给 JavaScript 初心者的 ES2015 实战

作者：小问

给 JavaScript 初心者的 ES2015 实战

前言

历时将近6年的时间来制定的新 ECMAScript 标准 ECMAScript 6（亦称 ECMAScript Harmony，简称 ES6）终于在 2015 年 6 月正式发布。自从上一个标准版本 ES5 在 2009 年发布以后，ES6 就一直以**新语法**、**新特性**的优越性吸引著众多 JavaScript 开发者，驱使他们积极尝鲜。

虽然至今各大浏览器厂商所开发的 JavaScript 引擎都还没有完成对 ES2015 中所有特性的完美支持，但这并不能阻挡工程师们对 ES6 的热情，于是乎如 babel、Traceur 等编译器便出现了。它们能将尚未得到支持的 ES2015 特性转换为 ES5 标准的代码，使其得到浏览器的支持。其中，babel 因其**模块化转换器(Transformer)**的设计特点赢得了绝大部份 JavaScript 开发者的青睐，本文也将以 babel 为基础工具，向大家展示 ES2015 的神奇魅力。

笔者目前所负责的项目中，已经在前端和后端全方位的使用了 ES2015 标准进行 JavaScript 开发，已有将近两年的 ES2015 开发经验。如今 ES2015 以成为 ECMA 国际委员会的首要语言标准，使用 ES2015 标准所进行的工程开发已打好了坚实的基础，而 ES7（ES2016） 的定制也走上了正轨，所以在这个如此恰当的时机，我觉得应该写一篇通俗易懂的 ES2015 教程来引导广大 JavaScript 爱好者和工程师向新时代前进。若您能从本文中有所收获，便是对我最大的鼓励。

我希望你在阅读本文前，已经掌握了 JavaScript 的基本知识，并具有一定的 Web App 开发基础和 Node.js 基本使用经验。

目录

本文的实战部份将以开发一个动态博客系统为背景，向大家展示如何使用 ES2015 进行项目开发。成品代码将在 GitHub 上展示。

一言蔽之 ES2015

说到 ES2015，有了解过的同学一定会马上想到各种新语法，如箭头函数（`=>`）、`class`、模板字符串等。是的，ECMA 委员会吸取了许多来自全球众多 JavaScript 开发者的意见和来自其他优秀编程语言的经验，致力于制定出一个更适合现代 JavaScript 开发的标准，以达到“和谐”(Harmony)。一言蔽之：

> ES2015 标准提供了许多新的语法和编程特性以提高 JavaScript 的开发效率和体验

从 ES6 的别名被定为 Harmony 开始，就注定了这个新的语言标准将以一种更优雅的姿态展现出来，以适应日趋复杂的应用开发需求。

ES2015 能为 JavaScript 的开发带来什么

语法糖

如果您有其他语言（如 Ruby、Scala）或是某些 JavaScript 的衍生语言（如 CoffeeScript、TypeScript）的开发经验，就一定会了解一些很有意思的语法糖，如 Ruby 中的 `Range -> 1..10`，Scala 和 CoffeeScript 中的箭头函数 `(a, b) => a + b`。ECMA 委员会**借鉴**了许多其他编程语言的标准，给 ECMAScript 家族带来了许多可用性非常高的语法糖，下文将会一一讲解。

这些语法糖能让 JavaScript 开发者更舒心地开发 JavaScript 应用，提高我们的工作效率~~，多一些时间出去浪~~。

工程优势

ES2015 除了提供了许多语法糖以外，还由官方解决了多年来困扰众多 JavaScript 开发者的问题：JavaScript 的模块化构建。从许多年前开始，各大公司、团队、大牛都相继给出了他们对于这个问题的不同解决方案，以至于定下了如 CommonJS、AMD、CMD 或是 UMD 等 JavaScript 模块化标准，RequireJS、SeaJS、FIS、Browserify、webpack 等模块加载库都以各自不同的优势占领著一方土地。

然而正正是因为这春秋战国般的现状，广大的前端搬砖工们表示很纳闷。

> 这™究竟哪种好？哪种适合我？求大神带我飞！

对此，ECMA 委员会终于是坐不住了，站了起来表示不服，并制订了 ES2015 的原生模块加载器标准。

```
import fs from 'fs'
import readline from 'readline'
import path from 'path'

let Module = {
  readLineInFile(filename, callback = noop, complete = noop) {
    let rl = readline.createInterface({
      input: fs.createReadStream(path.resolve(__dirname, './big_file.txt'))
    })

    rl.on('line', line => {
      //... do something with the current line
      callback(line)
    })

    rl.on('close', complete)

    return rl
  }
```

```
}

function noop() { return false }

export default Module
```

~~老实说，这套模块化语法不禁让我们又得要对那个很 silly 的问题进行重新思考了：JavaScript 和 Java 有什么关系？~~

可惜的是，目前暂时还没有任何浏览器厂商或是 JavaScript 引擎支持这种模块化语法。所以我们需要用 babel 进行转换为 CommonJS、AMD 或是 UMD 等模块化标准的语法。

ES2015 新语法详解

经过以上的介(xun)绍(tao)，相信你对 ES2015 也有了一定的了解和期待。接下来我将带大家慢慢看看 ECMA 委员会含辛茹苦制定的新语言特性吧。

`let`、`const` 和块级作用域

在 ES2015 的新语法中，影响速度最为直接，范围最大的，恐怕得数 `let` 和 `const` 了，它们是继 `var` 之后，新的变量定义方法。与 `let` 相比，`const` 更容易被理解：`const` 也就是 **constant** 的缩写，跟 C/C++ 等经典语言一样，用于定义常量，即不可变量。

但由于在 ES6 之前的 ECMAScript 标准中，并没有原生的实现，所以在降级编译中，会马上进行引用检查，然后使用 `var` 代替。

```
// foo.js
const foo = 'bar'

foo = 'newvalue'
```

```
$ babel foo.js
...
SyntaxError: test.js: Line 3: "foo" is read-only
  1 | const foo = 'bar'
  2 |
> 3 | foo = 'newvalue'
...
```

块级作用域

在 ES6 诞生之前，我们在给 JavaScript 新手解答困惑时，经常会提到一个观点：

> JavaScript 没有块级作用域

在 ES6 诞生之前的时代中，JavaScript 确实是没有块级作用域的。这个问题之所以为人所熟知，是因为它引发了诸如历遍监听事件需要使用闭包解决等问题。

```
<button>一</button>
<button>二</button>
<button>三</button>
<button>四</button>

<div id="output"></div>

<script>
  var buttons = document.querySelectorAll('button')
  var output = document.querySelector('#output')

  for (var i = 0; i < buttons.length; i++) {
    buttons[i].addEventListener('click', function() {
      output.innerText = buttons[i].innerText
    })
  }
```

```
</script>
```

前端新手非常容易写出类似的代码，因为从直观的角度看这段代码并没有语义上的错误，但是当我们点击任意一个按钮时，就会报出这样的错误信息：

```
Uncaught TypeError: Cannot read property 'innerText' of undefined
```

出现这个错误的原因是因为 `buttons[i]` 不存在，即为 `undefined`。

为什么会出现按钮不存在结果呢？通过排查，我们可以发现，每次我们点击按钮时，事件监听回调函数中得到的变量 `i` 都会等于 `buttons.length`，也就是这里的 4。而 `buttons[4]` 恰恰不存在，所以导致了错误的发生。

再而导致 `i` 得到的值都是 `buttons.length` 的原因就是因为 JavaScript 中没有块级作用域，而使对 `i` 的变量引用(Reference)一直保持在上一层作用域（循环语句所在层）上，而当循环结束时 `i` 则正好是 `buttons.length`。

而在 ES6 中，我们只需做出一个小小的改动，便可以解决该问题（假设所使用的浏览器已经支持所需要的特性）：

```
// ...
for (/* var */ let i = 0; i < buttons.length; i++) {
  // ...
}
// ...
```

通过把 `for` 语句中对计数器 `i` 的定义语句从 `var` 换成 `let`，即可。因为 `let` 语句会使该变量处于一个块级作用域中，从而让事件监听回调函数中的变量引用得到保持。我们不妨看看改进后的代码经过 babel 的编译会变成什么样子：

```
// ...
```

```
var _loop = function (i) {
  buttons[i].addEventListener('click', function () {
    output.innerText = buttons[i].innerText
  })
}

for (var i = 0; i < buttons.length; i++) {
  _loop(i)
}
// ...
```

实现方法一目了然，通过传值的方法防止了 `i` 的值错误。

箭头函数(Arrow Function)

继 `let` 和 `const` 之后，箭头函数就是使用率最高的新特性了。当然了，如果你了解过 Scala 或者曾经如日中天的 JavaScript 衍生语言 CoffeeScript，就会知道箭头函数并非 ES6 独创。

箭头函数，顾名思义便是使用箭头(`=>`)进行定义的函数，属于匿名函数（Lambda）一类。当然了，也可以作为定义式函数使用，但我们并不推荐这样做，随后会详细解释。

使用

箭头函数有好几种使用语法：

```
1. foo => foo + ' world' // means return `foo + ' world'`
2. (foo, bar) => foo + bar
3.
foo => {
  return foo + ' world'
}
4.
(foo, bar) => {
```

```
  return foo + bar
}
```

以上都是被支持的箭头函数表达方式，其最大的好处便是简洁明了，省略了 `function` 关键字，而使用 `=>` 代替。

箭头函数语言简洁的特点使其特别适合用於单行回调函数的定义：

```
let names = [ 'Will', 'Jack', 'Peter', 'Steve', 'John', 'Hugo', 'Mike' ]

let newSet = names
  .map((name, index) => {
    return {
      id: index,
      name: name
    }
  })
  .filter(man => man.id % 2 == 0)
  .map(man => [man.name])
  .reduce((a, b) => a.concat(b))

console.log(newSet) //=> [ 'Will', 'Peter', 'John', 'Mike' ]
```

如果你有 Scala + Spark 的开发经验，就一定会觉得这非常亲切，因为这跟其中的 RDD 操作几乎如出一辙：

1. 将原本的由名字组成的数组转换为一个格式为 `{ id, name }` 的对象，`id` 则为每个名字在原数组中的位置
2. 剔除其中 `id` 为奇数的元素，只保留 `id` 为偶数的元素
3. 将剩下的元素转换为一个包含当前元素中原名字的单元数组，以方便下一步的处理
4. 通过不断合并相邻的两个数组，最后能得到的一个数组，便是我们需要得到的目标值

箭头函数与上下文绑定

事实上，箭头函数在 ES2015 标准中，并不只是作为一种新的语法出现。就如同它在 CoffeeScript 中的定义一般，是用于对函数内部的上下文（`this`）绑定为定义函数所在的作用域的上下文。

```
let obj = {
  hello: 'world',
  foo() {
    let bar = () => {
      return this.hello
    }
    return bar
  }
}

window.hello = 'ES6'
window.bar = obj.foo()
window.bar() //=> 'world'
```

上面代码中的 `obj.foo` 等价于：

```
// ...
foo() {
  let bar = (function() {
    return this.hello
  }).bind(this)

  return bar
}
// ...
```

为什么要为箭头函数给予这样的特性呢？我们可以假设出这样的一个应用场景，我们需要创建一个实例，用于对一些数据进行查询和筛选。

```
let DataCenter = {
  baseUrl: 'http://example.com/api/data',
  search(query) {
    fetch(`${this.baseUrl}/search?query=${query}`)
      .then(res => res.json())
      .then(rows => {
        // TODO
      })
  }
}
```

此时，从服务器获得数据是一个 JSON 编码的数组，其中包含的元素是若干元素的 ID，我们需要另外请求服务器的其他 API 以获得元素本身（当然了，实际上的 API 设计大部份不会这么使用这么蛋疼的设计）。我们就需要在回调函数中再次使用 `this.baseUrl` 这个属性，如果要同时兼顾代码的可阅读性和美观性，ES2015 允许我们这样做。

```
let DataCenter = {
  baseUrl: 'http://example.com/api/data',
  search(query) {
    return fetch(`${this.baseUrl}/search?query=${query}`)
      .then(res => res.json())
      .then(rows => {
        return fetch(`${this.baseUrl}/fetch?ids=${rows.join(',')}`)
        // 此处的 this 是 DataCenter，而不是 fetch 中的某个实例
      })
      .then(res => res.json())
  }
}

DataCenter.search('iwillwen')
```

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

因为在单行匿名函数中，如果 `this` 指向的是该函数的上下文，就会不符合直观的语义表达。

注意事项

另外，要注意的是，箭头函数对上下文的绑定是强制性的，无法通过 `apply` 或 `call` 方法改变其上下文。

```
let a = {
  init() {
    this.bar = () => this.dam
  },
  dam: 'hei',
  foo() {
    return this.dam
  }
}

let b = {
  dam: 'ha'
}

a.init()

console.log(a.foo()) //=> hei
console.log(a.foo.bind(b).call(a)) //=> ha
console.log(a.bar.call(b)) //=> hei
```

另外，因为箭头函数会绑定上下文的特性，故不能随意在顶层作用域使用箭头函数，以防出错：

```
// 假设当前运行环境为浏览器，故顶层作上下文为 `window`
let obj = {
  msg: 'pong',

  ping: () => {
    return this.msg // Warning!
  }
}

obj.ping() //=> undefined
let msg = 'bang!'
obj.ping() //=> bang!
```

为什么上面这段代码会如此让人费解呢？

我们来看看它的等价代码吧。

```
let obj = {
  // ...
  ping: (function() {
    return this.msg // Warning!
  }).bind(this)
}

// 同样等价于
let obj = { /* ... */ }
obj.ping = (function() {
  return this.msg
}).bind(this /* this -> window */)
```

模板字符串

模板字符串模板出现简直对 Node.js 应用的开发和 Node.js 自身的发展起到了相当大的推动作用！我的意思并不是说这个原生的模板字符串能代替现有的模板引擎，而是说它的出现可以让非常多的字符串使用变得尤为轻松。

模板字符串要求使用 ` 代替原本的单/双引号来包裹字符串内容。它有两大特点：

1. 支持变量注入
2. 支持换行

支持变量注入

模板字符串之所以称之为“模板”，就是因为它允许我们在字符串中引用外部变量，而不需要像以往需要不断地相加、相加、相加……

```
let name = 'Will Wen Gunn'
let title = 'Founder'
let company = 'LikMoon Creation'

let greet = `Hi, I'm ${name}, I am the ${title} at ${company}`
console.log(greet) //=> Hi, I'm Will Wen Gunn, I am the Founder at LikMo
```

支持换行

在 Node.js 中，如果我们没有支持换行的模板字符串，若需要拼接一条SQL，则很有可能是这样的：

```
var sql =
  "SELECT * FROM Users " +
  "WHERE FirstName='Mike' " +
  "LIMIT 5;"
```

或者是这样的：

```
var sql = [
  "SELECT * FROM Users",
  "WHERE FirstName='Mike'",
  "LIMIT 5;"
].join(' ')
```

无论是上面的哪一种，都会让我们感到很不爽。但若使用模板字符串，仿佛打开了新世界的大门~

```
let sql = `
SELECT * FROM Users
WHERE FirstName='Mike'
LIMIT 5;
`
```

Sweet! 在 Node.js 应用的实际开发中，除了 SQL 的编写，还有如 Lua 等嵌入语言的出现（如 Redis 中的 SCRIPT 命令），或是手工的 XML 拼接。模板字符串的出现使这些需求的解决变得不再纠结了~

对象字面量扩展语法

看到这个标题的时候，相信有很多同学会感到奇怪，对象字面量还有什么可以扩展的？

确实，对象字面量的语法在 ES2015 之前早已挺完善的了。不过，对于聪明的工程师们来说，细微的改变，也能带来不少的价值。

方法属性省略 `function`

这个新特性可以算是比较有用但并不是很显眼的一个。

```
let obj = {
  // before
```

```
  foo: function() {
    return 'foo'
  },

  // after
  bar() {
    return 'bar'
  }
}
```

支持 `__proto__` 注入

在 ES2015 中，我们可以给一个对象硬生生的赋予其 `__proto__`，这样它就可以成为这个值所属类的一个实例了。

```
class Foo {
  constructor() {
    this.pingMsg = 'pong'
  }

  ping() {
    console.log(this.pingMsg)
  }
}

let o = {
  __proto__: new Foo()
}

o.ping() //=> pong
```

什么？有什么卵用？

有一个比较特殊的场景会需要用到：我想扩展或者覆盖一个类的方法，并生成一个实例，但觉得另外定义一个类就感觉浪费了。那我可以这样做：

```
let o = {
  __proto__: new Foo(),

  constructor() {
    this.pingMsg = 'alive'
  },

  msg: 'bang',
  yell() {
    console.log(this.msg)
  }
}

o.yell() //=> bang
o.ping() //=> alive
```

同名方法属性省略语法

也是看上去有点鸡肋的新特性，不过在做 JavaScript 模块化工程的时候则有了用武之地。

```
// module.js
export default {
  someMethod
}

function someMethod() {
  // ...
}

// app.js
```

```
import Module from './module'


Module.someMethod()
```

可以动态计算的属性名称

这个特性相当有意思，也是可以用在一些特殊的场景中。

```
let arr = [1, 2, 3]
let outArr = arr.map(n => {
  return {
    [ n ]: n,
    [ `${n}^2` ]: Math.pow(n, 2)
  }
})
console.dir(outArr) //=>
  [
    { '1': 1, '1^2': 1 },
    { '2': 2, '2^2': 4 },
    { '3': 3, '3^2': 9 }
  ]
```

在上面的两个 `[...]` 中，我演示了动态计算的对象属性名称的使用，分别为对应的对象定义了当前计数器 `n` 和 `n` 的 2 次方

表达式解构

来了来了来了，相当有用的一个特性。有啥用？多重复值听过没？没听过？来看看吧！

```
// Matching with object
function search(query) {
  // ...
```

```
  // let users = [ ... ]
  // let posts = [ ... ]
  // ...

  return {
    users: users,
    posts: posts
  }
}

let { users, posts } = search('iwillwen')

// Matching with array
let [ x, y ] = [ 1, 2 ]
// missing one
[ x, ,y ] = [ 1, 2, 3 ]


function g({name: x}) {
  console.log(x)
}
g({name: 5})
```

还有一些可用性不大，但也是有一点用处的：

```
// Fail-soft destructuring
var [a] = []
a === undefined //=> true

// Fail-soft destructuring with defaults
var [a = 1] = []
a === 1 //=> true
```

函数参数表达、传参

这个特性有非常高的使用频率，一个简单的语法糖解决了从前需要一两行代码才能实现的功能。

默认参数值

这个特性在类库开发中相当有用，比如实现一些可选参数：

```
import fs from 'fs'
import readline from 'readline'
import path from 'path'

function readLineInFile(filename, callback = noop, complete = noop) {
  let rl = readline.createInterface({
    input: fs.createReadStream(path.resolve(__dirname, filename))
  })

  rl.on('line', line => {
    //... do something with the current line
    callback(line)
  })

  rl.on('close', complete)

  return rl
}

function noop() { return false }

readLineInFile('big_file.txt', line => {
  // ...
})
```

后续参数

我们知道，函数的 `call` 和 `apply` 在使用上的最大差异便是一个在首参数后传入各个参数，一个是在首参数后传入一个包含所有参数的数组。如果我们在实现某些函数或方法时，也希望实现像 `call` 一样的使用方法，在 ES2015 之前，我们可能需要这样做：

```
function fetchSomethings() {
  var args = [].slice.apply(arguments)

  // ...
}
function doSomeOthers(name) {
  var args = [].slice.apply(arguments, 1)

  // ...
}
```

而在 ES2015 中，我们可以很简单的使用 `...` 语法糖来实现：

```
function fetchSomethings(...args) {
  // ...
}
function doSomeOthers(name, ...args) {
  // ...
}
```

要注意的是，`...args` 后不可再添加

虽然从语言角度看，`arguments` 和 `...args` 是可以同时使用 ，但有一个特殊情况则不可：`arguments` 在箭头函数中，会跟随上下文绑定到上层，所以在不确定上下文绑定结果的情况下，尽可能不要再箭头函数中再使用 `arguments`，而使用 `...args`。

虽然 ECMA 委员会和各类编译器都无强制性要求用 `...args` 代替 `arguments`，但从实践

经验看来，`...args` 确实可以在绝大部份场景下可以代替 `arguments` 使用，除非你有很特殊的场景需要使用到 `arguments.callee` 和 `arguments.caller`。所以我推荐都使用 `...args` 而非 `arguments`。

PS：在严格模式（Strict Mode）中，`arguments.callee` 和 `arguments.caller` 是被禁止使用的。

解构传参

在 ES2015 中，`...` 语法还有另外一个功能：无上下文绑定的 `apply`。什么意思？看看代码你就知道了。

```
function sum(...args) {
  return args.map(Number)
    .reduce((a, b) => a + b)
}

console.log(sum(...[1, 2, 3])) //=> 6
```

有什么卵用？我也不知道(⊙o⊙)... Sorry...

注意事项

默认参数值和**后续参数**需要遵循顺序原则，否则会出错。

```
function(...args, last = 1) {
  // This will go wrong
}
```

另外，根据函数调用的原则，无论是默认参数值还是后续参数都需要小心使用。

新的数据结构

在介绍新的数据结构之前，我们先复习一下在 ES2015 之前，JavaScript 中有哪些基本的数据结构。

- String 字符串
- Number 数字（包含整型和浮点型）
- Boolean 布尔值
- Object 对象
- Array 数组

其中又分为**值类型**和**引用类型**，Array 其实是 Object 的一种子类。

Set 和 WeakSet

我们再来复习下高中数学吧，集不能包含相同的元素，我们可以根据元素画出多个集的韦恩图.....

好了跑题了。是的，在 ES2015 中，ECMA 委员会为 ECMAScript 增添了集(Set)和“弱”集(WeakSet)。它们都具有元素唯一性，若添加了已存在的元素，会被自动忽略。

```
let s = new Set()
s.add('hello').add('world').add('hello')
console.log(s.size) //=> 2
console.log(s.has('hello')) //=> true
```

在实际开发中，我们有很多需要用到集的场景，如搜索、索引建立等。

咦？怎么还有一个 WeakSet？这是干什么的？我曾经写过一篇关于 JavaScript 内存优化的文章，而其中大部份都是在语言上动手脚，而 WeakSet 则是在数据上做文章。

WeakSet 在 JavaScript 底层作出调整（在非降级兼容的情况下），检查元素的变量引用情况。如果元素的引用已被全部解除，则该元素就会被删除，以节省内存空间。这意味著无法直接加入数字或者字符串。另外 WeakSet 对元素有严格要求，必须是 Object，当然

了，你也可以用 `new String('...')` 等形式处理元素。

```
let weaks = new WeakSet()
weaks.add("hello") //=> Error
weaks.add(3.1415) //=> Error

let foo = new String("bar")
let pi = new Number(3.1415)
weaks.add(foo)
weaks.add(pi)
weaks.has(foo) //=> true
foo = null
weaks.has(foo) //=> false
```

Map 和 WeakMap

从数据结构的角度来说，映射（Map）跟原本的 Object 非常相似，都是 Key/Value 的键值对结构。但是 Object 有一个让人非常不爽的限制：key 必须是字符串或数字。在一般情况下，我们并不会遇上这一限制，但若我们需要建立一个对象映射表时，这一限制显得尤为棘手。

而 Map 则解决了这一问题，可以使用任何对象作为其 key，这可以实现从前不能实现或难以实现的功能，如在项目逻辑层实现数据索引等。

```
let map = new Map()
let object = { id: 1 }

map.set(object, 'hello')
map.set('hello', 'world')
map.has(object) //=> true
map.get(object) //=> hello
```

而 WeakMap 和 WeakSet 很类似，只不过 WeakMap 的键和值都会检查变量引用，只要其一的引用全被解除，该键值对就会被删除。

```
let weakm = new WeakMap()
let keyObject = { id: 1 }
let valObject = { score: 100 }

weakm.set(keyObject, valObject)
weakm.get(keyObject) //=> { score: 100 }
keyObject = null
weakm.has(keyObject) //=> false
```

类(Classes)

类，作为自 JavaScript 诞生以来最大的痛点之一，终于在 ES2015 中得到了官方的妥协，**“实现”**了 ECMAScript 中的标准类机制。为什么是带有双引号的呢？因为我们不难发现这样一个现象：

```
$ node
> class Foo {}
[Function: Foo]
```

回想一下在 ES2015 以前的时代中，我们是怎么在 JavaScript 中实现类的？

```
function Foo() {}
var foo = new Foo()
```

是的，ES6 中的类只是一种语法糖，用于定义**原型(Prototype)**的。当然，饿死的厨师三百斤，有总比没有强，我们还是很欣然地接受了这一设定。

语法

定义

与大多数人所期待的一样，ES2015 所带来的类语法确实与很多 C 语言家族的语法相似。

```
class Person {
  constructor(name, gender, age) {
    this.name = name
    this.gender = gender
    this.age = age
  }

  isAdult() {
    return this.age >= 18
  }
}

let me = new Person('iwillwen', 'man', 19)
console.log(me.isAdult()) //=> true
```

与 JavaScript 中的对象字面量不一样的是，类的属性后不能加逗号，而对象字面量则必须要加逗号。

然而，让人很不爽的是，ES2015 中对类的定义依然不支持默认属性的语法：

```
// 理想型
class Person {
  name: String
  gender = 'man'
  // ...
}
```

而在 TypeScript 中则有良好的实现。

继承

ES2015 的类继承总算是为 JavaScript 类继承之争抛下了一根定海神针了。在此前，有各种 JavaScript 的继承方法被发明和使用。（详细请参见《JavaScript 高级程序设计》）

```
class Animal {
  yell() {
    console.log('yell')
  }
}

class Person extends Animal {
  constructor(name, gender, age) {
    super() // must call `super` before using `this` if this class has a

    this.name = name
    this.gender = gender
    this.age = age
  }

  isAdult() {
    return this.age >= 18
  }
}

class Man extends Person {
  constructor(name, age) {
    super(name, 'man', age)
  }
}

let me = new Man('iwillwen', 19)
console.log(me.isAdult()) //=> true
```

```
me.yell()
```

同样的，继承的语法跟许多语言中的很类似，ES2015 中若要是一个类继承于另外一个类而作为其子类，只需要在子类的名字后面加上 `extends {SuperClass}` 即可。

静态方法

ES2015 中的类机制支持 `static` 类型的方法定义，比如说 `Man` 是一个类，而我希望为其定义一个 `Man.isMan()` 方法以用于类型检查，我们可以这样做：

```
class Man {
  // ...

  static isMan(obj) {
    return obj instanceof Man
  }
}

let me = new Man()
console.log(Man.isMan(me)) //=> true
```

遗憾的是，ES2015 的类并不能直接地定义静态成员变量，但若必须实现此类需求，可以用`static` 加上 `get` 语句和 `set` 语句实现。

```
class SyncObject {
  // ...

  static get baseUrl() {
    return 'http://example.com/api/sync'
  }
}
```

遗憾与期望

就目前来说，ES2015 的类机制依然很鸡肋：

1. 不支持私有属性（`private`）
2. 不支持前置属性定义，但可用 `get` 语句和 `set` 语句实现
3. 不支持多重继承
4. 没有类似于协议（`Protocl`）或接口（`Interface`）等的概念

中肯地说，ES2015 的类机制依然有待加强。但总的来说，是值得尝试和讨论的，我们可以像从前一样，不断尝试新的方法，促进 ECMAScript 标准的发展。

生成器(Generator)

终于到了 ES2015 中我最喜欢的特性了，前方高能反应，所有人立刻进入战斗准备！

为什么说这是我最喜欢的新特性呢？对于一个纯前端的 JavaScript 工程师来说，可能 Generator 并没有什么卵用，但若你曾使用过 Node.js 或者你的前端工程中有大量的异步操作，Generator 简直是你的“贤者之石”。（不过，这并不是 Generator 最正统的用法。出于严谨，我会从头开始讲述 Generator）

来龙

Generator 的设计初衷是为了提供一种能够简便地生成一系列对象的方法，如计算斐波那契数列（Fibonacci Sequence）：

```
function* fibo() {
  let a = 1
  let b = 1

  yield a
  yield b

  while (true) {
```

```
    let next = a + b
    a = b
    b = next
    yield next
  }
}

let generator = fibo()

for (var i = 0; i < 10; i++)
  console.log(generator.next().value) //=> 1 1 2 3 5 8 13 21 34 55
```

如果你没有接触过 Generator，你一定会对这段代码感到很奇怪：为什么 `function` 后会有一个 `*`？为什么函数里使用了 `while (true)` 却没有进入死循环而导致死机？`yield` 又是什么鬼？

不著急，我们一一道来。

基本概念

在学习如何使用 Generator 之前，我们先了解一些必要的概念。

Generator Function

生成器函数用于生成生成器(Generator)，它与普通函数的定义方式的区别就在于它需要在 `function` 后加一个 `*` 。

```
function* FunctionName() {
  // ...Generator Body
}
```

生成器函数的声明形式不是必须的，同样可以使用匿名函数的形式。

```
let FunctionName = function*() { /* ... */ }
```

生成器函数的函数内容将会是对应生成器的运行内容，其中支持一种新的语法 `yield`。它的作用与 `return` 有点相似，但并非退出函数，而是**切出生成器运行时**。

你可以把整个生成器运行时看成一条长长的面条（`while (true)` 则就是无限长的），JavaScript 引擎在每一次遇到 `yield` 就要切一刀，而切面所成的“纹路”则是 `yield` 出来的值。

~~好吧这是瑞士卷~~

Generator

生(rui)成(shi)器(juan)在某种意义上可以看做为与 JavaScript 主线程分离的运行时（详细可参考我的另外一篇文章：http://lifemap.in/koa-co-and-coroutine/），它可以随时被 `yield` 切回主线程（生成器不影响主线程）。

每一次生成器运行时被 `yield` 都可以带出一个值，使其回到主线程中；此后，也可以从主线程返回一个值回到生成器运行时中：

```
let inputValue = yield outputValue
```

生成器切出主线程并带出 `outputValue`，主函数经过处理后（可以是异步的），把 `inputValue` 带回生成器中；主线程可以通过 `.next(inputValue)` 方法返回值到生成器运行时中。

基本使用方法

构建生成器函数

使用 Generator 的第一步自然是要构建生成器函数，理清构建思路，比如我需要做一个生成斐波那契数列（俗称兔子数列）的生成器们则需要如何构建循环体呢？如果我需要在主线程不断获得结果，则需要在生成器 中做无限循环，以保证其不断地生成。

而根据斐波那契数列的定义，第 n (n ≥ 3) 项是第 n - 1 项和第 n - 2 之和，而第 1 项和第 2 项都是 1。

```
function* fibo() {
  let [a, b] = [1, 1]

  yield a
  yield b

  while (true) {
    [a, b] = [b, a + b]
    yield b
  }
}
```

这样设计生成器函数，就可以先把预先设定好的首两项输出，然后通过无限循环不断把后一项输出。

启动生成器

生成器函数不能直接用来作为生成器使用，需要先使用这个函数得到一个生成器，用于运行生成器内容和接收返回值。

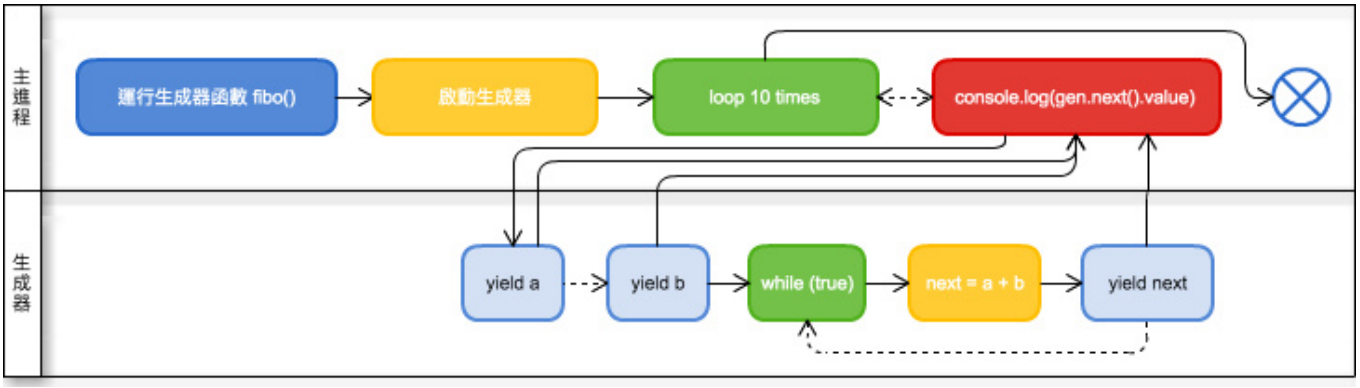
```
let gen = fibo()
```

运行生成器内容

得到生成器以后，我们就可以通过它进行数列项生成了。此处演示获得前 10 项。

```
let arr = []
for (let i = 0; i < 10; i++)
  arr.push(gen.next().value)

console.log(arr) //=> [ 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 ]
```

你也可以通过图示理解 Generator 的运行原理



事实上，Generator 的用法还是很多种，其中最为著名的一种便是使用 Generator 的特性模拟出 ES7 中的 async/await 特性。而其中最为著名的就是 co 和 koa(基于 co 的 Web Framework) 了。详细可以看我的另外一篇文章：Koa, co and coroutine。

原生的模块化

在前文中，我提到了 ES2015 在工程化方面上有著良好的优势，而采用的就是 ES2015 中的原生模块化机制，足以证明它的重要性。

历史小回顾

在 JavaScript 的发展历史上，曾出现过多种模块加载库，如 RequireJS、SeaJS、FIS 等，而由它们衍生出来的 JavaScript 模块化标准有 CommonJS、AMD、CMD 和 UMD 等。

其中最为典型的是 Node.js 所遵循的 CommonJS 和 RequireJS 的 AMD。

本文在此不再详细说明这些模块化方案，详细可以阅读 What Is AMD, CommonJS, and UMD?

基本用法

正如前文所展示的使用方式一样，ES2015 中的模块化机制设计也是相当成熟的。基本上所有的 CommonJS 或是 AMD 代码都可以很快地转换为 ES2015 标准的加载器代码。

```
import name from "module-name"
import * as name from "module-name"
import { member } from "module-name"
import { member as alias } from "module-name"
import { member1 , member2 } from "module-name"
import { member1 , member2 as alias2 , [...] } from "module-name"
import defaultMember, { member [ , [...] ] } from "module-name"
import defaultMember, * as alias from "module-name"
import defaultMember from "module-name"
import "module-name"
// Copy from Mozilla Developer Center
```

如上所示，ES2015 中有很多种模块引入方式，我们可以根据实际需要选择一种使用。

全局引入

全局引入是最基本的引入方式，这跟 CommonJS、AMD 等模块化标准并无两样，都是把目标模块的所有暴露的接口引入到一个命名空间中。

```
import name from 'module-name'
```

```
import * as name from 'module-name'
```

这跟 Node.js 所用的 CommonJS 类似：

```
var name = require('module-name')
```

局部引入

与 CommonJS 等标准不同的是，ES2015 的模块引入机制支持引入模块的部份暴露接口，这在大型的组件开发中显得尤为方便，如 React 的组件引入便是使用了该特性。

```
import { A, B, C } from 'module-name'

A()
B()
C()
```

接口暴露

ES2015 的接口暴露方式比 CommonJS 等标准都要丰富和健壮，可见 ECMA 委员会对这一部份的重视程度之高。

ES2015 的接口暴露有几种用法：

暴露单独接口

```
// module.js
export function method() { /* ... */ }

// main.js
import M from './module'
M.method()
```

基本的 `export` 语句可以调用多次，单独使用可暴露一个对象到该模块外。

暴露复盖模块

若需要实现像 CommonJS 中的 `module.exports = {}` 以覆盖整个模块的暴露对象，则需要在 `export` 语句后加上 `default`。

```
// module.js
export default {
  method1,
  method2
}

// main.js
import M from './module'
M.method1()
```

降级兼容

在实际应用中，我们暂时还需要使用 babel 等工具对代码进行降级兼容。庆幸的是，babel 支持 CommonJS、AMD、UMD 等模块化标准的降级兼容，我们可以根据项目的实际情况选择降级目标。

```
$ babel -m common -d dist/common/ src/
$ babel -m amd -d dist/amd/ src/
$ babel -m umd -d dist/umd/ src/
```

Promise

Promise，作为一个老生常谈的话题，早已被聪明的工程师们“玩坏”了。

光是 Promise 自身，目前就有多种标准，而目前最为流行的是 Promises/A+。而 ES2015 中的 Promise 便是基于 Promises/A+ 制定的。

概念

Promise 是一种用于解决回调函数无限嵌套的工具（当然，这只是其中一种），其字面意义为“保证”。它的作用便是“免去”异步操作的回调函数，保证能通过后续监听而得到返回值，或对错误处理。它能使异步操作变得井然有序，也更好控制。我们以在浏览器中访问一个 API，解析返回的 JSON 数据。

```
fetch('http://example.com/api/users/top')
  .then(res => res.json())
  .then(data => {
    vm.data.topUsers = data
  })
  // Handle the error crash in the chaining processes
  .catch(err => console.error(err))
```

Promise 在设计上具有原子性，即只有两种状态：未开始和结束（无论成功与否都算是结束），这让我们在调用支持 Promise 的异步方法时，逻辑将变得非常简单，这在大规模的软件工程开发中具有良好的健壮性。

基本用法

创建 Promise 对象

要为一个函数赋予 Promise 的能力，先要创建一个 Promise 对象，并将其作为函数值返回。Promise 构造函数要求传入一个函数，并带有 `resolve` 和 `reject` 参数。这是两个用于结束 Promise 等待的函数，对应的**成功**和**失败**。而我们的逻辑代码就在这个函数中进行。

此处，因为必须要让这个函数包裹逻辑代码，所以如果需要用到 `this` 时，则需要使用箭头函数或者在前面做一个 `this` 的别名。

```
function fetchData() {
  return new Promise((resolve, reject) => {
```

```
    // ...
  })
}
```

进行异步操作

事实上，在异步操作内，并不需要对 Promise 对象进行操作（除非有特殊需求）。

```
function fetchData() {
  return new Promise((resolve, reject) => {
    api.call('fetch_data', (err, data) => {
      if (err) return reject(err)

      resolve(data)
    })
  })
}
```

因为在 Promise 定义的过程中，也会出现数层回调嵌套的情况，如果需要使用 `this` 的话，便显现出了箭头函数的优势了。

使用 Promise

让异步操作函数支持 Promise 后，我们就可以享受 Promise 带来的优雅和便捷了~

```
fetchData()
  .then(data => {
    // ...

    return storeInFileSystem(data)
  })
  .then(data => {
    return renderUIAnimated(data)
  })
```

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

弊端

虽说 Promise 确实很优雅，但是这是在所有需要用到的异步方法都支持 Promise 且遵循标准。而且链式 Promise 强制性要求逻辑必须是线性单向的，一旦出现如并行、回溯等情况，Promise 便显得十分累赘。

所以在目前的最佳实践中，Promise 会作为一种接口定义方法，而不是逻辑处理工具。后文将会详细阐述这种最佳实践。

Symbol

Symbol 是一种很有意思的概念，它跟 Swift 中的 Selector 有点相像，但也更特别。在 JavaScript 中，对象的属性名称可以是字符串或数字。而如今又多了一个 Symbol。那 Symbol 究竟有什么用？

首先，我们要了解的是，Symbol 对象是具有唯一性的，也就是说，每一个 Symbol 对象都是唯一的，即便我们看不到它的区别在哪里。这就意味著，我们可以用它来保证一些数据的安全性。

```
console.log(Symbol('key') == Symbol('key')) //=> false
```

如果将一个 Symbol 隐藏于一个封闭的作用域内，并作为一个对象中某属性的键，则外层作用域中便无法取得该属性的值，有效保障了某些私有库的代码安全性。

```
let privateDataStore = {
  set(val) {
    let key = Symbol(Math.random().toString(32).substr(2))
    this[key] = val

    return key
```

```
  },

  get(key) {
    return this[key]
  }
}

let key = privateDateStore('hello world')
privateDataStore[key] //=> undefined
privateDataStore.get(key) //=> hello world
```

如果你想通过某些办法取得被隐藏的 key 的话，我只能说：理论上，不可能。

```
let obj = {}
let key = Symbol('key')

obj[key] = 1
JSON.stringify(obj) //=> {}
Object.keys(obj) //=> []

obj[key] //=> 1
```

黑科技

Symbol 除了带给我们数据安全性以外，还带来了一些很神奇的黑科技，简直了。

`Symbol.iterator`

除 Symbol 以外，ES2015 还为 JavaScript 带来了 `for...of` 语句，这个跟原本的 `for...in` 又有什么区别？

我们还是以前面的斐波那契数列作为例子。Iterator 在 Java 中经常用到中会经常用到，意为“迭代器”，你可以把它理解为用于循环的工具。

```
let fibo = {
  [ Symbol.iterator ]() {
    let a = 0
    let b = 1
    return {
      next() {
        [a, b] = [b, a + b]
        return { done: false, value: b }
      }
    }
  }
}

for (let n of fibo) {
  if (n > 100) break
  console.log(n)
}
```

Wow! 看到这个 `for…of` 是否有种兴奋的感觉？虽然说创建 `fibo` 的时候稍微有点麻烦……

不如我们先来看看这个 `fibo` 究竟是怎么定义出来了。首先，我们要了解到 JavaScript 引擎(或编译器)在处理 `for...of` 的时候，会从 `of` 后的对象取得 `Symbol.iterator` 这属性键的值，为一个函数。它要求要返回一个包含 `next` 方法的对象，用于不断迭代。而因为 `Symbol.iterator` 所在键值对的值是一个函数，这就让我们有了自由发挥的空间，比如定义局部变量等等。

每当 `for...of` 进行了一次循环，都会执行一次该对象的 `next` 方法，已得到下一个值，并检查是否迭代完成。随著 ES7 的开发，`for...of` 所能发挥的潜能将会越来越强。

还有更多的 Symbol 黑科技等待挖掘，再次本文不作详细阐述，如有兴趣，可以看看 Mozilla Developer Center 上的介绍。

Proxy(代理)

Proxy 是 ECMAScript 中的一种新概念，它有很多好玩的用途，从基本的作用说就是：Proxy 可以在不入侵目标对象的情况下，对逻辑行为进行拦截和处理。

比如说我想记录下我代码中某些接口的使用情况，以供数据分析所用，但是因为目标代码中是严格控制的，所以不能对其进行修改，而另外写一个对象来对目标对象做代理也很麻烦。那么 Proxy 便可以提供一种比较简单的方法来实现这一需求。

假设我要对 `api` 这一对象进行拦截并记录下代码行为，我就可以这样做：

```
let apiProxy = new Proxy(api, {
  get(receiver, name) {
    return (function(...args) {
      min.sadd(`log:${name}`, args)
      return receiver[name].apply(receiver, args)
    }).bind(receiver)
  }
})

api.getComments(artical.id)
  .then(comments => {
    // ...
  })
```

可惜的是，目前 Proxy 的兼容性很差，哪怕是降级兼容也难以实现。

到这里，相信你已经对 ES2015 中的大部份新特性有所了解了。那么现在，就结合我们原有的 JavaScript 技能，开始使用 ES2015 构建一个具有工程化特点的项目吧。

ES2015 的前端开发实战

事实上，你们都应该有听说过 React 这个来自 Facebook 的前端框架，因为现在它实在太火了。React 与 ES2015 的关系可谓深厚，React 在开发上便要求使用 ES2015 标准，因

其 DSL —— JSX 的存在，所以必须要依赖 Babel 将其编译成 JavaScript。

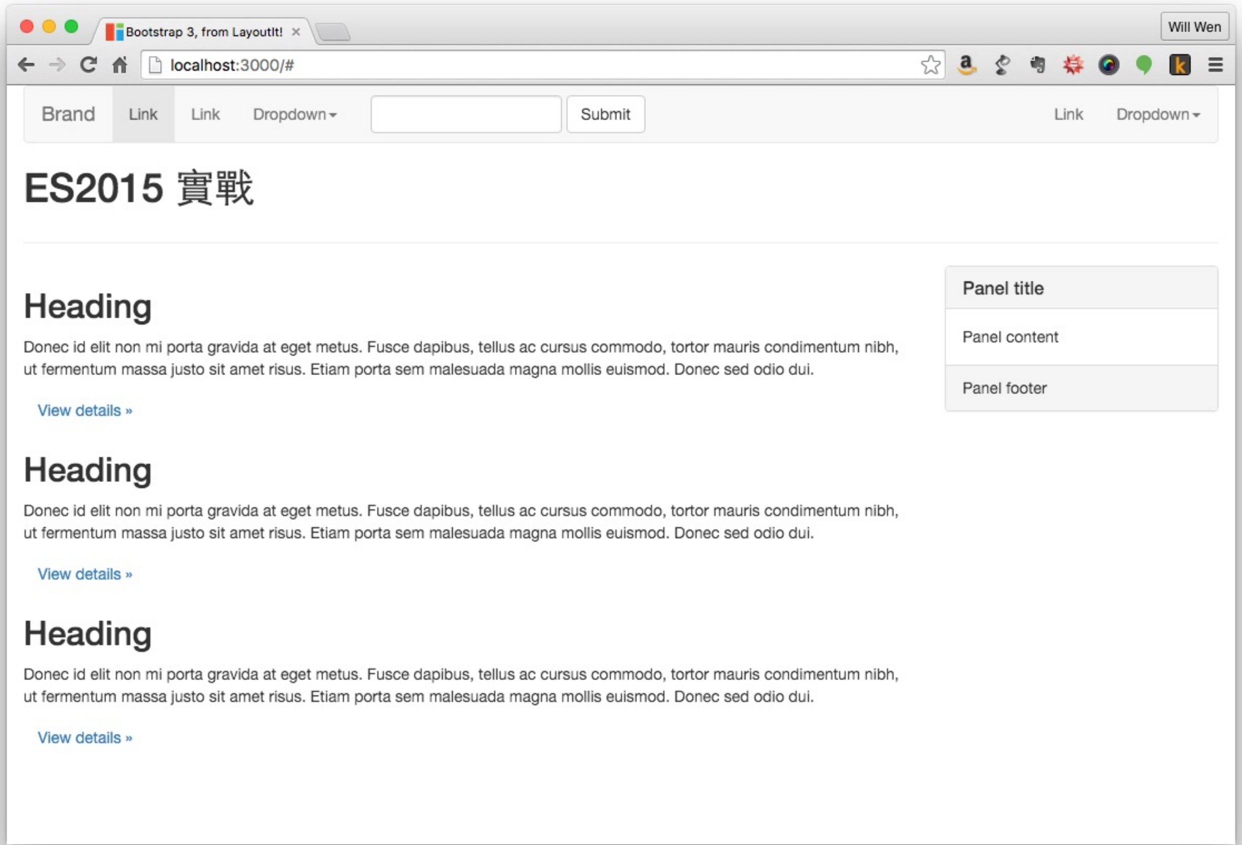
但同样是由于 JSX 的存在，本文章并不会采用 React 作为前端框架，以避免读者对 JSX 和 HTML 的误解。我们会采用同样优秀的前端 MVVM 框架 —— Vue 进行开发。

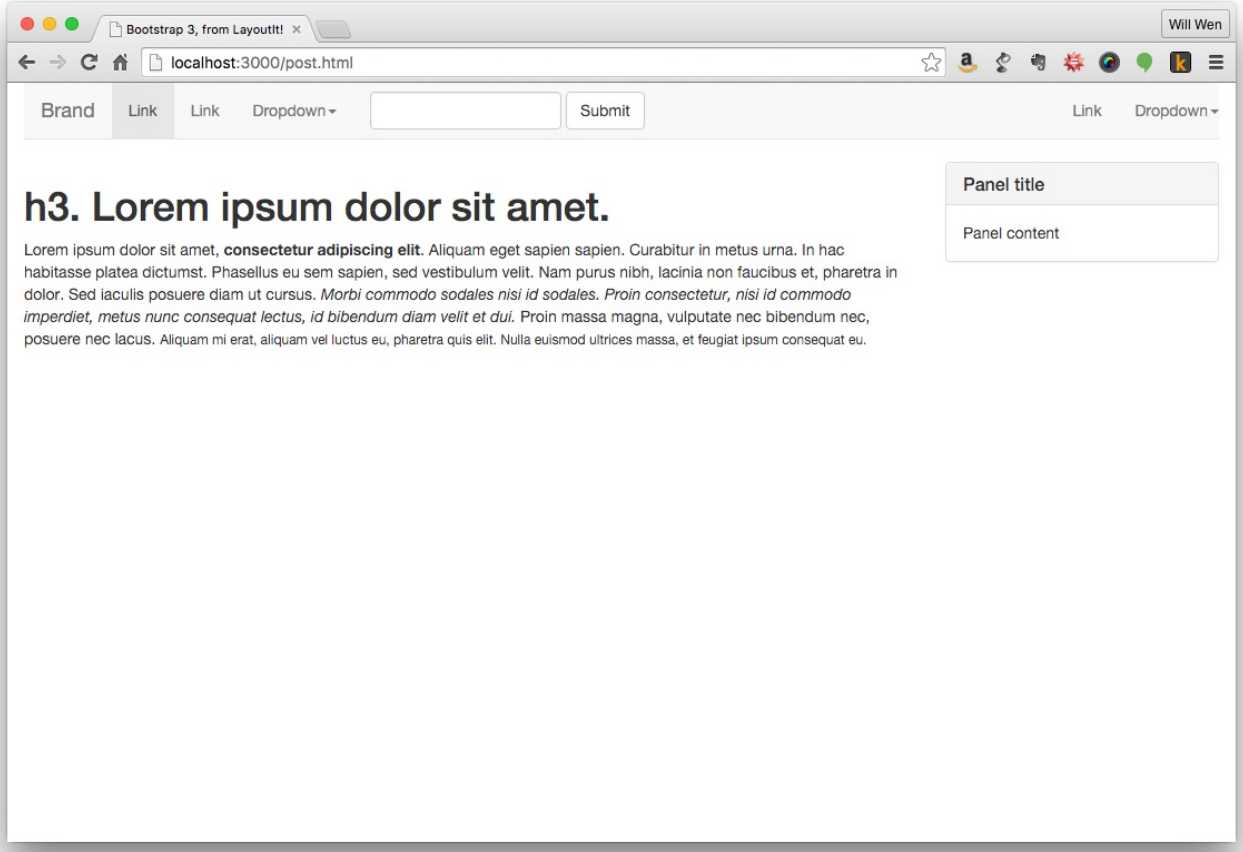
数据部份，将会使用 MinDB 进行存储和处理。MinDB 是由笔者开发的一个用于 JavaScript 环境的简易而健壮的数据库，它默认使用 DOM Storage 作为其存储容器，在其他环境中可以通过更换 Store Interface 以兼容绝大部份 JavaScript 运行环境。

Vue.js 的使用教程可以参考 Vue.js 的官方教程。

构建界面

我们首先简单地用 LayoutIt 搭建一个用 Bootstrap 构架的页面，其中包含了 DEMO 的首页和文章内容页，此后我们将会使用这个模板搭建我们的 JavaScript 代码架构。





接下来，我们需要通过对页面的功能块进行组件划分，以便于使用组件化的架构搭建前端页面。

我们可以大致分为 Index、Post 和 Publish 三个页面，也可以说三个路由方向；而我们还可以把页面中的组块分为：文章列表、文章、侧边栏、评论框等。

以此，我们可以设计出以下结构，以作为这个项目的组织结构：

```
Routes Components
 |- Index ----|- Posts
 |  |- Sidebar
 |
 |- Post -----|- Post
 |  |- Comments
 |
 |- Publish
```

首页包含了文章列表、侧边栏两个组件；文章页面包含文章内容组件和评论框组件（此处我们使用多说评论框作为我们的组件）；而文章发布页则可以单独为一个路由器，而不需要分出组件。

代码结构定义

因我们是以 babel 进行 ES2015 降级兼容的，所以我们最好可以采用分离的结构，这里我们使用 `src` 和 `dist`。

我们此处以比较简单的结构构建我们的DEMO：

```
app
 |- src 程序的源文件目录
 | |- controllers 后端的路由处理器
 | |- lib 后端需要引用的一些库
 | |- public 前端 JavaScript 源文件
 | | |- controllers 前端的路由处理器
 | | |- components 前端组件
 | | |- models 前端数据层
 | | |- config.js 前端的配置文件
 | | |- main.js 前端 JavaScript 入口
 | |- app.js 后端程序入口
 | |- routes.js 后端路由表
 |- dist 降级兼容输出目录
 | |- public
 | |- css
 | |- index.html 前端 HTML 入口
 |- gulpfile.js Gulp 构建配置文件
 |- package.json Node.js 项目配置文件
```

而我们在这一章节中则专注于 `public` 这一目录即可，Node.js 部份将在下一章节详细展示。

架构设计

模块化

因为有了 ES2015 自身的模块化机制，我们就不必使用 RequireJS 等模块加载库了，通过 Browserify 我们我可以将整个前端的 JavaScript 程序打包到一个 .js 文件中，而这一步骤我们使用 Gulp 来完成。

详细的 Gulp 使用教程可以参考：了不起的任务运行器Gulp基础教程

数据支持

我们的数据将从后端的 Node.js 程序中获取，以 API 的形式获得。

另外，为了得到更佳的用户体验，我们将使用 MinDB 作为这个 DEMO 的前端数据库，减少网络请求次数，优化使用体验。

界面渲染

为了能让我们的界面定义能够足够简单，我们在这使用了 Vue 作为前端开发框架，将其与 MinDB 对接，负责渲染我们的页面。

其中，我们会利用 Vue 的组件系统来实现我们制定下的组件设计。另外，我们还会使用 watchman.js 来实现前端的路由器，以便我们对不同的页面的逻辑进行分离。

构建应用

在开始编写业务代码之前，我们需要先安装好我们所需要的依赖库。

```
$ npm install vue min watchman-router marked --save
```

安装好依赖库以后，我们就开始编写入口文件吧！

入口文件

```
// main.js
import Vue from 'vue'
import watch from 'watchman-router'
import qs from 'querystring' // From Node.js

watch({
  // TODO
})
  .use((ctx, next) => {
    ctx.query = qs.parse(window.location.search.substr(1))
    next()
  })
  .run()
```

在入口中，我们将做如下任务：

1. 引入所有的路由相应器
2. 将路由相应器通过 watchman.js 绑定到对应的 url 规则中
3. 建立在共用模板中存在的需要实例化的组件

因为我们还么有开始动工路由相应器，所以我们先把“建立在共用模板中存在的需要实例化的组件”这一任务完成。

在共用模板中，有唯一一个必须的共用组件就是页面切换器 —— 一个用于包含所有的页面的元素。

```
<div class="row" id="wrapper" v-html="html"></div>
```

对应的，我们将在入口文件中使用 Vue 建立其对应的 ViewModel，并将其绑定至 DOM 元素。

```
let layoutVM = new Vue({
```

```
  el: '#wrapper',
  data: {
    html: ''
  }
})
```

以后可以通过改变 `layoutVM.$data.html` 来改变 `#wrapper` 的内容，配合 watchman.js 以加载不同的页面内容。

为了能在路由相应器中改变 `layoutVM` 的参数，我们将其作为 watchman.js 给予相应器的上下文参数中的一个属性。

```
// ...
  .use((ctx, next) => {
    ctx.query = qs.parse(window.location.search.substr(1))
    ctx.layoutVM = layoutVM
    next()
  })
```

数据层：文章

我们单独把文章的查询、读取和创建等操作抽象成一个库，使其与逻辑层分离，让代码更美观。

在此之前，我们先定义好从后端用于取得文章数据的 API：

1. URL: `/api/posts/list`
2. 参数:
    - `page` 当前页数，每页 10 条记录

我们可以直接用新的 Ajax API 来进行 API 访问。

```
import 'whatwg-fetch'
import min from 'min'

async function listPosts(page = 0) {
  const count = 10

  // 检查 MinDB 是否存在数据
  let existsInMinDB = await min.exists('posts:id')

  if (!existsInMinDB) {
    var posts = (await _fetchPost(page))
      .map(post => {
        return {
          id: post._id,
          title: post.title,
          content: post.content,
          author: post.author,
          comments: post.comments.length,
          get summary() {
            return post.content.substr(0, 20) + '...'
          }
        }
      })

    // 将数据存储到 MinDB 中
    for (let i = 0; i < posts.length; i++) {
      let post = posts[i]

      await min.sadd('posts:id', post.id)
      await min.hmset(`post:${post.id}`, post)
    }
  } else {
    // 从 MinDB 读取数据
    let ids = await min.smembers('posts:id')
```

```
    ids = ids.slice(page * count, (page + 1) * count)
    var posts = await min.mget(ids.map(id => `post:${id}`))
  }

  return posts
}

async function _fetchPost(page) {
  // 通过 `fetch` 访问 API
  let res = await fetch(`/api/posts/list?page=${page}`)
  let reply = await res.json()

  return reply.posts
}
```

其中，`min.sadd('posts:id', post.id)` 会将文章的 ID 存入 MinDB 中名为 `posts:id` 的集中，这个集用于保存所有文章的 ID；`min.hmset(`post:${post.id}`, post)` 则会将文章的所有数据存入以 `post:{id}` 命名的 Hash 中。

完成对首页的所有文章列表支持后，我们还需要简单的定义一个用于读取单篇文章数据的 API。

如果用户是从首页的文章列表进来的，那么我们就可以直接从 MinDB 中读取文章数据。但如果用户是直接通过 url 打开网址的话，MinDB 有可能并没有存储文章数据，那么我们就通过 API 从后端获取数据，并存储在 MinDB 中。

- URL：`/api/posts/:id`

```
async function getPost(id) {
  let existsInMinDB = await min.exists(`post:${id}`)

  if (existsInMinDB) {
    return await min.hgetall(`post:${id}`)
```

```
  } else {
    let res = await fetch(`/api/posts/${id}`)
    let post = (await res.json()).post

    await min.hmset(`post:${id}`, {
      id: post._id,
      title: post.title,
      content: post.content,
      author: post.author,
      comments: post.comments.length,
      get summary() {
        return post.content.substr(0, 20) + '...'
      }
    })

    return post
  }
}
```

完成用于读取的接口后，我们也该做做用于写入的接口了。同样的，我们也先来把 API 定义一下。

1. URL：`/api/posts/new`
2. Body（JSON）：
    - title
    - content
    - author

我们也可以通过 `fetch` 来发出 POST 请求。

```
async function publishPost(post) {
  let res = await fetch('/api/posts/new', {
    method: 'POST',
```

```
    headers: {
      'Accept': 'application/json',
      'Content-Type': 'application/json'
    },
    body: JSON.stringify(post)
  })
  var _post = await res.json()

  await min.sadd('posts:id', _post._id)
  await min.hmset(`post:${_post._id}`, {
    id: _post._id,
    title: _post.title,
    content: _post.content,
    author: _post.author,
    comments: 0,
    get summary() {
      return _post.title.substr(0, 20) + '...'
    }
  })

  _post.id = _post._id

  return _post
}
```

最后我们就可以暴露出这几个接口了。

```
export default {
  listPosts,
  getPost,
  publishPost
}
```

路由：首页

首先我们确定一下首页中我们需要做些什么：

1. 改变 `layoutVM` 的 HTML 数据，为后面的页面渲染做准备
2. 分别从后端加载文章数据和多说加载评论数，加载完以后存入 MinDB 中
3. 从 MinDB 中加载已缓存的数据
4. 建立对应的 VM，并传入数据

准备页面渲染

首先，我们需要为 watchman.js 的路由提供一个函数以作相应器，并包含一个为 `context` 的参数。我们则可以将其作为模块的暴露值。

```
export default function(ctx) {
  // 改变 layoutVM
  ctx.layoutVM.$data.html = `
    <h1>
      ES2015 实战 - DEMO
    </h1>
    <hr>
    <div id="posts" class="col-md-9">
      <post-in-list v-repeat="posts"></post-in-list>
    </div>
    <div id="sidebar" class="col-md-3">
      <panel title="侧边栏" content="{{content}}" list="{{list}}"></panel>
    </div>
  `
}
```

此处将首页的 HTML 结构赋予 `layoutVM`，并让其渲染至页面中。

加载数据

因为我们之前已经将数据层抽象化了，所以我们此处只需通过我们的抽象数据层读取我们

所需要的数据即可。

```
import Vue from 'vue'

import Posts from '../models/posts'

// ...
export default async function(ctx) {
  let refresh = 'undefined' != typeof ctx.query.refresh
  let page = ctx.query.page || 0

  let posts = await Posts.listPosts(page)
}
```

设计组件

在首页的 HTML 中，我们用到了两个 Component，分别为 `post-in-list` 和 `panel`，我们分别在 `components` 文件夹中分别建立 `post-in-list.js` 和 `panel.js`。我们从我们之前通过 LayoutIt 简单建立的 HTML 模板中，抽出对应的部份，并将其作为 Vue Component 的模板。

```
// post-in-list.js
import Vue from 'vue'
import marked from 'marked'

// 模板
const template = `
  <div class="post" v-attr="id: id">
    <h2><a href="/#!/post/{{id}}" v-text="title"></a></h2>
    <p v-text="summary"></p>
    <p>
      <small>由 {{author}} 发表</small> | <a class="btn" href="/#!/post/{{id}
    </p>
```

```
  </div>
`
```

我们可以通过 Vue 的双向绑定机制将数据插入到模板中。

```
Vue.component('post-in-list', {
  template: template,
  replace: true
})
```

根据 Vue 的组件机制，我们可以通过对组件标签中加入自定义属性来传入参数，以供组件中使用。

但此处我们先用 Vue 的 `v-repeat` 指令来进行循环使用组件。

```
<post-in-list v-repeat="posts"></post-in-list>
```

`panel.js` 同理建立。

路由：文章页面

相比首页，文章页面要简单得多。因为我们在首页已经将数据加载到 MinDB 中了，所以我们可以直接从 MinDB 中读取数据，然后将其渲染到页面中。

```
// ...
let post = await min.hgetall(`post:${this.id}`)
// ...
```

然后，我们再从之前设计好的页面模板中，抽出我们需要用来作为文章页面的内容页。

```
<h1 v-text="title"></h1>
```

```
<small>由 {{author}} 发表</small>
<div class="post" v-html="content | marked"></div>
```

我们同样是同样通过对 `layoutVM` 的操作，来准备页面的渲染。在完成渲染准备后，我们就可以开始获取数据了。

```
let post = await min.hgetall(`post:${this.id}`)
```

在获得相应的文章数据以后，我们就可以通过建立一个组件来将其渲染至页面中。其中，要注意的是我们需要通过 Vue 的一些 API 来整合数据、渲染等步骤。

在这我不再详细说明其构建步骤，与上一小节相同。

```
import Vue from 'vue'
import min from 'min'
import marked from 'marked'

const template = `
  <h1 v-text="title"></h1>
  <small>由 {{author}} 发表</small>
  <div class="post" v-html="content | marked"></div>
`

let postVm = Vue.component('post', {
  template: template,
  replace: true,
  props: [ 'id' ],

  data() {
    return {
      id: this.id,
      content: '',
```

```
      title: ''
    }
  },

  async created() {
    this.$data = await min.hgetall(`post:${this.id}`)
  },

  filters: {
    marked
  }
})

export default postVm
```

此处我们除了 ES2015 的特性外，我们还更超前地使用了正在制定中的 ES7 的特性，比如 `async/await`，这是一种用于对异步操作进行“打扁”的特性，它可以把异步操作以同步的语法编写。如上文所说，在 ES2015 中，我们可以用 co 来模拟 `async/await` 特性。

路由：发布新文章

在发布新文章的页面中，我们直接调用我们之前建立好的数据抽象层的接口，将新数据传向后端，并保存在 MinDB 中。

```
import marked from 'marked'

import Posts from '../models/posts'

// ...
new Vue({
  el: '#new-post',

  data: {
```

```
    title: '',
    content: '',
    author: ''
  },

  methods: {
    async submit(e) {
      e.preventDefault()

      var post = await Posts.publishPost({
        title: this.$data.title,
        content: this.$data.content,
        author: this.$data.author
      })

      window.location.hash = `#!/post/${post.id}`
    }
  },

  filters: {
    marked
  }
})
```

路由绑定

在完成路由响应器的开发后，我们就可以把他们都绑定到 watchman.js 上了。

```
// ...

import Index from './controllers/index'
import Post from './controllers/post'
import Publish from './controllers/publish'
```

```
watch({
  '/': Index,
  '#!/': Index,
  '#!/post/:id': Post,
  '#!/new': Publish
})

// ...
```

这样，就可以让我们之前的路由结构都绑定到入口文件中：

1. 首页绑定到 `/` 和 `#!/`
2. 文章页面则绑定到了 `#!/post/:id`，比如 `#!/post/123` 则表示 id 为 123 的文章页面
3. `#!/new` 则绑定了新建文章的页面

合并代码

在完成三个我们所构建的路由设计后，我们就可以用 Browserify 把我们的代码打包到一个文件中，以作为整个项目的入口文件。此处，我们再引入 Gulp 作为我们的构建辅助器，而不需要直接使用 Browserify 的命令行进行构建。

在开始编写 Gulpfile 之前，我们先安装我们所需要的依赖库：

```
$ npm install gulp browserify babelify vinyl-source-stream vinyl-buffer babel
```

将依赖库安装好以后，我们就可以开始编写 Gulp 的配置文件了。

在本文将近完成的时候，Babel 发布了版本 6，其 API 与 5 版本有著相当大的区别，且 Babel 6 并不向前兼容，详细更改此处不作介绍。

```
var gulp = require('gulp')
```

```
var browserify = require('browserify')
var babelify = require('babelify')
var source = require('vinyl-source-stream')
var buffer = require('vinyl-buffer')

gulp.task('browserify', function() {
  return browserify({
    entries: ['./src/public/main.js']
  })
    .transform(babelify.configure({
      presets: [ 'es2015-without-regenerator' ],
      plugins: [ 'transform-async-to-generator' ]
    }))
    .bundle()
    .pipe(source('bundle.js'))
    .pipe(buffer())
    .pipe(gulp.dest('dist/public'))
})

gulp.task('default', [ 'browserify' ])
```

在这个配置文件中，我们把前端的代码中的入口文件传入 babel 中，然后将其打包成 bundle.js。

最后，我们可以在我们最开始通过 Layoutit 所设计的页面中，把可以用于包含替换内容的部份去掉，然后引入我们通过 Browserify 生成的 bundle.js。

完成 JavaScript 部份的开发后，我们再将所需要的静态资源文件加载到 HTML 中，我们就可以看到这个基本脱离后端的前端 Web App 的效果了。

ES2015 的 Node.js 开发实战

就目前来说，能最痛快地使用 ES2015 中各种新特性进行 JavaScript 开发的环境，无疑就是 Node.js。就 Node.js 本身来说，就跟前端的 JavaScript 环境有著本质上的区别，Node.js 有著完整意义上的异步 IO 机制， 有著无穷无尽的应用领域，而且在语法角度上遇到问题机率比在前端大不少。甚至可以说，Node.js 一直是等著 ES2015 的到来的，Node.js 加上 ES2015 简直就是如虎添翼了。

从 V8 引擎开始实验性的开始兼容 ES6 代码时，Node.js 便开始马上跟进，在 Node.js 中开放 ES6 的兼容选项，如 Generator、Classes 等等。经过相当长一段时间的测试和磨合后，就以在 Node.js 上使用 ES6 标准进行应用开发这件事来说，已经变得越来越成熟，越来越多的开发者走上了 ES6 这条“不归路”。

一些针对 Node.js + ES6 的开发模式和第三方库也如雨后春笋般冒出，其中最为人所熟知的便是以 co 为基础所建立的 Web 框架 Koa。Koa 由 TJ 等 express 原班人马打造，目前

也有越来越多的中国开发者加入到 Koa 的开发团队中来，为前沿 Node.js 的开发做贡献。

co 通过使用 ES2015 中的 Generator 特性来模拟 ES7 中相当诱人的 `async/await` 特性，可以让复杂的异步方法调用及处理变得像同步操作一样简单。引用响马大叔在他所维护的某项目中的一句话：

> 用同步代码抒发异步情怀

在本章节中，我将以一个简单的后端架构体系，来介绍 ES2015 在 Node.js 开发中的基于 Koa 的一种优秀实践方式。

因为 Node.js 自带模块机制，所以用 babel 对 ES2015 的模块语法做降级兼容的时候，只需降至 Node.js 所使用的 CommonJS 标准即可。

不一样的是，ES2015 的模块语法是一种声明式语法，根据 ES2015 中的规定，模块引入和暴露都需要在当前文件中的最顶层，而不能像 CommonJS 中的 `require()` 那样可以在任何地方使用；使用 `import` 引入的模块所在命名空间将会是一个常量，在 babel 的降级兼容中会进行代码检查，保证模块命名空间的安全性。

架构设计

因为我们这个 DEMO 的数据结构并不复杂，所以我们可以直接使用 MongoDB 作为我们的后端数据 库，用于存储我们的文章数据。我们可以通过 `monk` 库作为我们读取、操作 MongoDB 的客户端库。在架构上，Koa 将作为 Web 开发框架，配合 co 等库实现全“同步”的代码编写方式。

构建应用

这就让我们一步一步来吧，创建 Node.js 应用并安装依赖。

```
$ npm init
$ npm install koa koa-middlewares koa-static monk co-monk thunkify --save
```

在上一个章节中，我们已经建立了整个 DEMO 的文件结构，在此我们再展示一遍：

```
app
 |- src 程序的源文件目录
 | |- controllers 后端的路由处理器
 | |- models 数据抽象层
 | |- lib 后端需要引用的一些库
 | |- public 前端 JavaScript 源文件
 | |- app.js 后端程序入口
 | |- routes.js 后端路由表
 |- dist 降级兼容输出目录
 |- gulpfile.js Gulp 构建配置文件
 |- package.json Node.js 项目配置文件
```

在 `src/controllers` 中，包含我们用来相应请求的控制器文件；`src/lib` 文件夹则包含了我们需要的一些抽象库；`src/public` 文件则包含了在上一章节中我们建立的前端应用程序；`src/app.js` 是该应用入口文件的源文件；`src/routes.js` 则是应用的路由表文件。

入口文件

在入口文件中，我们需要完成几件事情：

1. 创建 Koa 应用，并监听指定端口
2. 将所需要的 Koa 中间件接入我们所创建的 Koa 应用中，如静态资源服务
3. 引入路由表文件，将路由控制器接入我们所建立的 Koa 文件中

```
import koa from 'koa'
import path from 'path'
import { bodyParser } from 'koa-middlewares'
import Static from 'koa-static'


import router from './routes'
```

```
let app = koa()
// Static
app.use(Static(path.resolve(__dirname, './public')))
// Parse the body in POST requests
app.use(bodyParser())
// Router
app.use(router.routes())

let PORT = parseInt(process.env.PORT || 3000)
app.listen(PORT, () => {
  console.log(`Demo is running, port:${PORT}`)
})
```

数据抽象层

为了方便我们在 co 的环境中使用 MongoDB，我们选择了 `monk` 和 `co-monk` 两个库进行组合，并抽象出链接库。

```
// lib/mongo.js
import monk from 'monk'
import wrap from 'co-monk'

import config from '../../config.json'

const db = monk(config.dbs.mongo)

/**
 * 返回 MongoDB 中的 Collection 实例
 *
 * @param {String} name collection name
 * @return {Object} Collection
 */
function collection(name) {
```

```
  return wrap(db.get(name))
}

export default {
  collection
}
```

通过这个抽象库，我们就可以避免每次获取 MongoDB 中的 Collection 实例时都需要连接一遍数据库了。

```
// models/posts.js
import mongo from '../lib/mongo'

export default mongo.collection('posts')
```

Posts 控制器

完成了数据层的抽象处理后，我们就可以将其用于我们的控制器了，参考 `monk` 的文档，我们可以对这个 Posts Collection 进行我们所需要的操作。

```
import thunkify from 'thunkify'
import request from 'request'

import Posts from '../models/posts'

const requestAsync = thunkify((opts, callback) => {
  request(opts, (err, res, body) => callback(err, body))
})
```

此处我们使用 `thunkify` 对 `request` 做了一点小小的封装工作，而因为 `request` 库自身的 callback 并不是标准的 callback 形式，所以我们并不能直接把 `request` 函数传入

`thunkify` 中，我们需要的是 callback 中的第三个参数 `body`，所以我们需要自行包装一层函数以取得 `body` 并返回到 co 中。

API：获取所有文章

我们可以通过这个 API 获取存储在 MongoDB 中的所有文章，并支持分页。支持提供当前获取的页数，每页10篇文章，提供每篇文章的标题、作者、文章内容和评论。

```
// GET /api/posts/list?page=0
router.get.listPosts = function*() {
  let page = parseInt(this.query.page || 0)
  const count = 10

  let posts = yield Posts.find({}, {
    skip: page * count,
    limit: count
  })

  // 从多说获取评论
  posts = yield posts.map(post => {
    return function*() {
      let duoshuoReply = JSON.parse(yield requestAsync(`http://api.duoshuo.c

      var commentsId = Object.keys(duoshuoReply.parentPosts)
      post.comments = commentsId.map(id => duoshuoReply.parentPosts[id])

      return post
    }
  })

  // 返回结果
  this.body = {
    posts: posts
  }
}
```

此处我们用到了 co 的一个很有意思的特性，并行处理异步操作。我们通过对从 MongoDB 中取得数据进行 `#map()` 方法的操作，返回一组 Generator Functions，并将这个数组传给 `yield`，co 便可以将这些 Generator Functions 全部执行，并统一返回结果。同样的我们还可以使用对象来进行类似的操作：

```
co(function*() {
  let result = yield {
    posts: getPostsAsync(),
    hot: getHotPosts(),
    latestComments: getComments(10)
  }

  result //=> { posts: [...], hot: [...], latestComments: [...] }
})
```

API：获取指定文章

这个 API 用于通过提供指定文章的 ID，返回文章的数据。

```
// GET /api/posts/:id
router.get.getPost = function*() {
  let id = this.params.id

  let post = yield Posts.findById(id)

  let duoshuoReply = JSON.parse(yield requestAsync(`http://api.duoshuo.com/t

  var commentsId = Object.keys(duoshuoReply.parentPosts)
  post.comments = commentsId.map(id => duoshuoReply.parentPosts[id])

  this.body = {
    post: post
```

```
  }
}
```

API：发布新文章

相比上面两个 API，发布新文章的 API 在逻辑上则要简单得多，只需要向 Collection 内插入新元素，然后将得到的文档返回至客户端既可以。

```
// POST /api/posts/new
router.post.newPost = function*() {
  let data = this.request.body

  let post = yield posts.insert(data)

  this.body = {
    post: post
  }
}
```

Comments 控制器

除了文章的 API 以外，我们还需要提供文章评论的 API，以方便日后该 DEMO 向移动端扩展和弥补多说评论框在原生移动端上的不足。由于评论的数据并不是存储在项目数据库当中，所以我们也不需要为它创建一个数据抽象层文件，而是直接从控制器入手。

API：获取指定文章的评论

```
// GET /api/comments/post/:id
router.get.fetchCommentsInPost = function*() {
  let postId = this.params.id

  let duoshuoReply = JSON.parse(yield requestAsync(`http://api.duoshuo.com/t
```

```
  let commentsId = Object.keys(duoshuoReply.parentPosts)
  let comments = commentsId.map(id => duoshuoReply.parentPosts[id])

  this.body = {
    comments: comments
  }
}
```

API：发表新评论

同样是为了扩展系统的 API，我们通过多说的 API，允许使用 API 来向文章发表评论。

```
// POST /api/comments/post
router.post.postComment = function*() {
  let postId = this.request.body.post_id
  let message = this.request.body.message

  let reply = yield requestAsync({
    method: 'POST',
    url: `http://api.duoshuo.com/posts/create.json`,
    json: true,

    body: {
      short_name: duoshuo.short_name,
      secret: duoshuo.secret,
      thread_key: postId,
      message: message
    }
  })

  this.body = {
    comment: reply.response
  }
}
```

配置路由

完成控制器的开发后，我们是时候把路由器跟控制器都连接起来了，我们在 `src/routes.js` 中会将所有控制器都绑定到路由上，成为一个类似于路由表的文件。

```
import { router as Router } from 'koa-middlewares'
```

首先，我们要将所有的控制器引入到路由文件中来。

```
import posts from './controllers/posts'
import comments from './controllers/comments'
```

然后，创建一个路由器实例，并将所有控制器的响应器和 URL 规则一一绑定。

```
let router = new Router()

// Posts
router.get('/api/posts/list', posts.get.listPosts)
router.get('/api/posts/:id', posts.get.getPost)
router.post('/api/posts/new', posts.post.newPost)

// Comments
router.get('/api/comments/post/:id', comments.get.fetchCommentsInPost)
router.post('/api/comments/post', comments.post.postComment)

export default router
```

配置任务文件

经过对数据抽象层、逻辑控制器、路由器的开发后，我们便可以将所有代码利用 Gulp 进

行代码构建了。

我们先安装好所需要的依赖库。

```
$ npm install gulp gulp-babel vinyl-buffer vinyl-source-stream babelify brows
$ touch gulpfile.js
```

很可惜的是，Gulp 原生并不支持 ES2015 标准的代码，所以在此我们也只能通过 ES5 标准的代码编写任务文件了。

```
'use strict'

var gulp = require('gulp')
var browserify = require('browserify')
var babel = require('gulp-babel')
var babelify = require('babelify')
var source = require('vinyl-source-stream')
var buffer = require('vinyl-buffer')
var babel = require('gulp-babel')
```

我们主要需要完成两个构建任务：

1. 编译并构建前端 JavaScript 文件
2. 编译后端 JavaScript 文件

在构建前端 JavaScript 文件的过程中，我们需要利用 Browserify 配合 babelify 进行代码编译和合并。

```
gulp.task('browserify', function() {
  return browserify({
    cache: {},
    packageCache: {},
```

```
    entries: ['./src/public/main.js']
  })
    .transform(babelify.configure({
      presets: [ 'es2015-without-regenerator' ],
      plugins: [ 'transform-async-to-generator' ]
    }))
    .bundle()
    .pipe(source('bundle.js'))
    .pipe(buffer())
    .pipe(gulp.dest('dist/public'))
})
```

我们将代码编译好以后，便将其复制到 `dist/public` 文件夹中，这也是我们 Node.js 后端处理静态资源请求的响应地址。

而构建后端代码则更为简单，因为我们只需要将其编译并复制到 `dist` 文件夹即可。

```
gulp.task('babel-complie', function() {
  return gulp.src('src/**/*.js')
    .pipe(babel({
      presets: [ 'es2015-without-regenerator' ],
      plugins: [ 'transform-async-to-generator' ]
    }))
    .pipe(gulp.dest('dist/'))
})
```

好了，在完成 gulpfile.js 文件的编写以后，我们就可以进行代码构建了。

```
$ gulp
[07:30:27] Using gulpfile ~/path/to/app/gulpfile.js
[07:30:27] Starting 'babel-complie'...
[07:30:27] Starting 'browserify'...
```

```
[07:30:29] Finished 'babel-complie' after 2.35 s
[07:30:30] Finished 'browserify' after 3.28 s
[07:30:30] Starting 'default'...
[07:30:30] Finished 'default' after 29 µs
```

最后，我们就可以利用 `dist` 文件夹中已经编译好的代码运行起来了。

```
$ node dist/app.js
Demo is running, port:3000
```

不出意外，我们就可以看到我们想要的效果了。

前方高能反应

部署到 DaoCloud

完成了代码的开发 ，我们还需要将我们的项目部署到线上，让别人看到我们的成果~

Docker 是目前最流行的一种容器化应用搭建工具，我们可以通过 Docker 快速地将我们的应用部署在任何支持 Docker 的地方，哪怕是 Raspberry Pi 还是公司的主机上。

而 DaoCloud 则是目前国内做 Docker 商业化体验最好的公司，他们提供了一系列帮助开发者和企业快速使用 Docker 进行项目部署的工具。这里我们将介绍如何将我们的 DEMO 部署到 DaoCloud 的简易容器中。

Dockerfile

在使用 Docker 进行项目部署前，我们需要在项目中新建一个 Dockerfile 以表达我们的镜像构建任务。

因为我们用的是 Node.js 作为项目基础运行环境，所以我们需要从 Node.js 的官方 Docker 镜像中引用过来。

```
FROM node:onbuild
```

因为我们已经在 `package.json` 中写好了对 `gulp` 的依赖，这样我们就可以直接在 `docker build` 的时候对代码进行编译。

```
RUN ./node_modules/.bin/gulp
```

此外，我们还需要安装另外一个依赖库 `pm2`，我们需要使用 `pm2` 作为我们项目的守护程序。

```
$ npm install pm2 --save-dev
```

然后，我们简单地向 Docker 的宿主机器申请暴露 80 端口，并利用 `pm2` 启动 Node.js 程序。

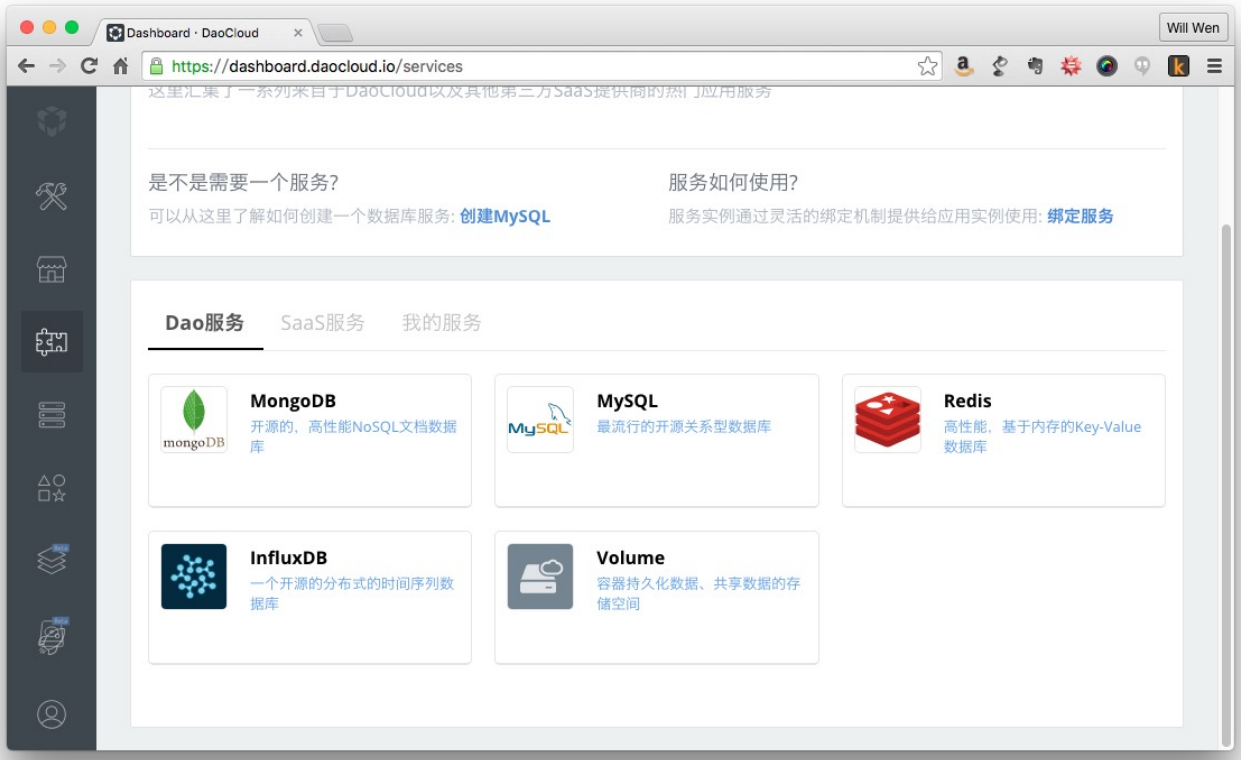
```
EXPOSE 80

CMD ./node_modules/.bin/pm2 start dist/app.js --name ES2015-In-Action --no-d
```

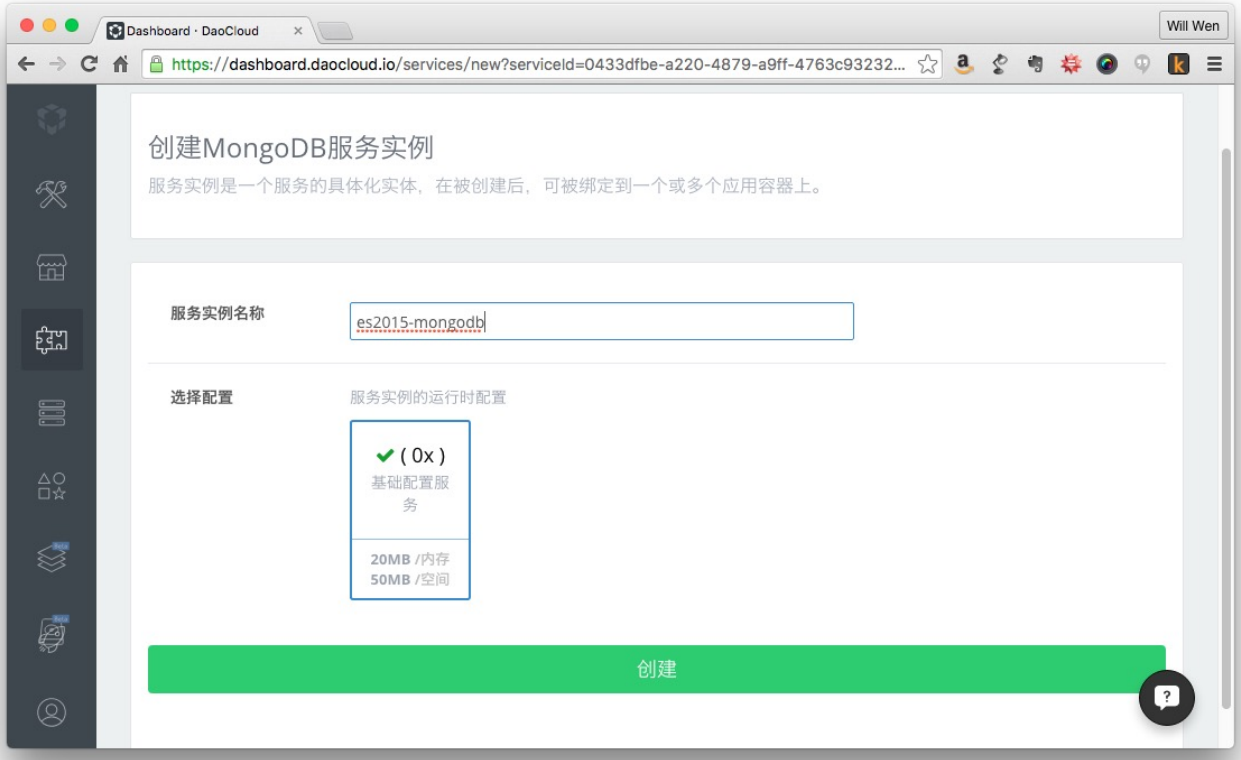
至此，我们已经完成了 Dockerfile 的编写，接下来就可以将项目代码上传到 GitHub 等地方，供 DaoCloud 使用了。这里我不再说明 Git 和 GitHub 的使用。

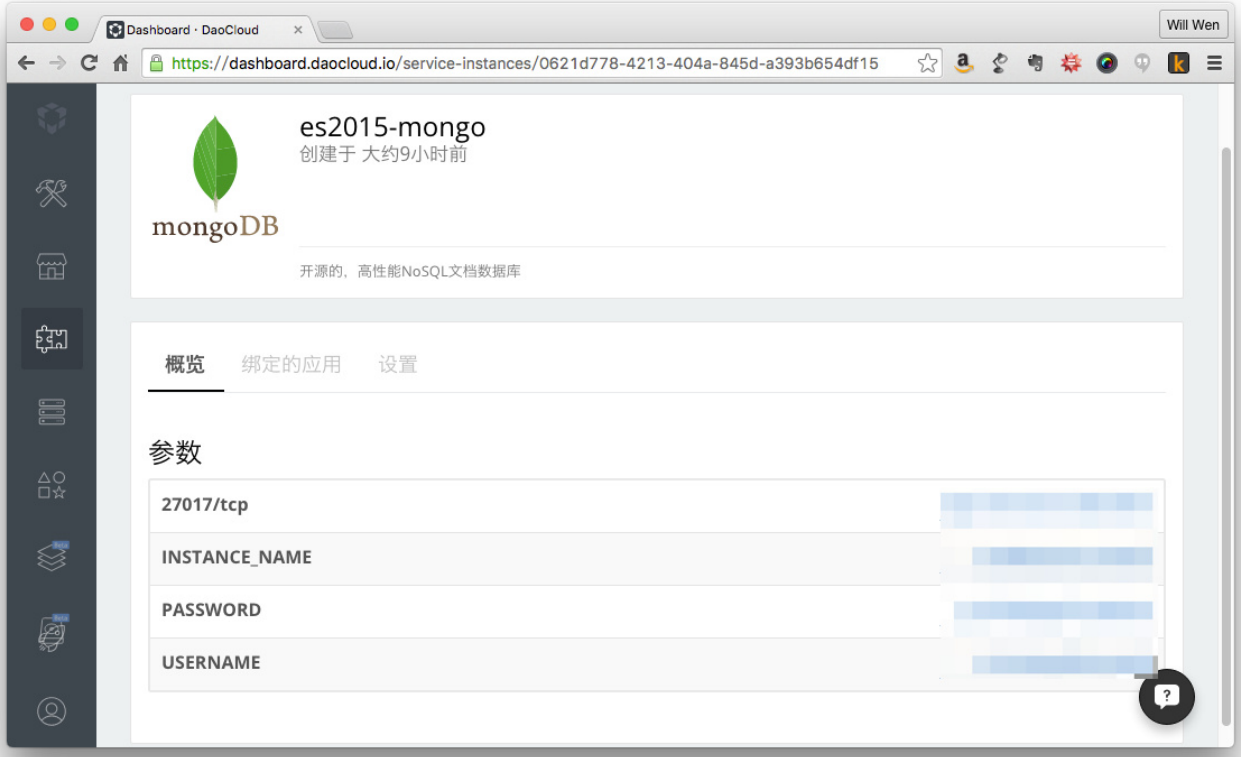
创建 DaoCloud 上的 MongoDB 服务

借助于 Docker 的强大扩容性，我们可以在 DaoCloud 上很方便地创建用于项目的 MongoDB 服务。

在“服务集成”标签页中，我们可以选择部署一个 MongoDB 服务。


Dashboard · DaoCloud
Will Wen
https://dashboard.daocloud.io/services
是不是需要一个服务?
可以从这里了解如何创建一个数据库服务: 创建MySQL
服务如何使用?
服务实例通过灵活的绑定机制提供给应用实例使用: 绑定服务
Dao服务
SaaS服务
我的服务
MongoDB
开源的，高性能NoSQL文档数据库
MySQL
最流行的开源关系型数据库
Redis
高性能，基于内存的Key-Value数据库
InfluxDB
一个开源的分布式的时间序列数据库
Volume
容器持久化数据、共享数据的存储空间

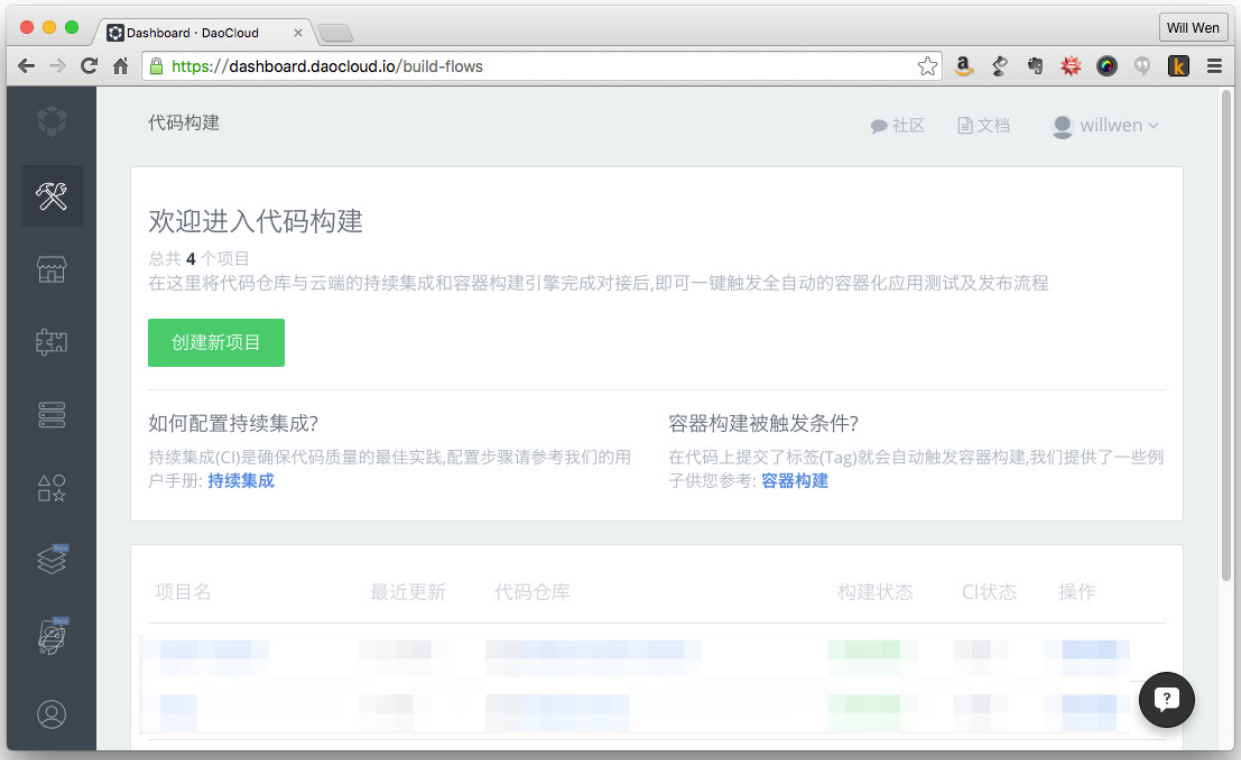

Dashboard · DaoCloud
Will Wen
https://dashboard.daocloud.io/services/new?serviceId=0433dfbe-a220-4879-a9ff-4763c93232...
创建MongoDB服务实例
服务实例是一个服务的具体化实体，在被创建后，可被绑定到一个或多个应用容器上。
服务实例名称
es2015-mongodb
选择配置
服务实例的运行时配置
(0x)
基础配置服务
20MB /内存
50MB /空间
创建



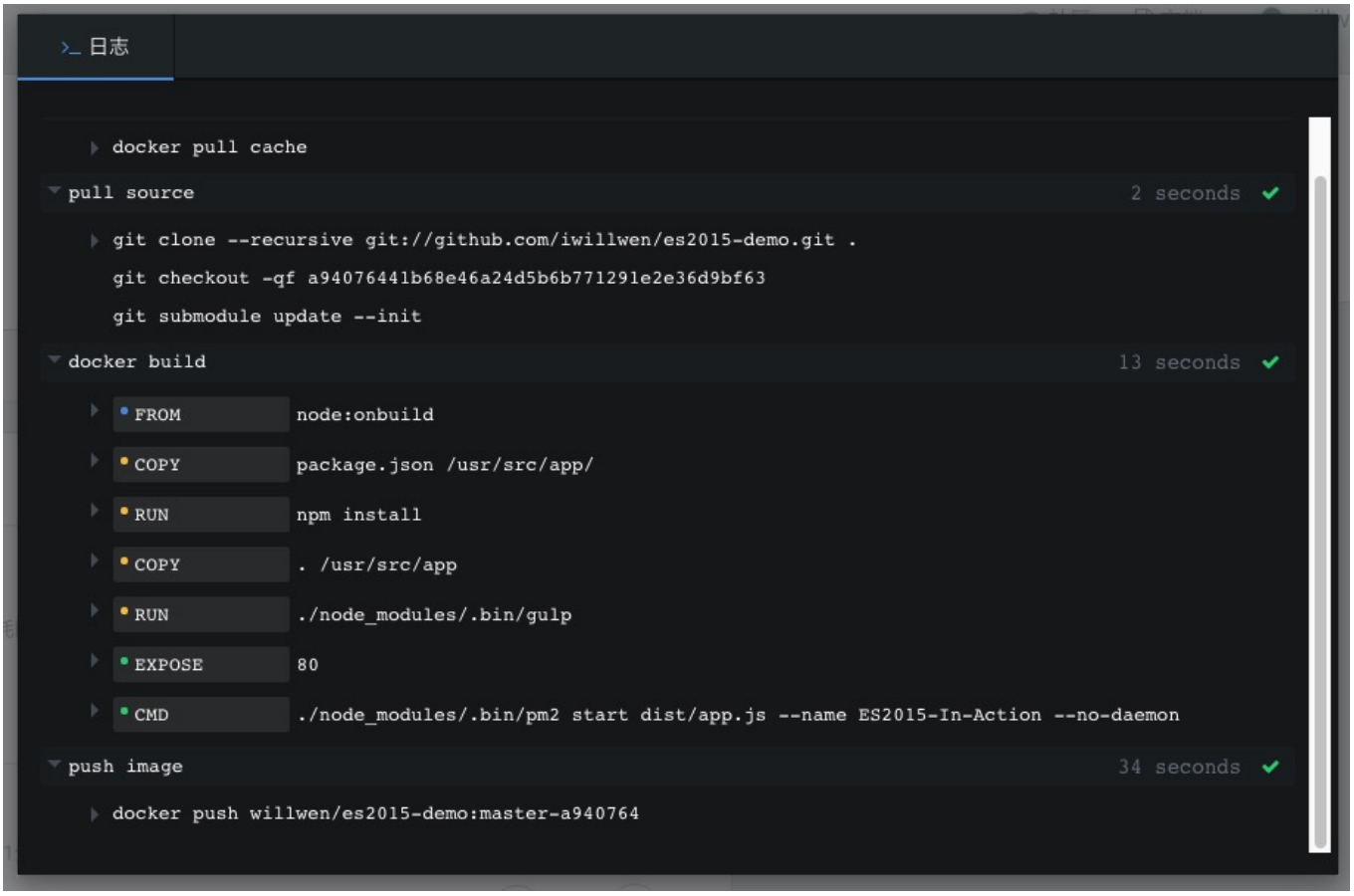
创建好 MongoDB 服务后，我们就要将我们上传到 GitHub 的项目代码利用 DaoCloud 进行镜像构建了。

代码构建

DaoCloud 提供了一个十分方便的工具，可以把我们存储在 GitHub、BitBucket、GitCafe、Coding 等地方的项目代码，通过其中的 Dockerfile 构建成一个 Docker 镜像，用于部署到管理在 DaoCloud 上的容器。

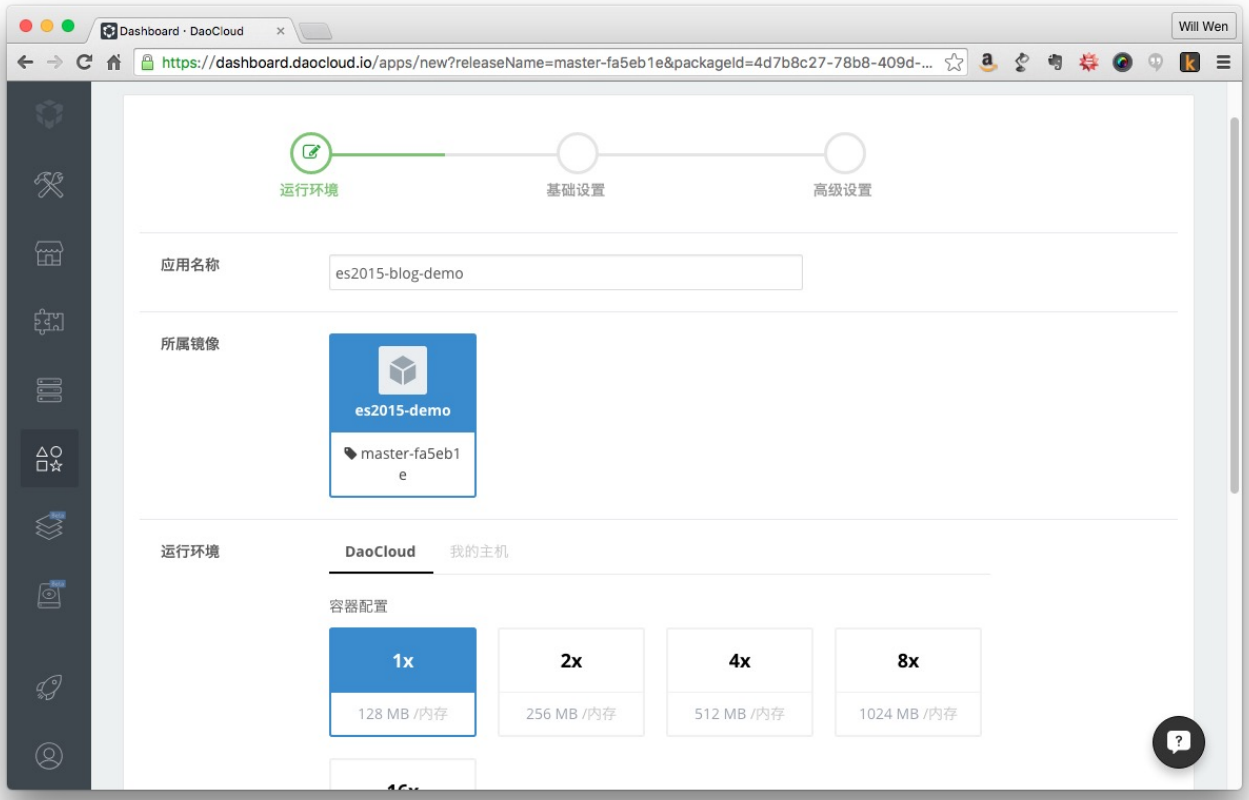


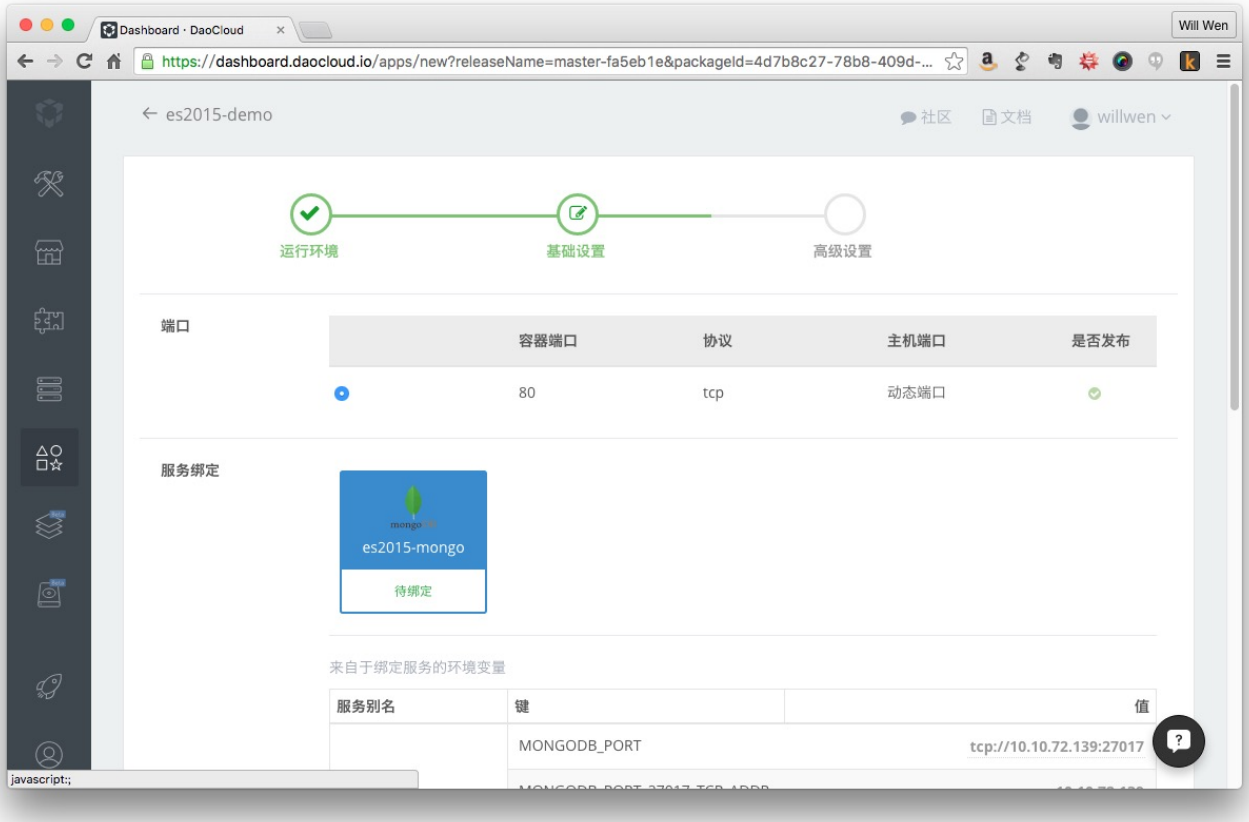
通过绑定 GitHub 账号，我们可以选择之前发布在 GitHub 上的项目，然后拉取到 DaoCloud 中。



构建完成以后，我们就可以在“镜像仓库”中看到我们的项目镜像了。

我们将其部署到一个 DaoCloud 的容器中，并且把它与之前创建的 MongoDB 服务绑定。





最后的最后，我们点击“立即部署”，等待部署成功就可以看到我们的项目线上状态了。

我们可以在这里看到我部署的 DEMO：http://es2015-demo.daoapp.io/

而我们的 DEMO，可以在这里查看详细代码：https://github.com/iwillwen/es2015-demo

至此，我们已经完成了 Node.js 端和前端的构建，并且将其部署到了 DaoCloud 上，以供浏览。下面，我们再来看看，正在发展中的 ES7 又能给我们带来什么惊喜吧。

一窥 ES7

`async/await`

上文中我们提及到 co 是一个利用 Generator 模拟 ES7 中 `async/await` 特性的工具，那么，这个 `async/await` 究竟又是什么呢？它跟 co 又有什么区别呢？

我们知道，Generator Function 与普通的 Function 在执行方式上有著本质的区别，在某种意义上是无法共同使用的。但是，对于 ES7 的 Async Function 来说，这一点并不存在！它可以以普通函数的执行方式使用，并且有著 Generator Function 的异步优越性，它甚至可以作为事件响应函数使用。

```
async function fetchData() {
  let res = await fetch('/api/fetch/data')
  let reply = await res.json()

  return reply
}

var reply = fetchData() //=> DATA...
```

遗憾的是，`async/await` 所支持的并不如 co 多，如并行执行等都暂时没有得到支持。

Decorators

对于 JavaScript 开发者来说，Decorators 又是一种新的概念，不过它在 Python 等语言中早已被玩出各种花式。

Decorator 的定义如下：

1. 是一个表达式
2. Decorator 会调用一个对应的函数
3. 调用的函数中可以包含 `target`（装饰的目标对象）、`name`（装饰目标的名称）和 `descriptor`（描述器）三个参数
4. 调用的函数可以返回一个新的描述器以应用到装饰目标对象上

PS：如果你不记得 `descriptor` 是什么的话，请回顾一下 `Object.defineProperty()` 方法。

简单实例

我们在实现一个类的时候，有的属性并不想被 `for..in` 或 `Object.keys()` 等方法检索到，那么在 ES5 时代，我们会用到 `Object.defineProperty()` 方法来实现：

```
var obj = {
  foo: 1
}

Object.defineProperty(obj, 'bar', {
  enumerable: false,
  value: 2
})

console.log(obj.bar) //=> 2
```

```
var keys = []
for (var key in obj)
  keys.push(key)
console.log(keys) //=> [ 'foo' ]

console.log(Object.keys(obj)) //=> [ 'foo' ]
```

那么在 ES7 中，我们可以用 Decorator 来很简单地实现这个需求：

```
class Obj {
  constructor() {
    this.foo = 1
  }

  @nonenumerable
  get bar() { return 2 }
}

function nonenumerable(target, name, descriptor) {
  descriptor.enumerable = false
  return descriptor
}

var obj = new Obj()

console.log(obj.foo) //=> 1
console.log(obj.bar) //=> 2

console.log(Object.keys(obj)) //=> [ 'foo' ]
```

黑科技

正如上面所说，Decorator 在编程中早已不是什么新东西，特别是在 Python 中早已被玩

出各种花样。聪明的工程师们看到 ES7 的支持当然不会就此收手，就让我们看看我们还能用 Decorator 做点什么神奇的事情。

假如我们要实现一个类似于 Koa 和 PHP 中的 CI 的框架，且利用 Decorator 特性实现 URL 路由，我们可以这样做。

```
// 框架内部
// 控制器
class Controller {
  // ...
}

var handlers = new WeakMap()
var urls = {}

// 定义控制器
@route('/')
class HelloController extends Controller {
  constructor() {
    super()

    this.msg = 'World'
  }

  async GET(ctx) {
    ctx.body = `Hello ${this.msg}`
  }
}

// Router Decorator
function route(url) {
  return target => {
    target.url = url
    let urlObject = new String(url)
```

```
    urls[url] = urlObject

    handlers.set(urlObject, target)
  }
}

// 路由执行部份
function router(url) {
  if (urls[url]) {
    var handlerClass = handlers.get(urls[url])
    return new handlerClass()
  }
}

var handler = router('/')
if (handler) {
  let context = {}
  handler.GET(context)
  console.log(context.body)
}
```

最重要的是，同一个修饰对象是可以同时使用多个修饰器的，所以说我们还可以用修饰器实现很多很多有意思的功能。

后记

对于一个普通的 JavaScript 开发者来说，ES2015 可能会让人觉得很模糊和难以学习，因为 ES2015 中带来了许多我们从前没有在 JavaScript 中接触过的概念和特性。但是经过长时间的考察，我们不难发现 ES2015 始终是 JavaScript 的发展方向，这是不可避免的。因此我要在很长一段时间内都向身边的或是社区中的 JavaScript 开发者推广 ES2015，推荐他们使用最新的技术。