับ IO เช่น ไคลเอนต์ติดต่อเข้ามาเยอะมาก ๆ ติดต่อฐานข้อมูลถี่ ๆ หรืออ่านเขียนข้อมูลจากฮาร์ดดิสก์หนัก ๆ เป็นต้น ซึ่งมันจะทำงานได้เร็ว ไหลลื่น ปิ๊ดปาด ...จนเป็นเครื่องหมายการค้าเลยนะ

ด้วยเหตุนี้มันจึงเหมาะจะใช้สร้างเว็บแอพพลิเคชั่นแบบเรียลไทม์เอามาก ๆ (Real time web applications) หรือแอพพลิเคชั่นเครือข่ายที่ขยายขนาดได้ (โทษทีครับ แปลตรงตัวไปหน่อยจากคำว่า “scalable network applications”)

แต่ถึงกระนั้นก็ดีการเรียนรู้ Node.js ก็เหมือนเปิดโลกทัศน์ของภาษาจาวาสคริปต์ ในมุมมองที่คุณไม่เคยเห็นมาก่อน อย่างน้อยก็ “รู้ไว้ใช่ว่า ใส่บ่าแบกหาม” เพราะจาวาสคริปต์ในตอนนี้ คุณคงไม่เถียงนะครับว่า ...มันผูกขาดการพัฒนาเว็บแอพพลิเคชั่นฝั่งไคลเอนต์ เป็นที่เรียบร้อยโรงเรียนจีน ...เลี้ยว แถม Node.js ก็ยังเกิดมาเพื่อสั้นสะเทือนวงการ Backend อีกด้วย

JAVASCRIPT
(ECMAScript)

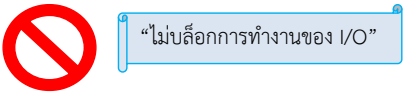
“ถ้า ...ชนใดไม่มีดนตรีกาล ในสันดานเป็นคนชอบกลนัก
แล้วโปรแกรมเมอร์พัฒนาเว็บแอพพลิเคชั่น
ไม่รู้จักจาวาสคริปต์ ก็เป็นคนชอบกลนัก ...เช่นเดียวกัน”

แอบกระซิบนิดหนึ่ง ตัว Node.js เวลาติดตั้งมันจะเล็กนิดเดียว และไม่ต้องติดตั้งซอฟต์แวร์ซึ่งทำหน้าที่เป็นเว็บเซิร์ฟเวอร์ให้เสียเวลาอีกด้วย ดังนั้นผู้สร้างเขาจึงกล้าบอกว่า Node.js มัน **lightweight** หรือเบาวิวนั้นเอง

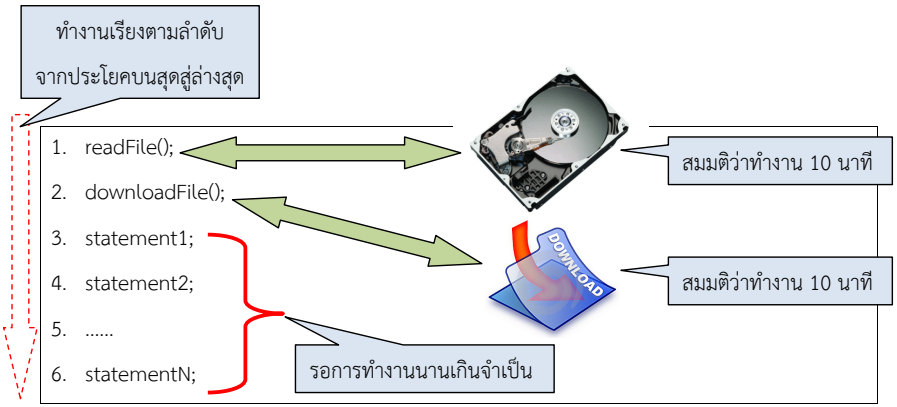
แนวคิดการทำงานของ Node.js

ก่อนอื่นต้องพูดถึงคำว่า I/O ชื่อเต็มมันคือ “input/output” ถ้าแปลเป็นไทยก็คือ “รับเข้า/ส่งออก” ซึ่งในที่นี้จะหมายถึง การรับเข้าและส่งออกข้อมูลจากโลกภายนอก เช่น การเปิดพอร์ตเพื่อรอให้ไคลเอนต์ติดต่อเข้ามา การดาวน์โหลดไฟล์ การอ่านไฟล์จากฮาร์ดดิสก์ และการติดต่อฐานข้อมูล เป็นต้น

เวลาคุณ “คิดจะพัก คิดถึง คิทแคท” ...เฮยไม่ช่าย คิดจะใช้งาน Node.js ต้องคิดแบบ non-blocking I/O ซึ่งประโยคนี้ถ้าแปลเป็นไทย อาจได้ใจความว่า ...



จากนิยามดังกล่าว อาจฟังดูแล้วงง ...แต่เดี๋ยวผมจะขอยกตัวอย่างปัญหา ที่ถือว่าใหญ่มาก ๆ ๆ ๆ เมื่อเกิดเหตุการณ์บล็อกการทำงานของ I/O ดังตัวอย่างโค้ดง่ายง๊าย ดังต่อไปนี้



ในโค้ดตัวอย่างที่เห็น มันจะทำงานเรียงตามลำดับ จากประโยคบนสุดสู่ล่างสุด โดยมีรายละเอียดดังนี้

- บรรทัดที่ 1 จะสมมติว่าอ่านไฟล์นาน 10 นาที (ติดต่อ I/O)
- บรรทัดที่ 2 จะสมมติว่าดาวน์โหลดไฟล์นาน 10 นาที (ติดต่อ I/O)
- สำหรับบรรทัดที่ 2 ต้องรอให้บรรทัด 1 ทำงานเสร็จก่อน ซึ่งเบ็ดเสร็จแล้วประโยคบรรทัดที่ 1 กับ 2 จะทำงานรวมกันทั้งสิ้น 20 นาที ...นานจนนั่งงีบไปรอบหนึ่งได้

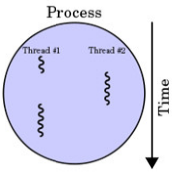
คุณจะเห็นว่าบรรทัด 1 มันบล็อกการทำงานของบรรทัด 2 (บล็อกการทำงานของ I/O) ซึ่งส่งผลทำให้ ฮึ ๆ ๆ ประโยคคำสั่งที่เหลือตามมาคือ statement1, statement2 จนถึง statementN ต้องรอนานขึ้นโดยไม่จำเป็น

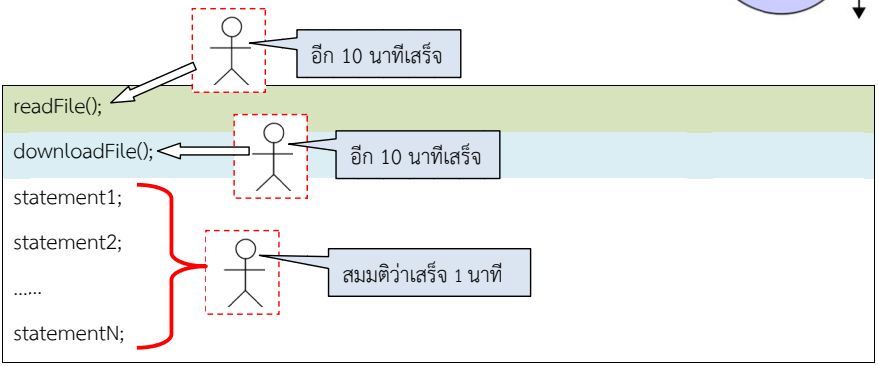
แล้วจะแก้ปัญหาอย่างไรดีละทีเนี่ย น่านะปัญหาทุกอย่างมีทางออก ถ้าออกไม่ได้ก็ให้ไปที่ทางเข้า ...อุ้ยล้อเล่น ให้คุณดูวิธีแก้ปัญหาในหัวข้อถัดไปนะคร้าบ 

แก้ปัญหาด้วยการใช้เทรด

จากปัญหาในหัวข้อก่อนหน้านี้ สามารถแก้ไขได้ด้วยการใช้เทรด (Thread) ซึ่งก็คือโพรเซส (Process) ย่อย ๆ โดยอาจเปรียบเทียบเทรด มันก็คือ**คนงาน** ส่วนโค้ดก็เหมือน**งาน** ที่ต้องมีคนงานมาทำอีกที

สำหรับบางภาษา เช่น Java และ C++ เป็นต้น สามารถเขียนให้มีเทรดมากกว่า 1 ตัว (แตกเทรด) ซึ่งสามารถทำงานคู่ขนานพร้อม ๆ กันได้ ในเวลาเดียวกัน ดังตัวอย่างโค้ดข้างล่าง



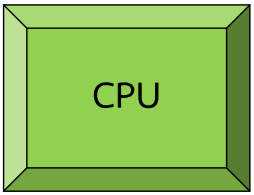


ในภาพจะมีเทรด 3 ตัว ซึ่งก็เหมือนคนงาน 3 คน ที่มาทำงานโค้ด (ประมวลผล) ดังรายละเอียดต่อไปนี้

- เทรดคนแรก จะประมวลผลประโยค readFile();
- เทรดคนที่สอง จะประมวลผลประโยค downloadFile();
- เทรดคนที่สาม จะประมวลผลประโยค statement1, statement2 จนถึง statementN

โดยเทรดทั้งสามตัวดังกล่าว จะทำงานคู่ขนานไปพร้อม ๆ กัน ไม่ได้ทำงานเรียงตามลำดับ ซึ่งส่งผลทำให้ประโยค statement1, statement2 จนถึง statementN ทำงานเสร็จก่อน (สมมติว่าเสร็จภายใน 1 นาที) ขณะที่ประโยค readFile(); และ downloadFile(); ทั้งคู่จะทำงานเสร็จภายใน 10 นาทีให้หลัง (ไม่ใช่เวลารวม 20 นาที) ด้วยแนวคิดเช่นนี้จึงสามารถแก้ปัญหา จะไม่เกิดเหตุการณ์บล็อกการทำงานของ I/O

การที่เทรดมันทำงานคู่ขนานไปพร้อม ๆ กันได้ ก็เพราะตัว CPU เป็นคนคอยบริหารจัดการเทรดให้ทำงานคู่ขนาน ซึ่งจะเรียกความสามารถนี้ของ CPU ว่า “Multithreading” แต่รายละเอียดเบื้องหลังการทำงาน คงต้องไปหาอ่านในวิชา OS (Operating System) แล้วละครับว่า ...มันทำงานยังไง แต่ถ้าคุณสนใจก็ลองดูได้ที่ http://www.no-poor.com/dssandos/os_ch03_process.htm



แต่การเขียนโปรแกรมให้มีหลาย ๆ เทรด ทำงานพร้อมกัน มันจะเขียนยุ่งยากมากแบบยกกำลังสอง[2] นะซิ ...เพราะมันเสี่ยงให้เกิดปัญหาหลาย ๆ อย่าง เช่น ปัญหาแต่ละเทรดแย่งชิงทรัพยากรกันเอง (Race condition) หรือแต่ละเทรดต่างรอคอยการทำงานซึ่งกันและกัน (Deadlock) เป็นต้น

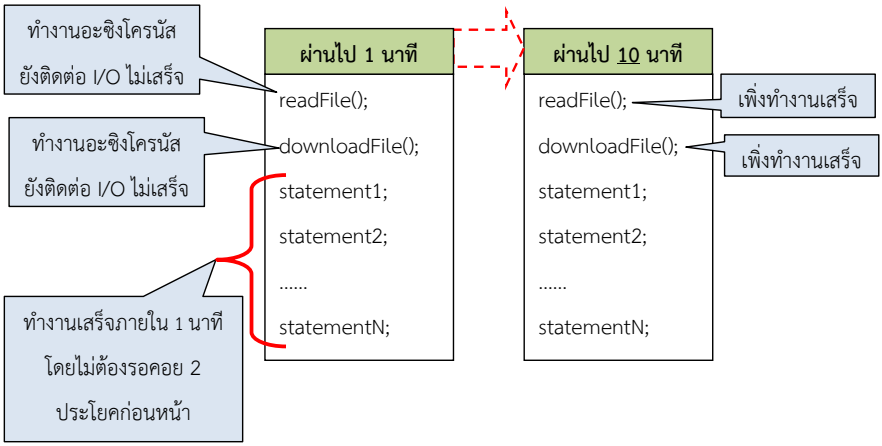
ด้วยเหตุนี้ Node.js จึงมีเพียงแค่เทรดเดียว (Single Thread) ...ไม่มีการสร้างเทรด และแตกเทรด ให้ปวดเศียรเวียนเกล้าเหมือนภาษาอื่นเขา

คุณอาจสงสัยแล้วว่า ทำไมผมถึงอธิบายเทรดให้ยืดยาว เสียเวลาทำไม ? ในเมื่อ Node.js เขียนโปรแกรมแตกเทรดไม่ได้ ...จริง ๆ แล้วผมต้องการสื่อถึงคนที่มาจากภาษาอื่น ที่เขามีเทรดใช้งานกัน เมื่อเปลี่ยนมาใช้ Node.js มันจะเป็น**เขตปลอดการแตกเทรด** แต่ก็ทำให้เกิด non-blocking I/O ได้เหมือนกันนะคร๊าบบบบ

เปลี่ยนมุมมองการเขียนโปรแกรมเสียใหม่

เวลาเขียนโปรแกรมใน Node.js จะต้องเขียนแบบอะซิงโครนัส (Asynchronous programming) เพื่อไม่ให้เกิดการบล็อกการทำงานของ I/O (อย่าเพิ่งขมวดคิ้วนะครับ)

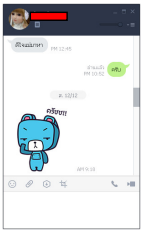
คำว่า “อะซิงโครนัส” อาจแปลตรงตัวได้ว่า “การทำงานที่ไม่พร้อมกัน” แต่ถ้าใช้ในความหมายของ**การเขียนโปรแกรมแบบอะซิงโครนัส** ก็อาจหมายถึง การทำงานของโปรแกรม ที่ไม่ต้องรอคอยให้ประโยคคำสั่งใดคำสั่งหนึ่งทำงานเสร็จก่อน ประโยคอื่นที่ตามทีหลัง มันสามารถทำงานได้ทันที ...ตามภาพหน้าถัดไปครับ



จากโค้ดข้างบน (เหมือนในหัวข้อก่อนหน้านี้) ถ้าเป็น Node.js จะทำงานแบบอะซิงโครนัสดังต่อไปนี้

1. คุณไม่ต้องรอให้ readFile(); ทำงานเสร็จก่อน หรือกล่าวอีกนัยหนึ่งไม่ต้องรอให้ฟังก์ชั่นติดต่อ I/O จนสำเร็จ แต่สามารถข้ามไปทำข้อ 2 ได้เลย (อีก 10 นาที จะทำงานเสร็จ)
2. ตรงประโยค downloadFile(); ก็ไม่ต้องรอให้ติดต่อ I/O เสร็จก่อนเช่นกัน จึงสามารถข้ามไปทำข้อ 3 ก่อนได้เลย (อีก 10 นาที จะทำงานเสร็จ)
3. ด้วยเหตุนี้ประโยค statement1, statement2 จนถึง statementN จึงสามารถทำงานเสร็จก่อนได้ภายใน 1 นาที (เป็นกรณีสมมติ)
4. พอหลังจาก 10 นาทีผ่านพ้นไป ก็เพิ่งจะมาทำประโยคในข้อ 1 กับ 2 เสร็จทีหลัง

หวังว่าเมื่อถึงตอนนี้ คุณคงเห็นภาพการทำงานแบบอะซิงโครนัสของ Node.js มันจะไม่บล็อกการทำงานของ I/O นะครับ ...ด้วยเหตุนี้คำว่า “non-blocking I/O” อาจเรียกใหม่เป็น “asynchronous I/O” ก็ได้เช่นกัน



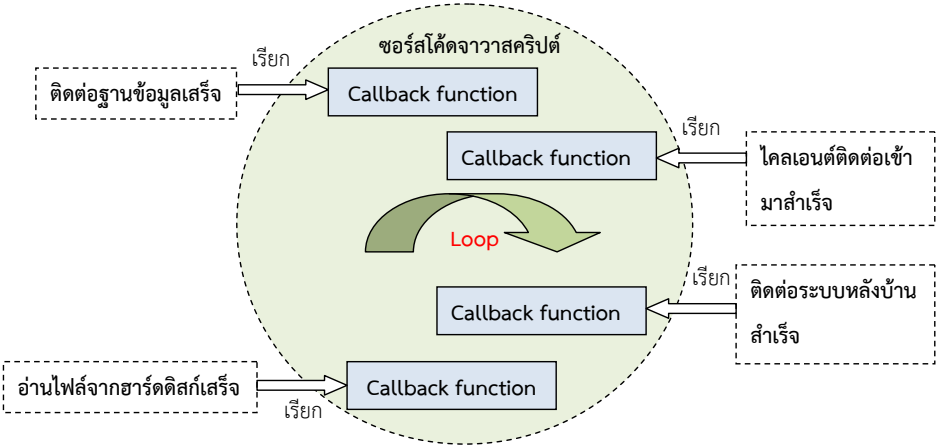
ถ้าจะเปรียบเทียบการทำงานแบบซิงโครนัส ก็เหมือนโทรศัพท์คุยกับแฟนที่คบกันใหม่ ๆ ห้ามวางสายทันที เดี๋ยวมันงอน ต้องคุยให้เสร็จ แล้วถึงจะไปเข้าห้องน้ำ กินข้าว ...บลา ๆ

ส่วนการทำงานแบบอะซิงโครนัส ก็เหมือนคุณส่งไลน์หาเขา/หล่อน แล้วเราก็แอบหนีไปทำธุระส่วนตัวส่วนตั้วก่อนได้ หลังจากนั้นค่อยมาเปิดอ่านไลน์ แล้วทักมันกลับไปอีกครั้ง

แนวคิด Event-driven

เพื่อให้ Node.js ทำงานแบบอะซิงโครนัสได้ เขาจะใช้แนวคิด Event-driven ซึ่งได้รับอิทธิผล และคล้ายกับ Event Machine ของ Ruby หรือ Twisted ของ Python (ไม่ต้องซีเรียสกับคำศัพท์ที่ผมยกมาก็ได้นะครับ)

Event-driven ถ้าแปลตรงตัวก็คือ “ถูกขับเคลื่อนด้วยเหตุการณ์” ซึ่งเหตุการณ์ที่ว่า ก็คือเหตุการณ์ที่ติดต่อกับ I/O ต่าง ๆ นั้นเองแหละครับ เพื่อให้เข้าใจมากขึ้น ผมจะขอเปลี่ยนจากคำว่า “Event-driven” เป็น “Event loop” หรือ “วนลูป รับเหตุการณ์” ซึ่งไม่รู้จะงงกันมากไปกว่านี้หรือไม่ แต่ยังไงก็ให้ดูภาพประกอบแล้วกันนะ



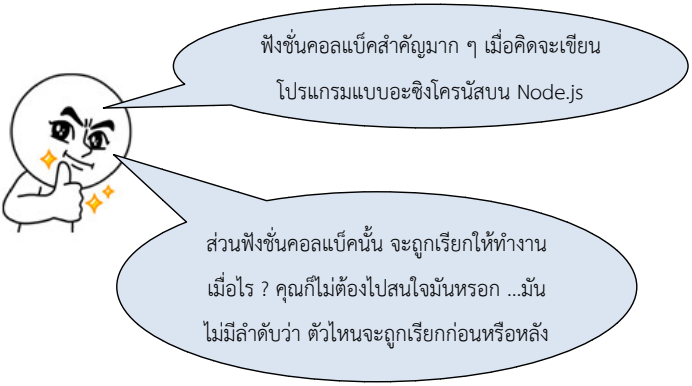
จากรูปภาพนี้ ให้คุณลองจินตนาการว่าเมื่อ Node.js มันประมวลไฟล์จาวาสคริปต์ ซึ่งเบื้องหลังการทำงาน มันจะมีเพียงเทรดเดียวเท่านั้น ที่มาอ่านและทำงานตามโค้ดที่เขียนไว้ ซึ่งเทรดดังกล่าวจะวนลูป (ตั้งแต่เริ่มแรกเลยครับ) เพื่อรับเหตุการณ์ที่ Node.js ต้องติดต่อกับ I/O ต่าง ๆ และเมื่อมันติดต่อเสร็จแล้ว ก็จะมาเรียกฟังชั่นคอลแบ็คที่อยู่ในโค้ดภายหลัง

คำถาม แล้วมันจะหยุดวนลูป เพื่อรับเหตุการณ์ต่าง ๆ เมื่อไร ?

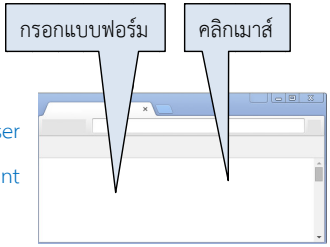
คำตอบ จะหยุดวนลูป เมื่อฟังก์ชั่นคอลแบ็คทุกตัวทำงานเสร็จเรียบร้อยแล้ว ไม่มีคอลแบ็คตัวใดหลงเหลือให้เรียกใช้

...และสิ่งที่คุณต้องรู้อย่างหนึ่งก็คือ ฟังก์ชั่นคอลแบ็คใน Node.js **คุณต้องเขียนเองน้า**...

ผมอยากให้คุณจำง่าย ๆ อย่างนี้แล้วกัน



จริง ๆ แล้ว ถ้าใครเคยเขียนโปรแกรมประเภท GUI (Graphic user interface) หรือใช้จาวาสคริปต์ยุ่งเกี่ยวกับ DOM (Document Object Model) ...น่าจะเข้าใจคอนเซปท์แบบนี้ไม่ยาก




```
<!-- รองรับเหตุการณ์
 เมื่อมีการคลิกเมาส์หน้าเว็บ-->

<button onclick="myFunction()">
    Click me
</button>
```

ผมจะให้คุณนึกถึงเวลาเขียนจาวาสคริปต์บนหน้าเว็บ (HTML) ตอนที่ผู้ใช้คลิกเมาส์ กรอกแบบฟอร์ม และการกระทำต่าง ๆ บนหน้าเว็บ ก็จะเกิดเหตุการณ์ขึ้นมา (Event) ซึ่งจะส่งผลทำให้ฟังก์ชั่นคอลแบ็ค ถูกเรียกให้ทำงานภายหลัง

ซึ่งมันเป็นแนวคิด Event-driven เหมือนกันกับ Node.js ไม่มีผิดเด๊ะเลยคร้าบ ...แต่ทว่ามันจะทำงานแบบอะซิงโครนัสในด้าน UI เป็นหลัก ส่วน Node.js จะทำงานอะซิงโครนัสในด้าน I/O เป็นหลัก

***จริง ๆ แล้ว ยังมีการทำงานแบบอะซิงโครนัสที่เหมือน ๆ กัน ไม่ว่าจะบนหน้าเว็บหรือ Node.js เช่น การหน่วงเวลาด้วยฟังก์ชั่น setTimeout() และ setInterval() เป็นต้น

ทบทวนฟังก์ชั่นคอลแบ็ค

ฟังก์ชั่นในจาวาสคริปต์จะถือว่าเป็น First-class functions (แปลว่า “ฟังก์ชั่นเต็มขั้น” ล้อมาจากคำว่า “First-class citizen” หรือประชาชนเต็มขั้น) หมายความว่า ฟังก์ชั่นจะเป็นข้อมูลตัวหนึ่ง ที่สามารถกำหนดค่าให้กับตัวแปรได้

*** **หมายเหตุ** ฟังก์ชั่นในจาวาสคริปต์ มันยังเป็นอ็อบเจ็กต์อีกด้วย

ด้วยเหตุนี้เราจึงสามารถส่งฟังก์ชั่น ให้เป็นค่าอากิวเมนต์แก่ฟังก์ชั่นตัวอื่นได้ หรือจะรีเทิร์นตัวฟังก์ชั่นออกมาก็ทำได้ด้วย ...ซึ่งจะเรียกคุณสมบัติแบบนี้ว่า “Higher Order Functions” ดังตัวอย่าง

```
function readFile( callback ) {          // ประกาศฟังก์ชั่น ให้มีพารามิเตอร์ชื่อ callback
    callback();                          // เรียกฟังก์ชั่นให้ทำงาน
}
// ประกาศตัวแปรแบบ let ใน ES6
let myFunction = function() {
    console.log('Reading a file from json.txt');
};
readFile(myFunction);                    // แสดงผลลัพธ์เป็น "Reading a file from json.txt"
```

สามารถนำฟังก์ชั่นมากำหนดค่าให้กับตัวแปร myFunction ได้

ในตัวอย่างนี้จะเห็นว่าฟังก์ชั่น readFile(callback){...} จะมีตัวแปรพารามิเตอร์ callback ที่รับค่าอากิวเมนต์เป็นฟังก์ชั่น ซึ่งในกรณีก็คือ myFunction จึงทำให้มันถูกเรียกให้ทำงานภายใน readFile() ด้วยประโยค callback(); ในบรรทัดที่สอง

อย่างที่ท่านทราบ ฟังชั่นที่ถูกใช้เป็นค่าอากิวเมนต์ เขาจะเรียกว่า**ฟังชั่นคอลแบ็ค** (Callback functions) หรือเรียกสั้น ๆ ไปเลยว่า**คอลแบ็ค**ก็ได้ (ขี้เกียจเขียนนะ)

ซึ่งมันแปลตรงตัวได้ว่า “ฟังชั่นที่ถูกเรียกกลับ” และในตัวอย่างดังกล่าวที่ยกมา ก็ชัดเจนดีนะครับว่า

...ฟังก์ชั่น myFunction() ได้ถูกเรียกกลับมาจากฟังก์ชั่น readFile() อีกที

ทบทวน Closures

ตัวอย่างต่อไปนี้ ผมจะขอทบทวนเรื่อง Closures ในจาวาสคริปต์

```
function readFile() {   // ประกาศฟังก์ชั่น
        let filename = 'json.txt';
        return function() {                    // รีเทิร์นฟังก์ชั่นภายใน (Inner functions)
                console.log(`Reading a file from ${filename}`);
        }
}
let myFunction = readFile();
myFunction();  // เรียกฟังก์ชั่นให้ทำงาน และแสดงผลลัพธ์เป็น "Reading a file from json.txt"
```

Closure

ถูกมองเห็น

เทมเพลตสตริง (Template string) ใน ES6 จะใช้เครื่องหมาย back-tick(`) แทนเครื่องหมายคำพูดคู่หรือเดี่ยว (" หรือ ')

เนื่องจากฟังก์ชั่นก็คือชนิดข้อมูลตัวหนึ่ง จากตัวอย่างเมื่อเรียกใช้ฟังก์ชั่น readFile(); ก็จะรีเทิร์นฟังก์ชั่นภายในที่สามารถมองเห็นตัวแปร filename ซึ่งประกาศอยู่ที่ฟังก์ชั่นตัวนอกสุด

...ด้วยเหตุนี้เมื่อเรียก myFunction(); ก็ยังคงเข้าถึงตัวแปร filename อย่างไม่มีปัญหาอะไรเลย

ต้องบอกอย่างนี้นะครับว่า ...ฟังก์ชั่นในจาวาสคริปต์นอกจากประกาศอยู่ในฟังก์ชั่นตัวอื่นได้แล้ว มันยังจำพื้นที่ซึ่งมันเคยอาศัยอยู่ได้ (Context) และมองเห็นตัวแปรที่ประกาศอยู่ในฟังก์ชั่นข้างนอกได้ด้วย

สำหรับฟังก์ชั่นที่ซ้อนอยู่ในฟังก์ชั่นหลัก และจำพื้นที่ซึ่งมันอาศัยอยู่ เขาจะเรียกฟังก์ชั่นแบบนี้ว่า **Closures** ที่แปลว่า “การปิด” ...ก็อาจหมายความว่า ฟังก์ชั่นที่ซ้อนอยู่ภายใน มันจะปิดล้อมตัวแปรที่อยู่นอกขอบเขตได้

สาเหตุที่ผมทบทวนคร่าว ๆ เกี่ยวกับฟังก์ชั่นคอลแบ็ค กับ closure ก็เพราะมันเป็นหัวใจหลักในการเขียนโปรแกรมด้วยจาวาสคริปต์บน Noe.js นะชิ

... แต่ถึง closures จะแปลว่าปิด แต่ก็ห้ามปิดใจไม่ให้เรียนรู้จาวาสคริปต์นะครับ อิ ๆ ๆ

เกร็ดความรู้ ☆

Ryan Dahl เริ่มสร้าง Node.js ด้วยการใช้ภาษา C แต่เมื่อทำไปทำมา พบว่าโค้ดมันซับซ้อนเกินไป จึงไปลองใช้ภาษา Lua แต่ก็ล้มเหลวเช่นกัน ก่อนที่จะมาตกล่องปล่องชิ้นกับจาวาสคริปต์ เนื่องจากมันมี closures กับ first-class functions ซึ่งมันพอเหมาะพอเจาะกับการเขียนโปรแกรมแบบอะซิงโครนัส ด้วยการใช้ Event-driven นั้นเอง

เกริ่นนำมอดูล

"ศัพท์บัญญัติราชบัณฑิตยสถาน" เขาเขียนมอดูลแบบนี้จริง ๆ

มันเป็นมาตรฐานในการโหลดมอดูลต่าง ๆ

มอดูล (Modules) ใน Node.js จะใช้มาตรฐาน CommonJS ...เอาเป็นว่าให้คุณลองนึกถึงไลบรารี่ในการเขียนโปรแกรม (Library) ซึ่งเราจะใช้มันเก็บโค้ดอะไรก็ตามที่ต้องเรียกใช้งานบ่อย ๆ ...และมอดูลก็มีแนวคิดเดียวกัน

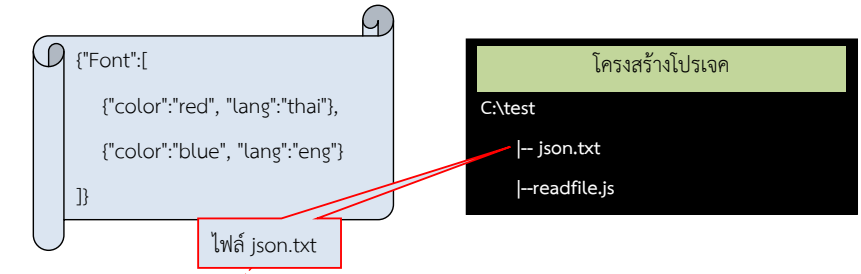
สำหรับมอดูลใน Node.js ส่วนใหญ่จะเก็บ API ซึ่งเป็นฟังก์ชั่นที่ต้องติดต่อกับ I/O แบบอะซิงโครนัส ...ส่วนการโหลดสิ่งที่บรรจุอยู่ในมอดูลมาใช้งาน ก็จะให้ใช้ประโยคคำสั่งดังตัวอย่างต่อไปนี้

```
require('module_name');
// module_name คือชื่อมอดูล
```

require() ก็คือฟังก์ชั่น โดยเราต้องระบุชื่อมอดูลเป็นค่าอากิวเมนต์ แล้วเมื่อนั้น require() จะรีเทิร์นอ็อบเจ็กต์ออกมา โดยค่าพร็อพเพอร์ตี้ (Property) ของมัน ก็คือสิ่งที่เราได้โหลดจากมอดูลเข้ามานั้นเอง

มอดูล ถ้าแปลตรงตัวก็จะได้ว่า "หน่วยที่นำมาประกอบ เป็นสิ่งหนึ่งสิ่งใดขึ้นมา"

จะขอยกตัวอย่างการใช้งานมอดูลดังนี้




```
var fs = require('fs');  // โหลดมอดูล fs เพื่อเอาไว้ใช้อ่านและเขียนไฟล์
fs.readFile(
        'json.txt',        // ชื่อไฟล์
        function(err, buffer) {
                if (err) {
                        console.log(err);    // แสดงข้อความผิดพลาดออกมา
                }
                console.log('Reading a file:', typeof buffer);
        } // สิ้นสุดฟังก์ชั่น
);
console.log('Last statement');
```

มอดูล fs เดี่ยวจะอธิบายอีกครั้งในบทถัดไป (เล่มหน้า)

คอลแบ็ค

ประโยคนี้จะแสดงผลลัพธ์ออกมาทางหน้าคอนโซลก่อนเพื่อน

จากโค้ดตัวอย่างดังกล่าว จะสมมติว่าบันทึกไว้เป็นไฟล์ชื่อ “readfile.js” เมื่อนำไปรันบน Node.js ด้วยการพิมพ์คำสั่งบนคอมมานไลน์เป็น “node readfile.js” ก็จะได้ผลลัพธ์ดังนี้

```
c:\test>node readfile.js
Last statement
Reading a file: object
```

สำหรับตัวอย่างดังกล่าว จะเป็นการใช้งานมอดูล ซึ่งภายในโค้ดจะมีรายละเอียดที่น่าสนใจดังนี้

- ประโยค let fs = require('fs'); จะโหลดมอดูลที่ชื่อ “fs” ด้วยการใช้ฟังก์ชั่น require ซึ่งจะรีเทิร์นอ็อบเจ็กต์ออกมา โดยมีตัวแปร fs มาอ้างอิงอ็อบเจ็กต์ดังกล่าวอีกที

- ประโยค fs.readFile('json.txt', ...) จะเป็นการเรียกฟังก์ชั่น readFile ให้อ่านไฟล์ที่ชื่อ "json.txt" (ติดต่อกับ I/O) ซึ่งการทำงานจะเป็นแบบอะซิงโครนัส จึงไม่ต้องรอให้ประโยคนี้ทำงานเสร็จก่อน แต่สามารถข้ามไปทำประโยค console.log('Last statement'); ที่อยู่บรรทัดสุดท้ายก่อนได้เลย
- เมื่ออ่านไฟล์ json.txt เสร็จเรียบร้อยแล้ว (เกิดเหตุการณ์อ่านไฟล์เสร็จ) ก็จะมาเรียกคอลแบ็ค ซึ่งเป็นค่าอากิเมนต์ตัวที่สองของ fs.readFile() ให้ทำงานภายหลังนั้นเอง

***__หมายเหตุ__ ปกติแล้วฟังก์ชั่นของอ็อบเจ็กต์ จะเรียกว่า “เมธอด (method)”

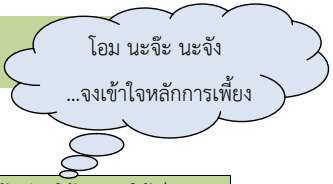
ถ้าคุณเห็นโค้ดของ Node.js ไปเรื่อย ๆ เวลาคุณโหลดมอดูลมาใช้งาน จะเห็นว่าไอ้พวกเมธอดที่ต้องติดต่อกับ I/O แบบอะซิงโครนัส เมธอดพวกนี้จะประกาศพารามิเตอร์ที่มีหน้าตาสะสวยเหมือน ๆ กัน ประมาณนี้ ...

```
apiObject.method(param1, param2, ..... [, callback_function]);
```

สำหรับเมธอดที่เราโหลดมาใช้งาน มันจะรับค่าอากิวเมนต์กี่ตัวก็ได้ ขึ้นอยู่กับว่าเมธอดประกาศไว้กี่ตัว แต่ค่าอากิวเมนต์ตัวสุดท้าย ส่วนใหญ่แล้วจะเป็นคอลแบ็ค ...จริง ๆ นะ ไม่ได้โม้ด้วย และตัวคอลแบ็คเราจะเขียนหรือไม่เขียนก็ได้เช่นกัน (เป็นออปชั่น)

สำหรับรายละเอียดการใช้งานมอดูล เดี๋ยวจะมาอธิบายต่ออีก แต่ตอนนี้ขอตัดจบก่อนครับ (ซะงั้น) 

คำแนะนำเรื่องโค้ด



เวลาคุณจะอ่านโค้ดใน Node.js ก็ขอให้เข้าใจหลักการ 3 ข้อที่ผมสรุปมา ได้แก่

1. โค้ดมักจะเริ่มต้นด้วยการใช้ require() เพื่อโหลดมอดูลที่เหมาะกับงาน แล้วเรียกใช้งานมันให้เป็น
2. เข้าใจการทำงานแบบอะซิงโครนัส โดยใช้กลไก Event-driven หรือ Event-loop
3. เข้าใจคอนเซปท์ฟังก์ชั่นคอลแบ็ค กับ closure (เพื่อเขียนโปรแกรมแบบ Functional programming)

การอ่านโค้ดบน Node.js จริง ๆ นะไม่ยากเลย ขอแค่รู้จักเปิดคู่มือศึกษา API แต่ละมอดูลที่เราโหลดเข้ามา (หลักการข้อ 1) พร้อมทั้งเข้าใจหลักการข้อ 2 กับ 3 ให้ดี ๆ ก็สามารถประยุกต์ใช้งาน Node.js แบบพลิกแพลงหลายร้อยกระบวนท่าได้แล้ว ...แต่ต้องรู้จาวาสคริปต์ด้วยนะ มันจะดีมาก ๆ

```
// โค้ดส่วนใหญ่ใน Node.js ก็จะประมาณนี้
var module = require('module_name');        // มักจะเริ่มต้นด้วยการโหลดมอดูลให้เหมาะกับงาน

/* สำหรับโค้ดที่ตามมา
...ขอให้เราเรียกใช้งานมอดูล ตามคู่มือให้เป็นก็พอแล้ว
...แต่คุณต้องเข้าใจการทำงานของโปรแกรมว่า ...เป็นแบบอะซิงโครนัส โดยใช้กลไก Event-driven
...ที่สำคัญคุณต้องเขียนจาวาสคริปต์แบบ Function programming ไม่ใช่แบบ OOP
*/
```

ยิ่งใครมาจากภาษาอื่น และเป็นแฟนพันธ์แท้เรื่องการเขียนโปรแกรมเชิงวัตถุ หรือเรียกสั้น ๆ ว่า OOP (Object oriented programming) อาจไม่ค่อยชอบโค้ดบน Node.js เลย มันจะเหมือนอยู่คนละกาแล็กซี่

~~Object oriented programming~~

VS

Functional programming

เพราะทุกอนุของโค้ดใน Node.js จะเต็มไปด้วยฟังก์ชั่น ที่เห็นมันอยู่ได้หมดทุกแห่งหน ทั้งเป็นค่าอากิวเมนต์หรือซ้อนอยู่ในฟังก์ชั่นตัวอื่น หรือถูกรีเทิร์นออกมาจากฟังก์ชั่นตัวอื่น จะมีเรื่อง closure อีก แถมแต่ละฟังก์ชั่นคอลแบ็คจะถูกเรียกให้ทำงานตอนไหนก็ยังไม่รู้ เพราะมันทำงานแบบอะซิงโครนัส ตามเหตุการณ์ที่เกิดขึ้น

....แน่นอนโค้ดใน Node.js อาจดูแล้วตาลาย สำหรับผู้ไม่คุ้นชินครั้งแรก ห้า ห้า ๆ

ด้วยเหตุนี้ ผมจึงแนะนำว่าให้คุณเลิกมองโค้ดแบบ OOP แต่ให้มองเป็นแบบ Functional programming หรือการเขียนโปรแกรมเชิงฟังก์ชั่นแท้ ๆ จะดีกว่า

สำหรับแนวคิด Functional programming ในจาวาสคริปต์เรื่องมันยาว เลยขอไม่อธิบายนะครับ ...เพราะมันเกินขอบเขตของหนังสือเล่มนี้พอควร

***__หมายเหตุ__ จาวาสคริปต์ยังไม่สนับสนุนแนวคิด functional programming แบบแท้ ๆ 100 % นะครับ

สไตล์การเขียนโค้ดใน Node.js

การเขียนโค้ดใน Node.js จะมีไสตล์ที่คุณควรรู้ไว้ ดังตัวอย่าง

```
// function callback (param){          // ประกาศฟังก์ชั่นแบบนี้ได้เหมือนกัน
var callback = function(param){
        console.log('to do something:', param);
}
function myFunction(param, func){   // ค่าอากิวเมนต์ตัวที่สอง จะรับค่าเป็นฟังก์ชั่น
        func(param);                        // ฟังก์ชั่นคอลแบ็ค
}
myFunction('say', callback);            // แสดงผลลัพธ์เป็น "to do something: say"
```

myFunction() ในตัวอย่าง เวลาเรียกใช้งานจะรับค่าอากิเมนต์ตัวที่สองเป็นตัวแปร callback ซึ่งมีค่าเป็นฟังก์ชั่น ...ซึ่งโค้ดดูปกติธรรมดาไม่มีอะไร

แต่โค้ดส่วนใหญ่ที่คุณเห็นใน Node.js เขาจะนิยมเรียก myFunction() พร้อมทั้งประกาศฟังก์ชั่นไร้ชื่อ (Anonymous functions) ในตำแหน่งที่เป็นค่าอากิวเมนต์ทีเดียวจบเลย (Inline) ...ถ้านึกภาพไม่ออก ก็ดูตัวอย่างหน้าถัดไปครับ

```
function myFunction(param, func){
        func(param);
}
myFunction('say', function(param){    // ประกาศฟังก์ชั่นไร้ชื่อ ในตำแหน่งที่เป็นค่าอากิวเมนต์
        console.log('to do something:', param);
});                                   // แสดงผลลัพธ์เป็น "to do something: say"
```

ในตัวอย่างจะเรียกฟังก์ชั่น myFunction() ให้ทำงาน พร้อมทั้งประกาศฟังก์ชั่นไร้ชื่อ ในตำแหน่งที่เป็นค่าอากิวเมนต์ตัวที่สอง ...ซึ่งการเขียนแบบนี้คุณจะเห็นเยอะมาก จนแทบจะตาแฉะ

อีกอย่างหนึ่งที่คุณควรรู้ไว้ ...Node.js จะเขียนในสไตล์ที่เรียกว่า Continuation-passing style (CPS) มันคือวิธีเขียนให้คอลแบ็คทำงานอย่างต่อเนื่อง ...และผมจะขออธิบายด้วยการยกตัวอย่างโค้ด (ทำงานจริงไม่ได้นะ)

```
http.request( options, function(response) {
        response("data", function(data) {
                console.log("display", options);   // บรรทัด 3
                console.log("some data from the response", data);
        }
    );
}
);
```

1

2

ปกติแล้วการเรียกฟังก์ชั่นปกติธรรมดา เมื่อทำงานเสร็จจะรีเทิร์นค่าออกมาด้วยประโยค return แต่วิธี CPS เมื่อฟังก์ชั่นทำงานเสร็จ มันจะเรียกคอลแบ็คให้ทำงานเป็นลำดับสุดท้ายแทน

ในตัวอย่าง เมื่อฟังก์ชั่น request(options,....) ทำงานเสร็จ คอลแบ็คหมายเลข 1 จะถูกเรียกให้ทำงาน แล้วไปเรียกฟังก์ชั่น response("data",...) เมื่อมันทำงานเสร็จ ก็จะเรียกคอลแบ็คหมายเลข 2 ให้ทำงาน

คงพอนึกภาพออกนะครับว่า คอลแบ็คใน Node.js มันจะเขียนซ้อน ๆ กันแบบนี้ และเวลาทำงานก็จะถูกเรียกต่อเนื่องไปเรื่อย ๆ ...เท่านั้นยังไม่พอ ในบรรทัดที่ 3 ก็ยังเห็นค่า options ด้านนอกอีกด้วย

เมื่อถึงตรงนี้คิดว่าคนที่ชอบ OOP บางคน อาจไม่ชอบวิธีเขียนแบบนี้ เพราะมันดูชุลมุนวุ่นวายพิลึกดี จัดไม่เป็นระเบียบเหมือน OOP ...แต่ก็ทำไงได้เนอะ ต้องทำใจ เมื่อคุณคิดนอกใจจากภาษาอื่นมาใช้ Node.js

อีกทั้งถ้าเขียนโค้ดใน Node.js ไม่ดี ก็อาจเกิด**คอลแบ็คนรก (Callback Hell)** เนื่องจากมีการเรียกใช้คอลแบ็คต่อเนื่องกันเป็นลูกโซ่ จนโค้ดอ่านย้ากยาก ดังตัวอย่าง

```
var express = require('express');        // มอดูล express จะได้เห็นในเล่ม 2
var app = express();
var fs = require('fs');

app.get('/', function(req, res) {  1
        res.send('<h1>Hello world </h1>');

        fs.writeFile('msg.txt', 'callback hell', function (err) {  2
                if (err) {  console.log(err);  }

                fs.readFile('msg.txt',  function(err, buffer) {  3
                        if (err) {  console.log(err);  }
                        console.log(buffer.toString());  // เข้าถึงข้อมูลในไฟล์ json.txt
                }); // สิ้นสุด fs.readFile(...)

        });       // สิ้นสุด fs.writeFile(...)

});       // สิ้นสุด app.get(...)

app.listen(8080, function() {
      console.log('Server running at http://localhost:8080/');
});
console.log("Start server");
```

ในตัวอย่างจะมีเพียงคอลแบ็ค 1, 2, 3 ซ้อนกันเท่านั้น แต่ถ้ามีตัวที่ 4, 5, 6 ....N ซ้อนกันไปเรื่อย ๆ มันก็จะกลายเป็น**คอลแบ็คนรก แบบตกขุมอเวจี** เพราะโค้ดจะอ่านยากมาก จนตาลาย ...ซึ่งเราต้องระมัดระวังให้ดี (อาจใช้ Promise ซึ่งมีอยู่ใน ES6 มาช่วยเขียนโค้ดให้ดีขึ้นก็ได้)

อะซิงโครนัส กับ ซิงโครนัส ที่อยู่ในโค้ด

ผมจะขอยกตัวอย่างโค้ด Node.js ต่อไปนี้ เพื่อให้คุณระมัดระวังตัว มิฉะนั้นจะเหมือนความผิดบาป

```
var fs = require('fs');
console.log('Before while loop');        // แสดงออกมา แค่ข้อความเดียว
while(true);
fs.readFile(
        'test.txt',                      // ชื่อไฟล์
        function(err, buffer) { // คอลแบ็คไม่เคยถูกเรียก (เส้นทางการทำงานของโปรแกรม มาไม่ถึง)
                console.log('After while loop');
        }
);
```

เส้นทางการทำงานของโปรแกรม จะติดอยู่ที่ลูปนี้ชั่วกัปชั่วกัลป์

จากตัวอย่างนี้ จะมีรายละเอียดที่น่าสนใจดังนี้

- เมื่อนำไปรันบน Node.js จะมีแค่ข้อความเดียวถูกแสดงออกมา ได้แก่ 'Before while loop'

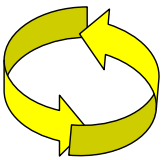
```
c:\test>node readfile.js
Before while loop
```

- ส่วนข้อความ 'After while loop' จะไม่ถูกแสดงออกมา

ซึ่งคุณต้องเข้าใจนะครับว่า Node.js มันมีแค่เทรดเดียว และตรงประโยค while(true); ก็คือการเขียนจาวาสคริปต์ธรรมดา ไม่ได้ทำงานแบบอะซิงโครนัสแต่อย่างใดเลย ...อ้าว (ร้อนเหรอ)

...คือใน Node.js นะ โค้ดบางส่วนจะทำงานแบบซิงโครนัส จนเสร็จเรียบร้อยไปก่อน

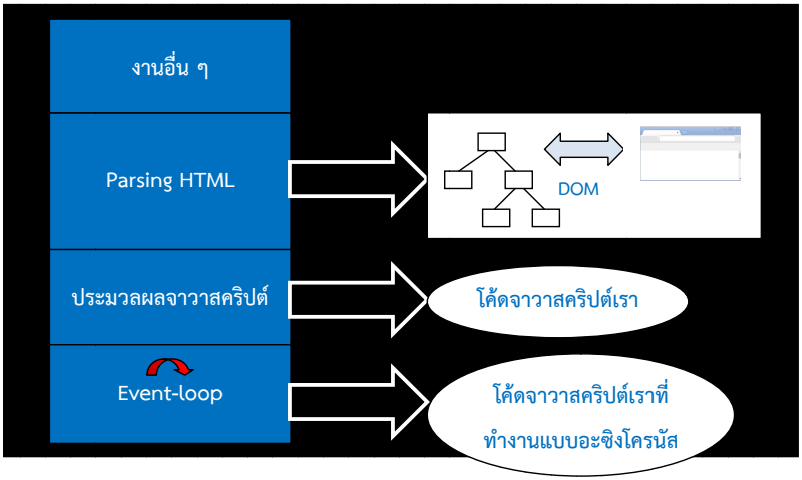
...ส่วนโค้ดที่ทำงานแบบอะซิงโครนัสของตัวอย่างนี้ ก็คือคอลแบ็คของ fs.readFile() เท่านั้นเองแหละ



ด้วยเหตุนี้ ประโยคคำสั่ง while(true); จึงวนติดลูปตลอดกาลชั่วฟ้าดินสลาย ตราบใดที่เราไม่ยอมปิดคอมมานไลน์ หรือกด Ctrl +C เพื่อยกเลิก

เบื้องหลังของ Node.js

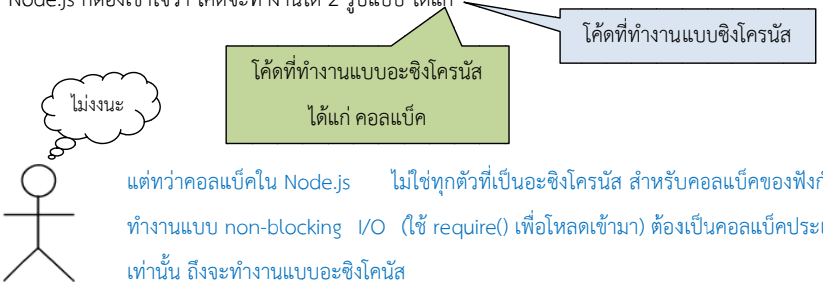
ก่อนจะใช้งาน Node.js ควรเข้าใจเบื้องหลังการทำงานของเว็บบราวเซอร์ส่วนใหญ่ ซึ่งมันจะมีแค่เทรดเดียว เมื่อมันเริ่มทำงานด้วยการอ่านไฟล์ HTML ก็จะมีรายละเอียดตามภาพ



ตามภาพที่ผมยกมาให้ดู เว็บบราวเซอร์ส่วนใหญ่ก็จะทำงานประมาณนี้แหละ ซึ่งผมจะอธิบายการทำงานที่สำคัญดังนี้

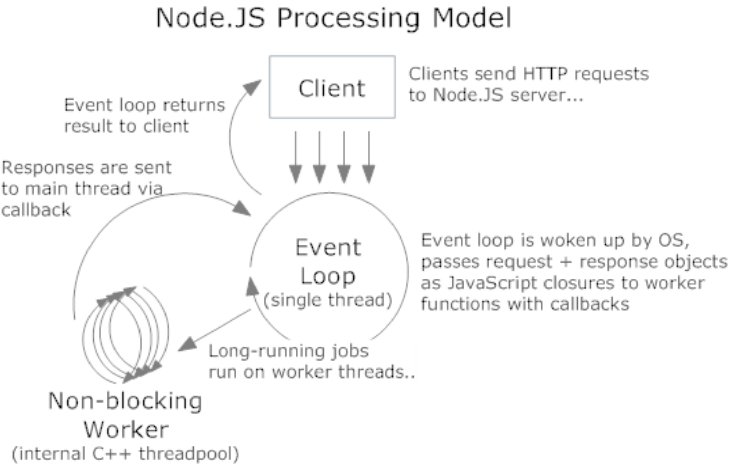
1) **Parsing HTML** คือการเอาไฟล์ HTML มาแปลงให้กลายเป็นโครงสร้างข้อมูลแบบลำดับชั้นที่เรียกว่า DOM (Document Object Model) พร้อมทั้งแสดงผลออกมาทางหน้าจอเว็บบราวเซอร์
2) หลังจากนั้น**จาวาสคริปต์เอนจิน**ในเว็บบราวเซอร์ จะมาประมวลผลโค้ดจาวาสคริปต์เรา (แท็ก <script>....</script>) ให้เสร็จครั้งเดียวไปเลย ซึ่งการทำงานจะเป็นแบบซิงโครนัส
3) แต่ทว่าจาวาสคริปต์เอนจินยังทำงานไม่เสร็จดีนะครับ เพราะยังมี Event-loop ที่รอรับเหตุการณ์ต่าง ๆ ในหน้าเว็บ เช่น คลิกเมาส์ พิมพ์ดีดบนหน้าเว็บ เป็นต้น ซึ่งจาวาสคริปต์เอนจินจะมาเรียกคอลแบ็คที่ตรงกับเหตุการณ์นั้น ๆ ให้ทำงาน ...แน่นอนละครับ คอลแบ็คดังกล่าวจะทำงานแบบอะซิงโครนัส

เมื่อ Node.js มันถอดแบบมาจาก V8 จึงมีการทำงานแค่ข้อ 2 กับ 3 เท่านั้น ด้วยเหตุนี้เวลาจะเอ่เห็นโค้ดใน Node.js ก็ต้องเข้าใจว่า โค้ดจะทำงานได้ 2 รูปแบบ ได้แก่



แต่ทว่าคอลแบ็คใน Node.js ไม่ใช่ทุกตัวที่เป็นอะซิงโครนัส สำหรับคอลแบ็คของฟังก์ชั่นที่ทำงานแบบ non-blocking I/O (ใช้ require() เพื่อโหลดเข้ามา) ต้องเป็นคอลแบ็คประเภทนี้เท่านั้น ถึงจะทำงานแบบอะซิงโคนัส

เพื่อตอกย้ำความเข้าใจให้หนักหัวเล่น ๆ ก็จะแสดงภาพดังต่อไปนี้

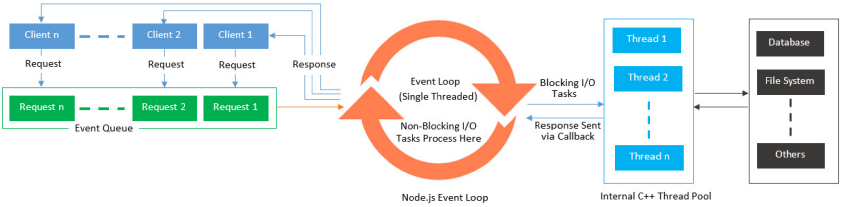


ภาพนี้นำมาจากเว็บ http://kikobeats.com/synchronously-asynchronous/ ซึ่งผมจะขออธิบายสั้น ๆ เกี่ยวกับกลไก Event-loop ใน Node.js ดังต่อไปนี้ แล้วกันเนอะ

เบื้องหลังการทำงานของ Node.js จริง ๆ แล้ว มันจะใช้ V8 เพื่อแปลงโค้ดจาวาสคริปต์ ให้กลายมาเป็นโค้ดในภาษา C++ แล้วจึงแตกเทรดย่อย ๆ ออกมา (ในรูปคือ Non-blocking Worker) เพื่อจัดการเรื่อง non-blocking I/O โดยใช้ไลบรารี่ libuv (มีไว้เพื่อจัดการเรื่องอะซิงโครนัส I/O โดยเฉพาะ) ส่วนเทรดที่กล่าวมาเมื่อทำงานเสร็จก็จะส่งเรสปอนส์ (Response) ไปเรียกคอลแบ็คในโค้ดเรา ที่ตรงกับเหตุการณ์นั้น ๆ ให้ทำงาน

***ถ้าท่านสนใจเกี่ยวกับ libuv ก็สามารถอ่านได้ที่ https://github.com/libuv/libuv

ลองดูอีกภาพที่อธิบายการทำงานของ Node.js ดังนี้ครับ



ถ้าสนใจรายละเอียดเรื่องรูปนี้ ก็ลองดูคำอธิบายต่อที่ลิงค์ด้านล่างแล้วกันนะครับ (ผมเอารูปมาจากลิงค์นี้แหละ) ...และขออนุญาตตัดจบแค่นี้แล้วกันนะ แฮ ๆ ๆ

- http://www.dotnet-tricks.com/Tutorial/nodejs/QFI4 3 0 1 2 1 5 -Exploring-Node.js-Code-Execution-Process.html

เอาเป็นว่าถ้าคุณอยากปวดหัวยุ่งยากยกกำลัง[2] ด้วยการแตกเทรดเล่นเอง เพื่อจัดการด้าน I/O หนังสือเล่มนี้จะไม่ตอบโจทย์คุณเลย แต่ถ้าไม่อยากคิดเรื่องเทรดให้มันรกสมอง อยากมีเวลาไปโฟกัสโค้ดที่ต้องการทำงานจริง ๆ เท่านั้น (Business logic) การใช้งาน Node.js คือ 1 ในคำตอบดีที่สุดในชั่วโมงนี้ครับ (ช่วงที่ผมแต่งหนังสือนะ)

แต่แอบกระซิบนิดหนึ่ง แม้ว่า Node.js มันถูกออกแบบให้มีเทรดเดียว แต่คุณก็สามารถใช้มอดูล “cluster” เพื่อแตกโพรเซสการทำงานย่อย (แบบเทรด) ได้เช่นกัน (ทำง่ายด้วย)

มอดูล

ปกติแล้วถ้าเราจะนำเข้า (Import) ไฟล์ *.js ตัวอื่น ๆ ในหน้าเว็บ HTML เราก็จะใช้โค้ดดังตัวอย่างนี้

```
<script src="lib.js"></script>
```

ในจาวาสคริปต์ฝั่ง Node.js สามารถนำเข้าไฟล์ *.js ตัวอื่นเข้ามาได้ (แต่ผมชอบเรียกว่า "โหลด") ด้วยประโยคดังนี้

```
var module = require('module_name');
```

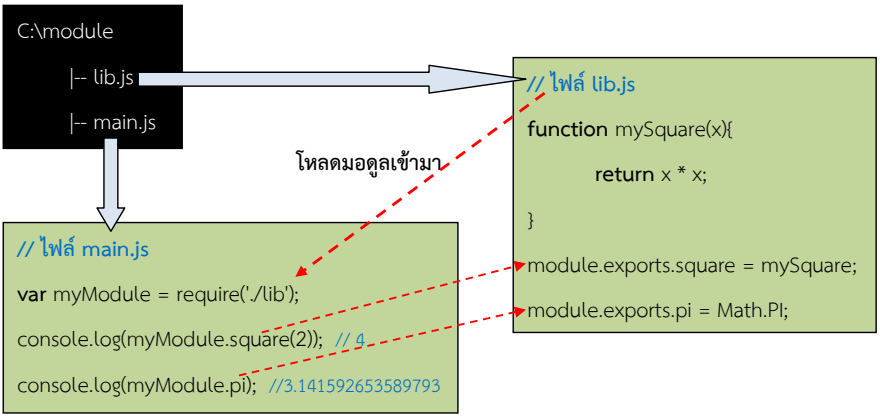
...โค้ดนี้คุณเคยเห็นแล้วแหละ เวลาโหลดมอดูลในบทก่อน ๆ แต่ผมจะอธิบายให้ลึกลงไปอีกในบทนี้ จากที่เคยค้างคาเอาไว้ 

ในบทก่อน ๆ คุณคงเคยเห็นมอดูลชื่อ "fs" ซึ่งจะเป็นมอดูลหลักของ Node.js ที่มีมาให้อยู่แล้ว แต่ถ้าเป็นมอดูลเสริม (third-party) คุณต้องโหลดมาติดตั้ง ด้วยคำสั่ง npm บนคอมมานไลน์ดังนี้

```
npm install <ชื่อมอดูล>
```

วิธีสร้างมอดูลใช้งานเอง

วิธีสร้างมอดูลขึ้นมาใช้เอง จะอธิบายง่าย ๆ โดยให้คุณดูโครงสร้างโปรเจค และโค้ดข้างล่างต่อไปนี้



ถ้าคุณลองรันด้วยคำสั่ง "node main.js" ก็จะได้ผลลัพธ์ดังนี้

```
C:\module>node main.js
4
3.141592653589793
```

จากตัวอย่างที่ผมยกมาจะเป็นการสร้างมอดูล “lib” (ไฟล์ชื่อ lib.js) ซึ่งคุณน่าจะเห็นคอนเซปท์การสร้างมอดูลว่าไม่ยากเลย ซึ่งผมจะขอสรุปวิธีสร้างดังนี้

- มอดูลจะต้องเขียนเป็นไฟล์จาวาสคริปต์ ที่มีนามสกุล *.js
- สำหรับสิ่งที่ส่งออกไป หรือ export (ให้ไฟล์อื่นมาโหลดไปใช้งาน) มันจะเป็นอะไรก็ได้ เช่น ฟังก์ชั่น ตัวแปร คลาส และอื่น ๆ โดยจะมีรูปแบบประโยคเวลาจะส่งออกไปดังนี้

```
module.exports.xxx  = yyy;
// โดยที่ yyy คือสิ่งที่จะส่งออกไป
// แต่เมื่อไฟล์จาวาสคริปต์ตัวอื่น มาโหลด yyy ไปใช้งาน ก็จะให้เข้าถึงโดยใช้ชื่อ xxx แทน
// หรือจะเขียนเป็น
module.exports = object; // เมื่อ object คืออ็อบเจ็กต์ในจาวาสคริปต์
```

วิธีโหลดมอดูล

ถ้าเป็นมอดูลหลัก (Core modules) ที่มีมาให้แล้วใน Node.js เราก็โหลดมาใช้งานได้เลย โดยไม่ต้องระบุชื่อพาธ (path) เช่น

```
var http = require('http');        // http คือมอดูลหลัก
```

แต่ถ้ามอดูลที่โหลดเข้ามานั้น เราเขียนขึ้นมาใช้เอง ก็ต้องโหลดมาเป็นไฟล์ .js โดยการระบุชื่อพาธเต็ม ๆ ไปเลย ดังนี้

```
var myModule = require('/module/lib');
// let myModule = require('/module/lib.js');              // จะมี .js ต่อท้ายชื่อไฟล์ก็ได้
```

ประโยคคำสั่งนี้จะโหลดมอดูล “lib” ที่ผมเขียนขึ้นเองกับมือ (ในหัวข้อก่อน) โดยมีรายละเอียดที่น่าสนใจดังนี้

- เนื่องจากผมใช้งานบนวินโดวส์ และตัว Node.js ก็ติดตั้งอยู่ที่ตำแหน่งไดร์ฟ C: ...ด้วยเหตุนี้ชื่อพาธ '/module/lib' ก็คือ 'c:\module\lib.js' นั้นเองละครับ
- เราสามารถระบุชื่อมอดูลเป็นชื่อไฟล์ โดยจะมี หรือไม่มีนามสกุล .js ต่อท้ายก็ได้ ...เพราะตอนแรก Node.js จะมองเป็นชื่อโฟลเดอร์ก่อน แล้วถ้าค้นหาไม่เจอ ก็จะเติม .js ต่อท้ายชื่อโฟลเดอร์เข้าไป (เดี๋ยวต่อไปคุณจะเห็นเองว่า เราสามารถโหลดมอดูลเป็นโฟลเดอร์ได้ด้วย)

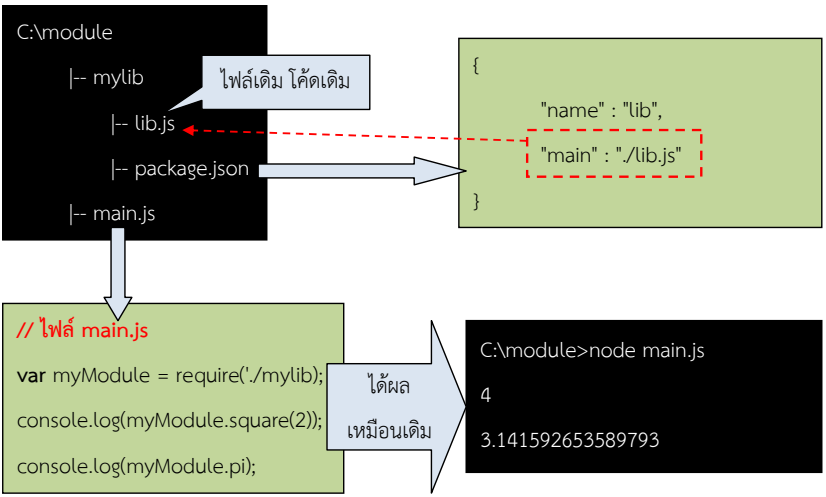
แต่ถ้าเราอ้างพาธแบบ Relative ก็ให้ใช้ “./” นำหน้า เพื่อบอกตำแหน่งไดเรคเทอรี่ปัจจุบันที่ไฟล์กำลังทำงานอยู่ ซึ่งในตัวอย่างเดิม ไฟล์ที่โหลดมอดูล lib.js มันทำงานอยู่ที่ c:\module จึงอาจเขียนใหม่ได้เป็น

```
var myModule1 = require('./lib');
var myModule2 = require('./lib');
var myModule3 = require('../module/lib');    // บรรทัดที่ 3
```

ในตัวอย่างนี้ คุณจะเห็นว่าประโยคคำสั่งบรรทัดที่ 3 เราสามารถใช้ “../” เพื่อขยับไดเรคเทอรี่ขึ้นมาอีกหนึ่งระดับ (มันเป็นการอ้างพาธที่เราคุ้นเคยกันดีนี้แหละเนอะ)

โหลดมอดูลจากโฟลเดอร์

เราสามารถโหลดมอดูลด้วยการระบุเป็นชื่อโฟลเดอร์ก็ได้ โดยก่อนอื่นจะขอจัดโครงสร้างโปรเจคใหม่ ดังนี้



ในตัวอย่างที่ยกมาในหน้าก่อนนี้ คุณจะเห็นประโยคคำสั่ง ⇨ `var myModule = require('./mylib');`

มันคือการอ้างถึงชื่อโฟลเดอร์เป็น “mylib” โดยทั้งนี้ Node.js จะเข้าไปอ่านไฟล์ "package.json" (ในโฟลเดอร์ mylib) ซึ่งข้างในไฟล์จะระบุพร็อพเพอร์ตี้เป็น "main" : "./lib.js" เพื่อบอกชื่อมอดูลที่จะถูกโหลดเข้ามาว่า ...มันมีชื่อเป็น lib.js (อยู่ในไดเรคเทอรี่ปัจจุบัน mylib)

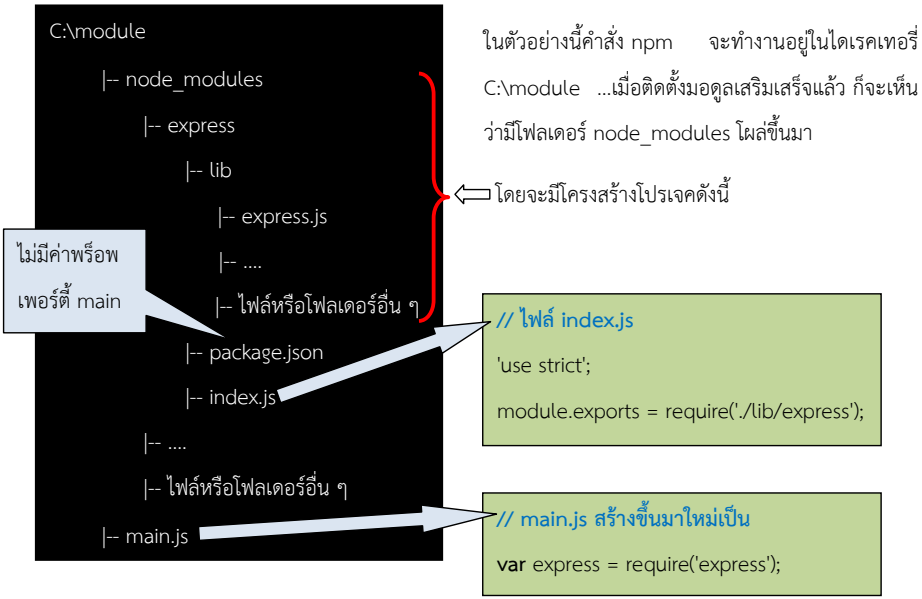
แต่ถ้าไฟล์ package.json ไม่มีชื่อพร็อพเพอร์ตี้ "main" ละก็ ...โดยดีฟอลต์แล้ว ตัวฟังก์ชั่น require() จะโหลดมอดูลที่มีชื่อไฟล์เป็น “index.js” เข้ามาแทน

สรุป ฟังก์ชั่น require() เมื่อเจอโฟลเดอร์ ก็จะเข้าไปอ่านไฟล์ package.json เพื่อหาที่อยู่ของไฟล์มอดูลที่จะโหลดเข้ามา ถ้าไม่มีระบุไว้ในค่าพร็อพเพอร์ตี้ main ก็จะใช้ไฟล์ชื่อ index.js แทนครับผม

โหลดมอดูลจากโฟลเดอร์ node_modules

ก่อนจะอธิบายการโหลดมอดูลด้วยวิธีนี้ ผมจะโหลดมอดูลเสริมเข้ามา ด้วยคำสั่งบนคอมมานไลน์ดังนี้

```
npm install express --save
```



ในตัวอย่างนี้ไฟล์ main.js จะโหลดมอดูลที่อยู่ในโฟลเดอร์ node_modules ด้วยการเขียนโค้ดเป็น

```
var express = require('express');
```

เราต้องเข้าใจอย่างนี้นะครับ ถ้าไม่ได้โหลดมอดูลหลัก และไม่ได้ระบุชื่อพาธ มันจะวิ่งไปหามอดูลที่พาธคือ "./node_modules/" และในตัวอย่างนี้จะหามอดูล "express" ที่พาธ "C:\module\node_modules\"

ซึ่งมันจะหาโฟลเดอร์ express เจอ แต่ทว่าไฟล์ package.json ที่อยู่ในโฟลเดอร์ ไม่ได้ระบุค่าพร็อพเพอร์ตี้ main ด้วยเหตุนี้ฟังก์ชั่น require() จึงโหลด index.js เข้ามาแทน ซึ่งข้างในไฟล์จะบอกให้ไปโหลดไฟล์มอดูล './lib/express.js' เข้ามา เพราะมันใช้ประโยคคำสั่ง

```
module.exports = require('./lib/express');
```

...แต่ยังไม่จบ ถ้าสมมติน่ะครับ ถ้าไม่มีโฟลเดอร์ node_modules ในไดเรคเทอรี่ปัจจุบัน ฟังก์ชั่น require() ก็จะวิ่งไปหาที่โฟลเดอร์แม่ ได้แก่ "../node_modules/" และถ้าไม่เจออีก ก็จะไปค้นหายังไดเรคเทอรีซึ่งได้ติดตั้ง npm

ในวินโดวส์ก็จะเป็น "C:\Users\username\AppData\Roaming\npm"
(เมื่อ username คือชื่อโฟลเดอร์ของผู้ใช้งานบนเครื่อง)

แต่ถ้าเป็น MacOS หรือ Linux ตัว npm ก็จะติดตั้งอยู่ที่ "/usr/local/share/npm"

เกร็ดความรู้ เมื่อโหลดมอดูลมาครั้งแรกด้วยฟังก์ชั่น require() มันจะเก็บไว้ในหน่วยความจำ (Cached) และเมื่อโหลดมอดูลชื่อเดิมครั้งต่อ ๆ มา มันก็จะใช้ตัวเดิมที่อยู่ในหน่วยความจำ เช่น

```
let http1 = require('http');  // บรรทัด 1
let http2  = require('http');  // บรรทัด 2
let http3  = require('http');  // บรรทัด 3
```

ในตัวอย่างนี้แม้ว่าจะโหลดมอดูล http สามครั้งก็จริง แต่มันจะใช้งานมอดูลในหน่วยความจำที่เดียวกัน

ไกด์ไลน์เล่ม 2

“สำหรับเล่ม 1 ผมจบแค่นี้นะครับ”

คอนเซปท์เล่ม 2 ที่จะเขียนต่อไป มันจะเป็นแนวตะลอนทัวร์พาใช้งาน Node.js เน้นพาทำเวิร์คช็อปง่าย ๆ เพื่อให้มองเห็นภาพรวมในการใช้งานมากกว่า

และถ้าพูดกันตามตรง ผมก็ไม่แนะนำให้คุณใช้งาน Node.js ดิบ ๆ เถื่อน ๆ หรอก เพราะเวลาคุณทำงานจริง มันมีเครื่องมือช่วยเขียน ไม่ว่าจะเป็น เฟรมเวิร์ค แพลทฟอร์ม และอื่น ๆ ที่จะทำให้ชีวิตเขียนโปรแกรมคุณดีกว่านี้ครับ

สำหรับเนื้อหาเล่ม 2 ก็จะประมาณนี้

- การอ่านและเขียนไฟล์
- สร้างเว็บแอพพลิเคชั่นง่าย ๆ
- สร้างเว็บแอพพลิเคชั่นอย่างเร่งด่วน
- เทมเพลตหน้าเว็บ
- ติดต่อฐานข้อมูล MySQL
- ติดต่อฐานข้อมูล MongoDB
- สร้างเว็บแอพพลิเคชั่นแบบเรียลไทม์
- EMITTER PATTERN
- และอื่น ๆ ที่นึกไม่ออกในตอนนี้

***** อนาคตเนื้อหาอาจปรับปรุงเปลี่ยนแปลงได้ ตามเทคโนโลยีที่หมุนเร็วปานติดจรวด**

อ้างอิง

หนังสือ

[1] Pedro Teixeira, **“Professional Node.js Building Javascript Based Scalable Software”**, John Wiley & Sons, Inc., 2013.

เอกสารจากเว็บไซต์ เข้าถึงล่าสุด 30 ธ.ค. 2558

[1] https://en.wikipedia.org/wiki/Node.js

[2] https://nodejs.org/en/

[3] http://expressjs.com/

[4] http://getbootstrap.com/

[5] http://ejs.co/

[6] http://expressjs.com/en/starter/generator.html

[7] http://kikobeats.com/synchronously-asynchronous/