



# NLP與網路應用

## 聊JS (TFjs)

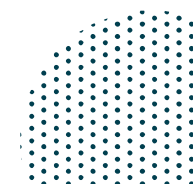
2022/12/1

Sean Tseng





# 今天不會講太難的東西(應該...)

- 今天不會講到React, 那是下禮拜的事。
  - 但今天會講如何使用Tensorflow.js, 用官方提供模型
  - 在那之前先講一些故事: 前端本來就很瘋狂
  - 這一切其來有自, Javascript本來就是個很妙的語言
- 
- 
- 

# QA的例子

- 這就是tensorflow.js包好的 MobileBERT，他可以拿來做QA。
- 雖然只有英文版，不過已經滿好玩的了。
- 今天不會細講這個例子，但等一下toxicity detection的例子和這個非常像。

## QnA with MobileBERT (TFjs)

The model can be used to build a system that can answer users' questions in natural language. It was created using a pre-trained BERT model fine-tuned on SQuAD 2.0 dataset.

How is the model created?

Answer

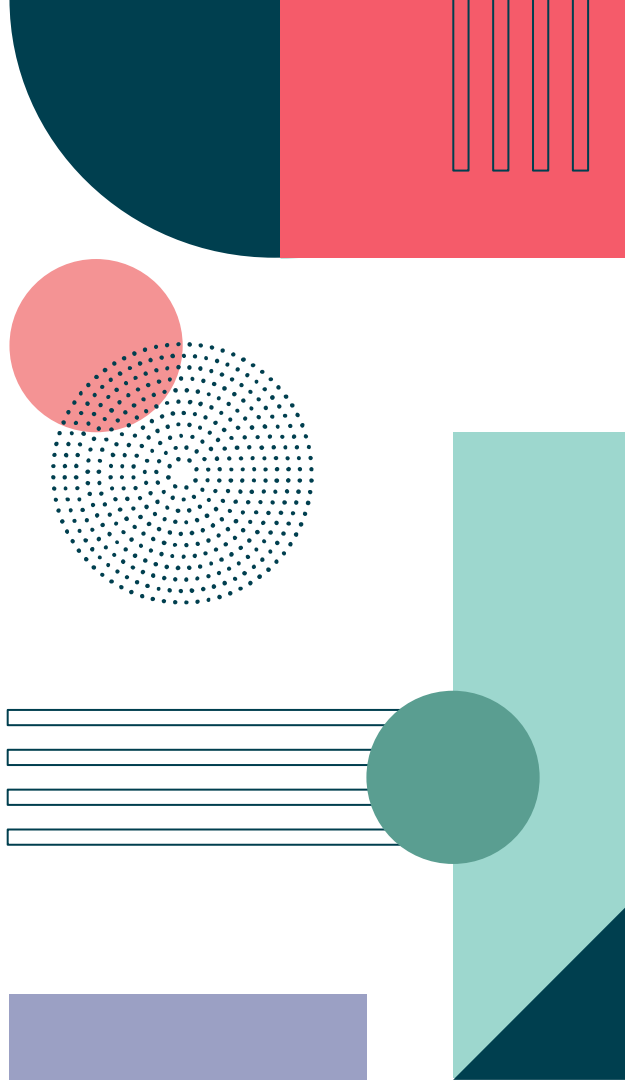
using a pre-trained BERT model fine-tuned on SQuAD 2.0 dataset

提醒: MobileBERT雖比BERT小很多，但下載仍需要100M，需記憶體500M左右。載入需要一些時間。

1

# Frontend

THE CRAZY WORLD



# 前端世界超瘋狂

1. 從來沒有人規定我們一定要完全精熟一項工具，才有資格使用或談論該工具。
  - a. 的確有的時候社會壓力很大，但那不是來自於程式本身
2. 前端的世界變化非常快，但幸好逐漸穩定下來。可是從瘋狂的2016-2018年到現在，也夠令人崩潰了。
3. 而且...2018年，React才剛開始，還沒有hook呢！
4. 這個現象已經有名詞了：frontend fatigue

# 我自己的前端經驗

## 1. 前端真的是變很快的東西：

- a. 第一個網頁(烘焙機時代)：FrontPage (~2000)
- b. 第一個動態網頁：HTML/PHP (~2005)
- c. 第一個Javascript專案：jQuery/d3.js (~2012)
- d. 第二個前端專案：Angular (~2018)
- e. 第n個前端專案：Vue、React、Next (~2020/2022)

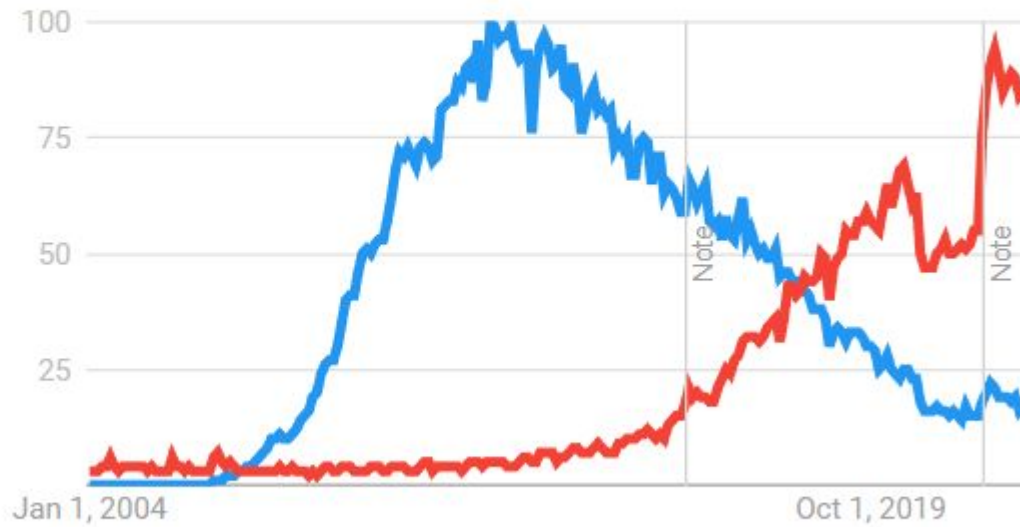
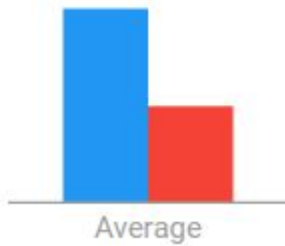
## 2. 相較之下，其他程式領域沒變那麼多：

- a. Python從2010年到現在，除了tensorflow、pyTorch出來攪局以外，現在大同小異。

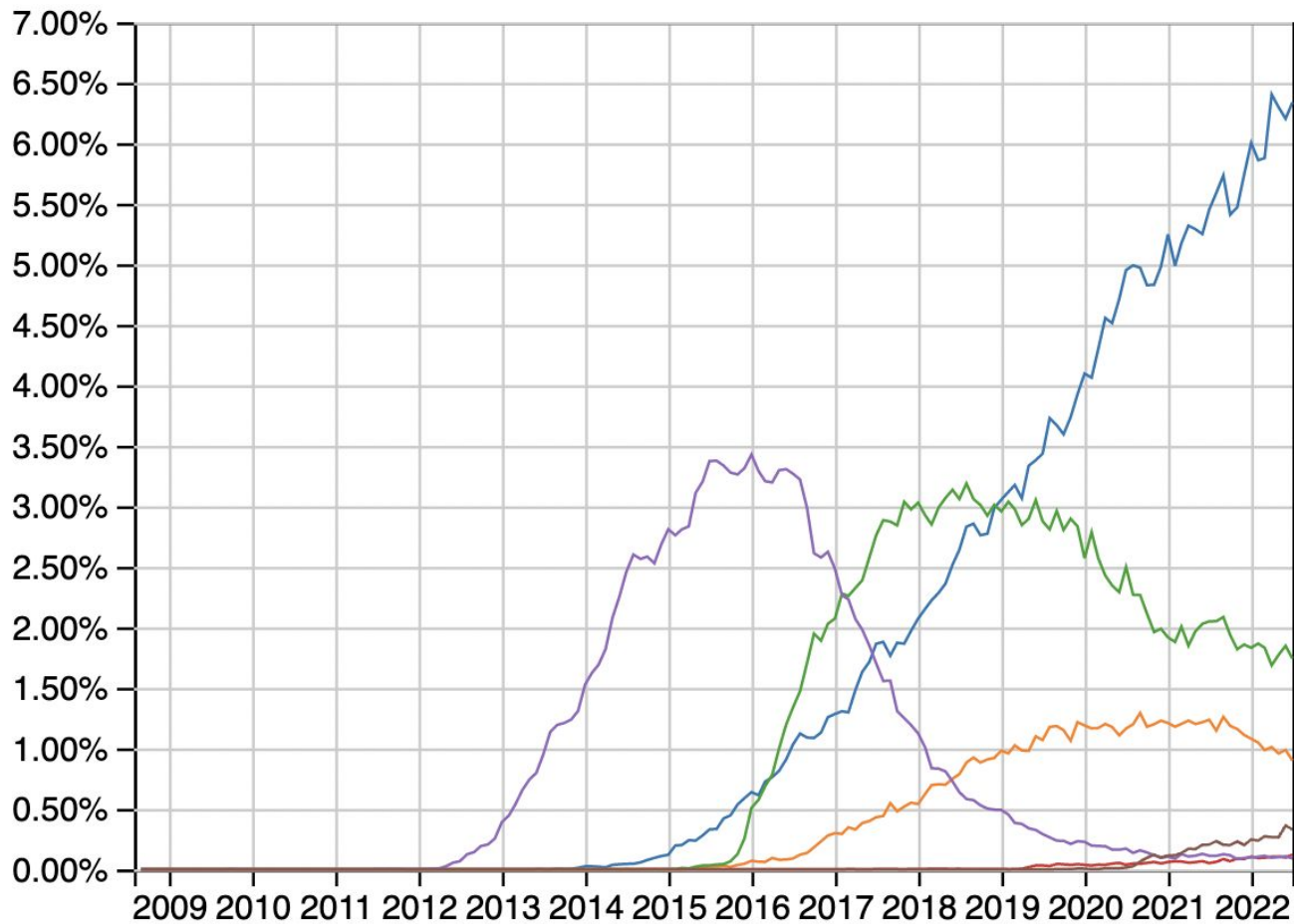
## Interest over time



jQuery / React



% of Stack Overflow questions that month



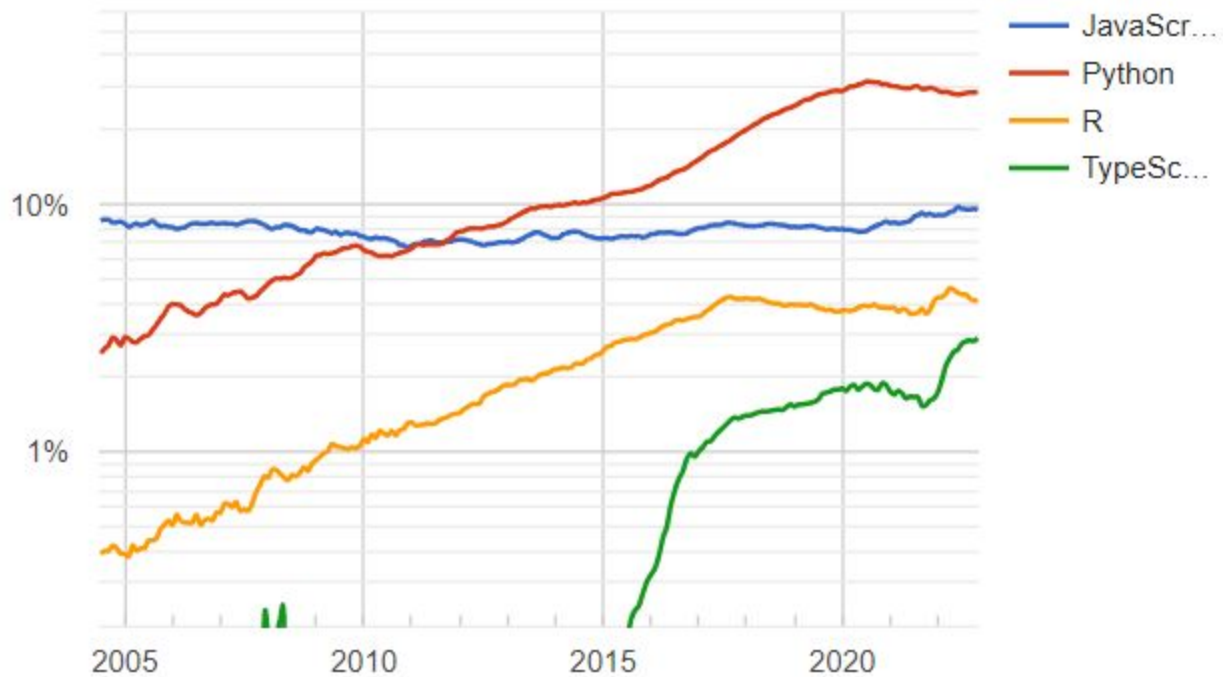
Tag

- reactjs
- angularjs
- angular
- vue.js
- vuejs3
- svelte

Year



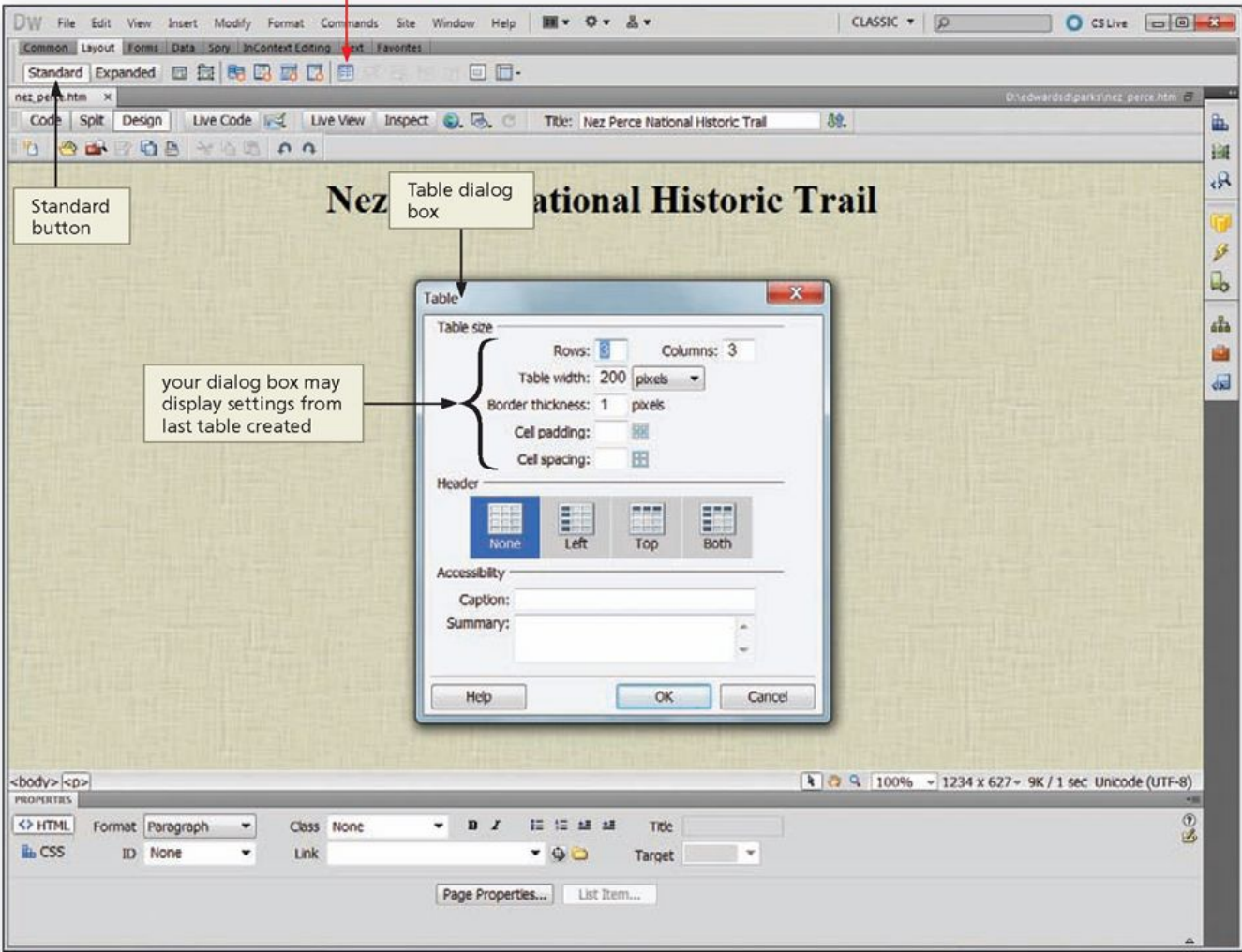
## PYPL PopularitY of Programming Language



# 以前的網頁設計



Table button



Standard button

Nez Perce National Historic Trail

Table dialog box

your dialog box may display settings from last table created

Table

Table size

Rows: 3 Columns: 3

Table width: 200 pixels

Border thickness: 1 pixels

Cell padding:

Cell spacing:

Header

None Left Top Both

Accessibility

Caption:

Summary:

Help OK Cancel

<body><p>

PROPERTIES

HTML Format Paragraph Class None ID None Link Title

CSS ID None Link Target

Page Properties... List Item...

# 有些東西本來就難

1. 大家曾試著把網頁設計變簡單，所以有FrontPage / DreamWeaver / Flash，這些「所視即所得」(WYSIWYG)的工具。
2. 但後來，網站愈做愈複雜，才發現視覺化工具沒辦法表達那麼複雜的關係。現在的前端，視覺設計和程式設計完全是兩組人兩組工具。
3. 所以，千萬，千萬，千萬別覺得自己為什麼學不會。因為：它·們·超·難

# 前端哪裡難

1. 如果前端是當年1989年Tim Berners-Lee想的這樣，那當然沒什麼好難的。
2. 問題不在視覺元素，而是它是唯讀的。而且只有一個輸出，而且頁面本身「沒有狀態」(stateless)。

## World Wide Web

The WorldWideWeb (W3) is a wide-area **hypermedia** information retrieval initiative aiming to give universal access to a large universe of documents.

Everything there is online about W3 is linked directly or indirectly to this document, including an **executive summary** of the project, **Mailing lists**, **Policy**, November's **W3 news**, **Frequently Asked Questions**.

### What's out there?

Pointers to the world's online information, **subjects**, **W3 servers**, etc.

### Help

on the browser you are using

### Software Products

A list of W3 project components and their current state. (e.g. **Line Mode**, **X11 Viola**, **NeXTStep**, **Servers**, **Tools**, **Mail robot**, **Library**)

### Technical

Details of protocols, formats, program internals etc

### Bibliography

Paper documentation on W3 and references.

### People

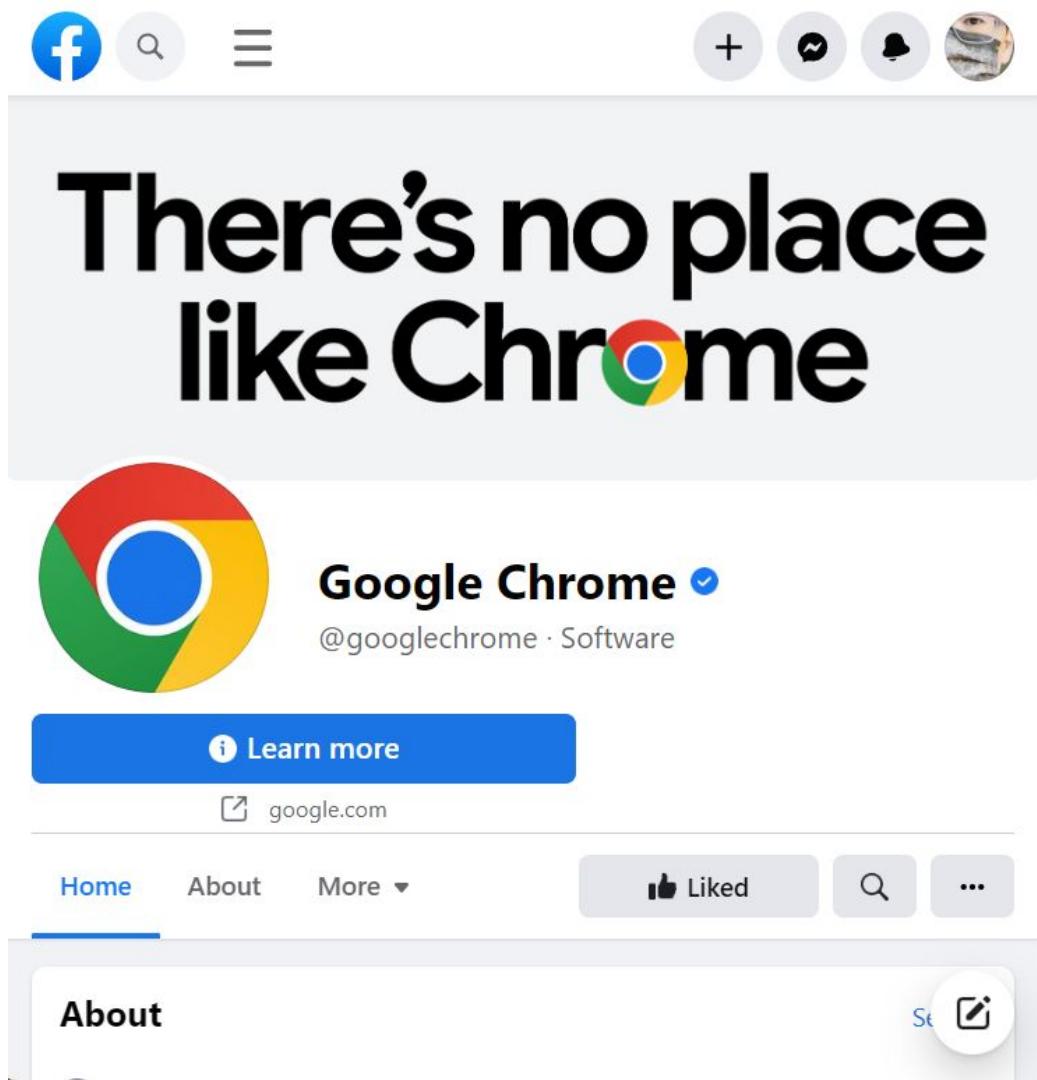
A list of some people involved in the project.

-- press space for next page --

Arrow keys: Up and Down to move. Right to follow a link; Left to go back.  
H)elp O)ptions P)rint G)o M)ain screen Q)uit /=search [delete]=history list

## 但當胃口養大之後

1. FB有很多輸入: Like、搜尋、回覆、分享、回應
2. 也有很多輸出: 訊息提示、留言更新、圖片、文字訊息
3. 這些輸入和輸出牽涉到不同訊息:
  - a. 訊息、頭貼是「我的」
  - b. 留言、回覆是「粉專」的



# 知道什麼叫難，才能夠閃開它

1. 前端複雜是因為它牽涉到的因素太多，每個因素背後都可能造成狀態改變，開發者會很像在打地鼠：
  - a. 圖形化使用者介面 (GUI): 這從來沒有簡單過，因為使用者可以在任何時候，介面上的任何地方跟程式互動
  - b. 牽涉到多台電腦：前端瀏覽器，和後端伺服器常常不在同一台電腦上，甚至也是不同語言寫的。這代表程式裡有些東西我們一直沒辦法確定和控制。

# 那什麼東西不難？

1. 一個輸入、一個輸出，同個輸入就是同個輸出。而且一個邏輯直通到底（線性的）。
2. 純資料分析的程式大部分能滿足這種條件：
  - a. 一群數字丟進pandas算平均數，只要資料相同，平均數就相同。
3. 雖然大家喜歡的前端就是不滿足這種條件，但我們可以從簡單的東西開始。
4. 簡單有兩個senses，因為簡單的東西不見得簡單。



# 簡單但不簡單

1. 例如，我覺得React簡單的地方：

- a. Component讓我們把複雜的頁面分門別類獨立出來
- b. Hooks 在詞彙層次上讓我們把會變的狀態跟不會變的單純邏輯區分開來
- c. JSX直覺的連結了視覺結構(HTML/CSS)跟程式邏輯(JS)

2. 我覺得React不簡單的點：

- a. 為什麼上面這些事情是簡單的

# Vanilla JS

1. 有的人動機夠強，就是想要直上React；那當然很好，但那是程式設計以外的動力：
  - a. 同儕壓力、時代精神.....
2. 但如果我們覺得React就是一大塊餅，不知從何下手，那我們當然有其他選擇：
  - a. 從VanillaJS開始寫也滿好的
  - b. 而且，React是設計來解決問題的，如果我們還沒遇到React要解決的問題，那...也很好阿(?!

Ready to try *Vanilla JS*? Choose exactly what you need!

Core Functionality

Prototype-based Object System

Animations

Regular Expressions

Closures

Array Library

DOM (Traversal / Selectors)

AJAX

Event System

Functions as first-class objects

Math Library

String Library

## Options

Minify Source Code

Use "CRLF" line breaks (Windows)

Produce UTF8 Output

**Final size:** 0 bytes uncompressed, 25 bytes gzipped.  Show human-readable sizes

Download

# 滿幽默的...

To use *Vanilla JS*, just put the following code anywhere in your application's HTML:

```
1. <script src="path/to/vanilla.js"></script>
```

When you're ready to move your application to a production deployment, switch to the much faster method:

```
1.
```

That's right - no code at all. *Vanilla JS* is *so popular* that browsers have been automatically loading it for over a decade.

# 但絕對不是要從vanilla JS苦練

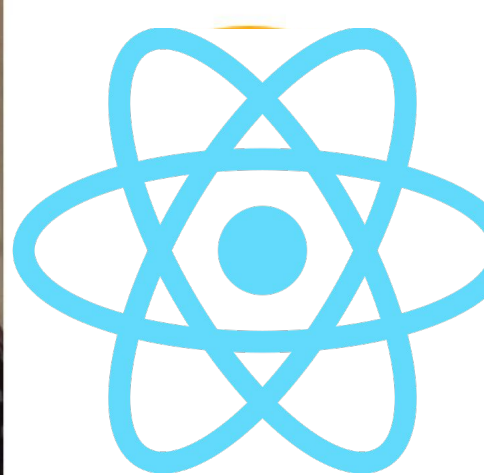
1. 學習不是線性的;講vanilla JS不是說要一步步從零開始慢慢爬。
2. Vanilla JS代表另一種前端設計的觀點和邏輯,有點old-school;如果這個不合你胃口, **完全沒關係**。
3. 但很可能你會發現某一行程式或某句話突然讓你莫名其妙就知道為什麼React要搞那麼複雜/簡單。
4. 然後就突然知道怎麼寫React了。

# 接下來

1. JS程式應該是個「回饋感」很強的東西
  - a. 有瀏覽器, 每寫一行就可以看看畫面現在變什麼樣子
2. 以前的JS有個好處, 一個檔案就可以跑
  - a. 不需要CRA
  - b. 不需要bundler/transpiler (那些CRA幫我們裝的東西)
3. 今天結合tensorflow.js, 我們有兩個這樣的專案:
  - a. Toxicity detection
  - b. QnA

# That smile 😊

這個meme適用於任何一開始  
看起來很美好但漸漸看見真相  
而甦醒成長的過程

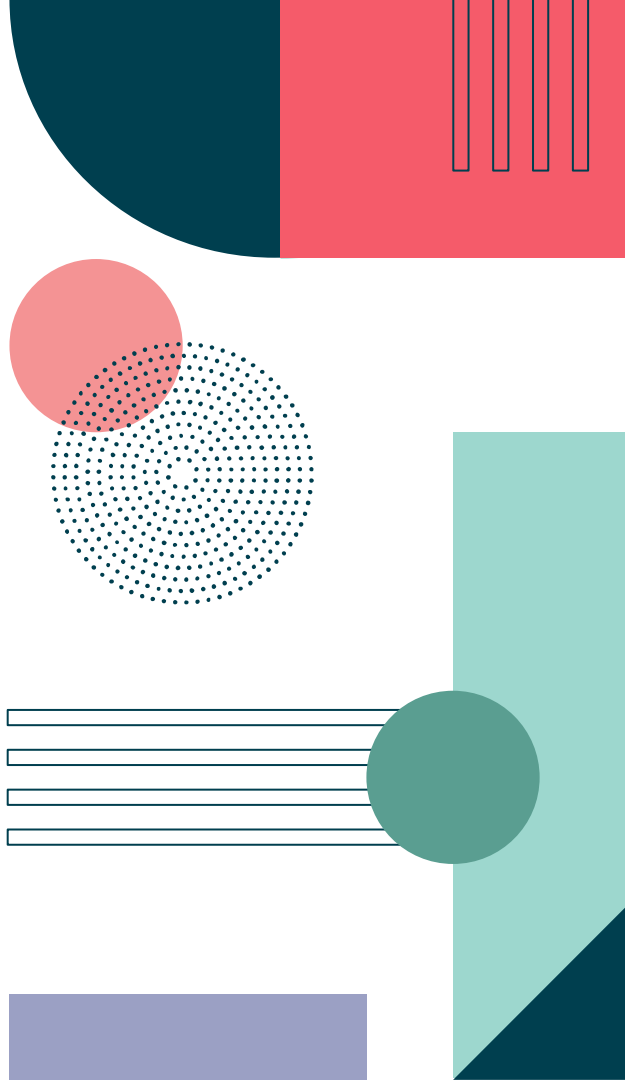




2

# Toxicity

用TFjs培養一點JS直覺





# Toxicity detection

1. 大致上長得像這樣
2. 這是單純的網頁，沒用到React、也不用node
3. 但保險起見，還是需要一個靜態的web server (如VSCode的Live Server)
4. 這頁在[這裡](#)

Toxicity with TFJs

stupid

insult(99%)

toxicity(99%)

```
<script>
  let tox_model = {};
  function load_model() {
    toxicity
      .load()
      .then(model => tox_model = model);
  }

  function detect_toxicity(sentence) {
    document.querySelector("#response-wrap").classList.toggle("visually-hidden");
    document.querySelector("#response-spinner").classList.toggle("visually-hidden");

    tox_model.classify([sentence]).then(predictions => {
      let matchedPreds = predictions.filter(x=>x.results[0].match);
      let results = "";
      for(let pred_x of matchedPreds){
        let prob = Math.round(pred_x.results[0].probabilities[1]*100);
        results += pred_x.label + `(${prob}%) <br>`;
      }
      document.querySelector("#response-spinner").classList.toggle("visually-hidden");
      document.querySelector("#response-wrap").classList.toggle("visually-hidden");
      document.querySelector("#pred-label").innerHTML = results
    })
  }
  load_model();
</script>
```

程式碼大概像這樣

# HTML 部分

```
<div id="container" class="container mt-5" >
  <div class="row text-center font-weight-bold mb-3 text-dark">
    <h2>Toxicity with TFjs</h2>
  </div>
  <div id="input-wrap" class="row">
    <div class="col text-center fs-3 mx-auto">
      <input type="text" class="w-50 rounded border-light"
        placeholder="input something"
        onChange="detect_toxicity(this.value)"/>
    </div>
  </div>
  <div class="row">
    <div id="response-spinner" class="visually-hidden col text-center mx-auto mt-5">
      <div class="spinner-border mx-auto text-dark">
        <span class="visually-hidden">Loading</span>
      </div>
    </div>
    <div id="response-wrap" class="col fs-2 text-center mx-auto mt-5">
      <span id="pred-label"></span>
    </div>
  </div>
</div>
```

為了介面漂亮，這個網頁有用到一個CSS套件，[Bootstrap](#)。

所以，在這頁中，所有class裡的東西都只是為了美觀，跟前面的程式沒關係。

# 以前的JS

1. 這種JS的寫法大概是2010年左右的風格。
2. 現在這種寫法少用，但不是因為不支援或效能，而是它複雜不起來，不符合現代前端的需求。
3. 但從這種風格中比較好理解JS的特性，而且即便在這短短幾行code裡，已經可以看到JS很獨特的思考方式：
  - a. 事件(event) 和回呼 (callback)
  - b. DOM manipulation

# 我們先確定一些基本觀念

1. HTML的元素可以直接指定事件，如onchange、onclick，這是HTML和JS互動的原型。事實上這個語法也沿用到JSX。
2. 相反的，JS要和HTML互動的原始方法是document.querySelector，或者大家可能比較熟悉的document.getElementById等，取得DOM object。
3. JS的執行不一定是從第一行到最後一行，每個函數都可能是觸發點：來自事件或回呼。

# 我們來做個sandbox

1. 請用VSCode打開一個專案，然後建立一個index.html
2. 在裡面先打好基本HTML5 結構(下一頁有template)。
3. 然後打開瀏覽器確定你可以預覽頁面
4. 然後用你想得到最簡單的方法，讓頁面上出現一些文字:hello world之類的東西也可以
5. 確定頁面上可以顯示出你剛打的東西。
6. sandbox的範例在[這裡](#)

# Barebone HTML5

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title></title>
</head>
<body>
</body>
</html>
```

# 試試看button的 onclick 事件

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title></title>
</head>
<body>
<button onclick="alert('hello')">Click me</button>
</body>
</html>
```



# 試試看input的 onchange 事件

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title></title>
</head>
<body>
<button onclick="alert('hello')">Click me</button>
<input onchange="alert(this.value)"/>
</body>
</html>
```

# What's "this"

1. input的onchange裡，我們用this.value當引數 (argument)
2. 這裡的this，因為出現在DOM事件 (onchange) 裡，所以指的是觸發事件的event target——文字方塊這個element。
3. this.value的 .value 是 DOM中 input element的內建屬性 (property)，意思是文字方塊中的內容。
4. JS的this概念相當複雜，算是大魔王等級；初學階段有碰到再瞭解就好。

# 把背景變成土黃色

1. 在事件屬性(onclick, onchange) 後面接的, 就是JS了。它完全就是在JS環境裡的東西。

```
<body>
<button onclick="alert('hello')">Click me</button>
<input onchange="alert(this.value)"/>
<button
  onclick="document.body.style.backgroundColor='bisque'">
  bisque</button>
</body>
</html>
```

# document 物件

1. `document`是瀏覽器內建的HTML DOM物件 (root object), 它讓我們可以用JS操作我們看到的HTML頁面。
2. 在以前沒有JSX的時代, 所有JS跟頁面的互動都是透過「操作DOM」(DOM manipulation) 完成的。像剛剛換背景顏色:
  - a. `document.body` 指的就是body元素
  - b. `document.body.style`是指這個元素的(CSS)樣式
  - c. `document.body.style.backgroundColor` 是指body的background-color, 我們可以直接改變這個屬性

# 把事件處理函數獨立出來

1. 跟剛剛完全一樣，只是把在onclick裡的東西獨立出來寫在<script>裡，把它取名為 snap() 函數 (function)，我們就不用擠在字串裡寫程式了。
2. <script> 裡的字串全部會被當成JS，裡面不能再出現HTML了

```
<button onclick="snap()">color</button>
<script>
  function snap(){
    document.body.style.backgroundColor = "bisque";
  }
</script>
```

請試試看打出這段

```
<script>
function snap(){
  console.log("snap called - enter");
  setTimeout(()=>{
    console.log("setTimeout callback - enter");
    document.body.style.backgroundColor="bisque";
    document.querySelector("#btn").textContent = "snap";
    console.log("setTimeout callback - exit");
  }, 500);
  console.log("snap called - exit");
}
</script>
```

# snap() 裡發生的事情

1. snap() 裡的揮灑空間很大，可以做任何在JS裡可以發生的事情。我們現在嘗試的是：
  - a. 按下按鈕後，過1000毫秒頁面才會變色。
  - b. 然後按鈕文字會變成 "snapped"
2. `document.querySelector`是用CSS selectors來找尋頁面上的HTML元素 (element)，並且傳回 DOM object。
  - a. HTML element: 頁面視覺上的元素，是我們看到的
  - b. DOM object: 是JS可操作的元素，是一個程式物件。

# setTimeout

snap called - enter	<code>.(index):13</code>
snap called - exit	<code>.(index):20</code>
setTimeout callback - enter	<code>.(index):15</code>
setTimeout callback - exit	<code>.(index):18</code>

1. setTimeout是JS的內建函數，它會延後某個函數的執行時間。它接收兩個參數：
  - a. 要被延後執行的函數
  - b. 被延後的時間，以毫秒為單位。
2. snap() 裡的函數是用arrow function語法，`() => {...}` 代表一個沒有參數的函數。
3. 我們在snap() 裡加了很多console.log，這可以告訴我們函數的執行順序。



# JS的程式執行順序

1. 很重要的觀念：電腦讀程式都是從上往下讀，這幾乎不會有例外，JS也是。所以變數宣告的順序是固定的，用到之前要先宣告。
2. 原則上，**同個函數裡**的程式碼都是一行接一行順序跑完。這是同步(synchronous)的。
3. 但是，像setTimeout：既然都要延後函數執行，那「裡層」函數的執行順序一定會在「外層」之後，與程式碼看到的執行順序不同。它就是非同步的(asynchronous)

# JS的非同步特性

1. 這種「非同步特性」很違反人類直覺，所以描述這件事情的語言（語法、詞彙），在這10年間JS出現了好多種：
  - a. 最早是回呼函數 (callback function), setTimeout就是這種。
  - b. 現在比較常用的是Promise:「我現在沒有結果，但我保證等一下給你結果」
  - c. Promise概念很抽象，async/await也是在同個脈絡裡的事情。這在現代JS裡都是「不簡單的簡單」的事。

# 小結一下

1. 在HTML元素的事件屬性(e.g., onclick)裡，我們可以用JS描述事件發生後「要做什麼」
2. `<script>`裡可以寫所有JS，包含撰寫各種函數。
3. JS裡可以透過 `document.querySelector(...)` 取得代表HTML元素的DOM object
4. JS的程式執行順序不見得是程式碼的書寫順序；在現代的JS裡，這通常發生在DOM事件或Promise。
5. 例如，在toxicity的程式碼裡。

# Promise

- Toxicity裡用到兩個 Promises: 一個是載入模型, 一個是實際偵測。
- toxicity是引入tfjs提供的物件。
- toxicity.load()會開始下載模型, 並載入瀏覽器, 這步可能很久 (~sec)。
- 這裡的Promise很像:「我正在載入模型, 你先忙其他事, 好了叫你。」我們被叫後要幹嘛, 就是then(...)裡寫的事

```
let tox_model = {};  
// Load the model.  
function load_model() {  
  toxicity  
    .load() 這裡回傳Promise  
    .then(model => tox_model = model);  
}
```

## getImage

Credit: [Lydia Hallie](#)



```
.then(res => console.log(res))
```

```
.catch(err => console.log(err))
```

```
node  
  
> getImage("./image.png")  
  .then(res => console.log(res))  
  .catch(err => console.log(err))
```

## getImage

Credit: [Lydia Hallie](#)



```
.then(res => console.log(res))
```

```
.catch(err => console.log(err))
```

```
node  
  
> getImage("./image.png")  
  .then(res => console.log(res))  
  .catch(err => console.log(err))
```

# 整個程式流程

1. 頁面會先載入模型：`<script>`的最後一行。
2. 接著頁面就在等使用者動作。當使用者完成文字輸入，按下Enter後，HTML觸發onchange事件
3. HTML呼叫JS的`detect_toxicity(...)`；在函數裡，`tox_model.classify(...)`會負責預測，但它同樣也是傳回promise。
4. 所以回傳處理結果的程式，就要發生在`Promise.then(...)`的回呼函數裡。

# 頁面的HTML

```
<div id="container" class="container mt-5" >
  <div class="row text-center font-weight-bold mb-3 text-dark">
    <h2>Toxicity with TFjs</h2>
  </div>
  <div id="input-wrap" class="row">
    <div class="col text-center fs-3 mx-auto">
      <input type="text" class="w-50 rounded border-light"
        placeholder="input something"
        onChange="detect_toxicity(this.value)"/>
    </div>
  </div>
  <div class="row">
    <div id="response-spinner" class="visually-hidden col text-center mx-auto mt-5">
      <div class="spinner-border mx-auto text-dark">
        <span class="visually-hidden">Loading</span>
      </div>
    </div>
    <div id="response-wrap" class="col fs-2 text-center mx-auto mt-5">
      <span id="pred-label"></span>
    </div>
  </div>
</div>
```



```
<script>
  let tox_model = {};
  function load_model() {
    toxicity
      .load()
      .then(model => tox_model = model);
  }

  function detect_toxicity(sentence) {
    document.querySelector("#response-wrap").classList.toggle("visually-hidden");
    document.querySelector("#response-spinner").classList.toggle("visually-hidden");

    tox_model.classify([sentence]).then(predictions => {
      let matchedPreds = predictions.filter(x=>x.results[0].match);
      let results = "";
      for(let pred_x of matchedPreds){
        let prob = Math.round(pred_x.results[0].probabilities[1]*100);
        results += pred_x.label + `(${prob}%) <br>`;
      }
      document.querySelector("#response-spinner").classList.toggle("visually-hidden");
      document.querySelector("#response-wrap").classList.toggle("visually-hidden");
      document.querySelector("#pred-label").innerHTML = results

    })
  }
  load_model();
</script>
```

顯示「轉圈圈」

```
<script>
  let tox_model = {};
  function load_model() {
    toxicity
      .load()
      .then(model => tox_model = model);
  }

  function detect_toxicity(sentence) {
    document.querySelector("#response-wrap").classList.toggle("visually-hidden");
    document.querySelector("#response-spinner").classList.toggle("visually-hidden");

    tox_model.classify([sentence]).then(predictions => {
      let matchedPreds = predictions.filter(x=>x.results[0].match);
      let results = "";
      for(let pred_x of matchedPreds){
        let prob = Math.round(pred_x.results[0].probabilities[1]*100);
        results += pred_x.label + `(${prob}%) <br>`;
      }
      document.querySelector("#response-spinner").classList.toggle("visually-hidden");
      document.querySelector("#response-wrap").classList.toggle("visually-hidden");
      document.querySelector("#pred-label").innerHTML = results

    })
  }
  load_model();
</script>
```

**這裡是 Promise!!**  
有偵測結果後，  
整理原始的偵測  
結果，然後建立  
要顯示的字串

```
<script>
  let tox_model = {};
  function load_model() {
    toxicity
      .load()
      .then(model => tox_model = model);
  }

  function detect_toxicity(sentence) {
    document.querySelector("#response-wrap").classList.toggle("visually-hidden");
    document.querySelector("#response-spinner").classList.toggle("visually-hidden");

    tox_model.classify([sentence]).then(predictions => {
      let matchedPreds = predictions.filter(x=>x.results[0].match);
      let results = "";
      for(let pred_x of matchedPreds){
        let prob = Math.round(pred_x.results[0].probabilities[1]*100);
        results += pred_x.label + ` (${prob}%) <br>`;
      }
      document.querySelector("#response-spinner").classList.toggle("visually-hidden");
      document.querySelector("#response-wrap").classList.toggle("visually-hidden");
      document.querySelector("#pred-label").innerHTML = results
    })
  }
  load_model();
</script>
```

把轉圈圈關掉, 然後呈現結果

# 這可以寫成React嗎？

1. 當然可以，但就目前的程式需求，以及TFjs的程式架構，好像划不來；而且可能要照顧一些額外的細節。
2. TFjs完全只跟模型有關：一個輸入一個輸出。它的複雜性不在頁面狀態，React不會讓它更簡單，雖然：
  - a. 「模型」和「預測結果」可想成狀態(state)
  - b. 不用做DOM manipulation, React有JSX
3. Promise.then(...)是必然的，React不會改變TFjs的非同步
4. PS. React有約200M的deps、剛剛的網頁只有3K

# 下次再來寫React

1. React是JS的「架構」(framework): 他幫助我們講一些用JS很難講的事情, 但JS本身已經能講滿多事的了。
2. Tensorflow.js想要在瀏覽器做DL, 但目前看起來有很多挑戰。官方支援模型不多, 自己轉模型的話需要很多技術工。
3. 下次我們會用React接Deep learning, 不過是用Huggingface包好的inference API, 就有中文模型可以用了。

# React

# Angular

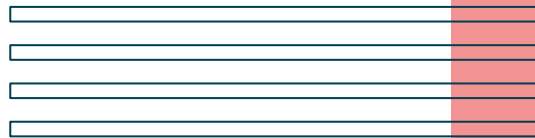


vanilla javascript

Credit: [Ben Awad](#), SrGrafo



SRGRAFO



## NLP與網路應用

聊JS (TFjs)

