前端开发核心知识进阶：50 讲从夯实基础到突破瓶颈
来自 Lucas ... · 盐选专栏

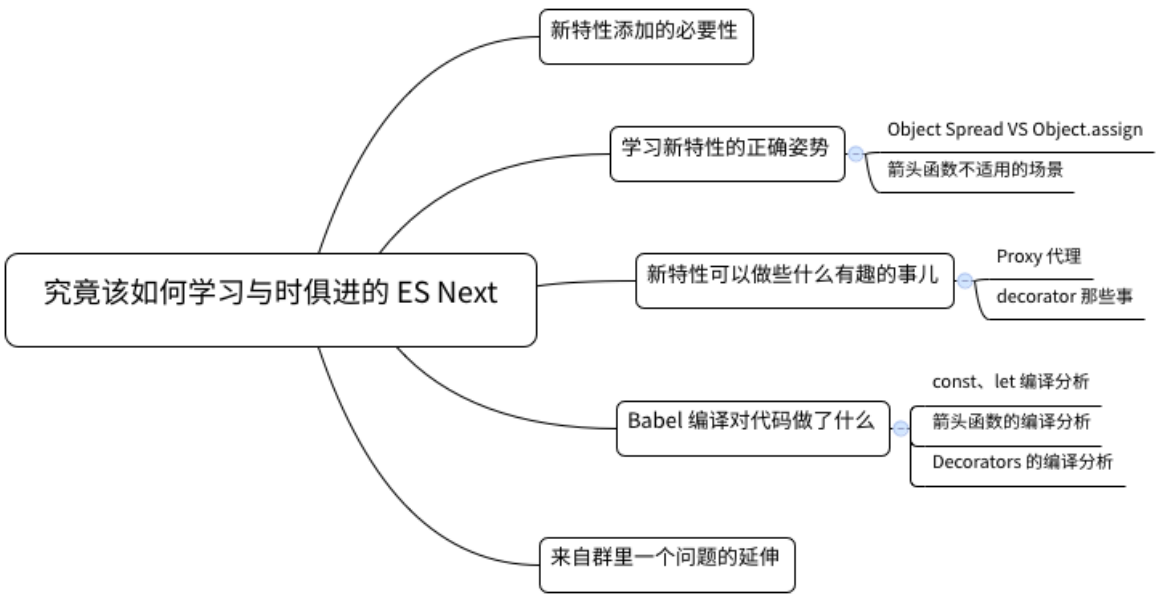
查看详情 >

究竟该如何学习与时俱进的 ES Next

JavaScript 语言规范始终在与时俱进，除了过于激进的 ES4 被「废除」之外，ES Next 始终茁壮发展。到如今，TC39（Technical Committee 39，JavaScript 委员会）已经明确表示每年更新一个版本，因此使用 ES Next 表示那些「正在演进、正在发展」的新特性集。

作为前端开发者，我们该如何看待每年一版的 ES Next，又该如何去保持学习呢？这一讲，我们就来谈谈 ES Next。我认为列举新特性没有价值，这些东西随处可见，更重要的是分析新特性的由来，剖析如何学习新特性，分析如何利用新特性。

相关知识点如下：



新特性添加的必要性

有很多人不幸患上了「JavaScript fatigue」，表示我「再也学不动了」，求「不要再更新」了。到底要不要更新？我们来从一处细节看看 ES Next 发展的必要性。

ES7 规范中定义了一个新的数组 API，签名如下：

```
Array.prototype.includes(value : any): boolean
```

它用起来就像这样：

```
[1, 2, 3].includes(3)
// true
```

从命名上就不难理解，这是判断数组中是否含有一个元素的方法，该方法最终返回一个布尔值。有的开发者可能会问，判断数组中是否含有一个元素，不是有很多现成的方法可以使用吗？我能列举出来很多：

```
[1, 2, 3].findIndex(i => i === 2)
// 1

[1, 2, 3].find(i => i == 2)
// 2

[1, 2, 3].indexOf(2)

// 1
```

难道这还不够吗？我们甚至完全可以实现一个「一模一样」的 API：

```
const includes = (array, target) =>  !!~
array.indexOf(target)

includes([1,2,3], 3)
// true
```

```
includes([1,2,3], 4)
// false
```

对于任何 ES Next 的新特性，开发者若有疑问，都可以在 TC39 提议的 GitHub 中找到，这个也不例外。我们就来分析这一新特性的意义：首先，在语义上它直观明朗，这是 indexof 所无法取代的。当然还有更深层次的必要性和不可替代性。

我们认真审视 Array.prototype.includes 这个 API，它用来判断数组是否包含某一元素，那么「是否包含」必然有判断「是否相等」的逻辑。那么这个「相等」，又是如何定义的呢？最简单的，是 == 还是 ===?

这里可以说明的是：Array.prototype.indexOf 采用的是 === 比较，而 Array.prototype.includes 不同，它采用了 SameValueZero() 比较。

SameValueZero() 是什么呢？这个是引擎内置的比较方式，并没有对外接口，其实现采用了 Map 和 Set。采用这种比较，最直接的收益就是可以判断 NaN：

```
[NaN].includes(NaN) // true
[NaN].indexOf(NaN) // -1
```

因为：

```
NaN === NaN
// false
```

而 SameValueZero() 却不受干扰，可以准确地判断 NaN === NaN。

这就是新特性区别于老传统的不同，很多都体现在细节上，需要开发者用心体会，这也是学习 ES Next 的「正确姿势」之一。

当然，新特性除了体现在这些细节上，也体现在更多更有意义的方面，比如异步处理，相信学习过前面课程的读者已经能有所体会了。想想异步处理从回调到 Promise，再到 generator 和 async/await，也许你就会明白语言发展的必要性。

学习新特性的正确姿势

前面我们已经通过剖析一个细节，为大家介绍了学习的「正确姿势」。除了认真、事无巨细以外，有的时候还需要一些「刨根问底」、「吹毛求疵」的态度。我们来进入场景。

Object Spread VS Object.assign

Object Spread 和 Object.assign 在很多情况下做的事情是一致的，它们都属于 ES Next 的新特性，当然 Object Spread 更新。事实上，规范说明中，也告诉我们 「object spread」：{... obj} 和 Object.assign（{}，obj） 是等价的。

但是一定还具有区别。实际上，Object.assign() 将会修改它的第一个参数对象，这个修改可以触发其第一个参数对象的 setter。熟悉函数式编程，了解 React/Redux 技术栈的读者，可能会听说过「不可变性」的概念。从这个层面上讲，Object spread 操作符会创建一个对象副本，而不会修改任何值，这也许是更好的选择。

当然，喜欢「抬杠」的读者可以说，如果使用 Object.assign()，我们始终保证一个空对象作为第一个参数，也能实现同样的「不可变性」。话虽是如此，但是既然你「抬杠」，那我也「抬杠」，我就告诉你这么做的话，性能比 Object Spread 就差的比较多了。

采用 object-assign-vs-object-spread 提供的 benchmark：

```
const Benchmark = require('benchmark');

const suite = new Benchmark.Suite;

const obj = { foo: 1, bar: 2 };

suite.
 add('Object spread', function() {
   ({ baz: 3, ...obj });
 }).
 add('Object.assign()', function() {
```

```
  Object.assign({}, obj, { baz: 3 });
})
```

得出结果：

```
Object spread x 3,065,831 ops/sec +-2.12% (85 runs
sampled)
Object.assign() x 2,461,926 ops/sec +-1.52% (88 runs
sampled)
Fastest is Object spread
```

使用 Object spread 性能要明显领先于 Object.assign。

箭头函数不适用的场景

我们再来分析一道思考题，「**哪些场景下不适合使用 ES6 箭头函数**」？

这个问题不是死板地考察队 ES Next 箭头函数的理解，而是反其道行之，考察其不适用的场景。 回答这个问题，我们思考：开发者习惯使用箭头函数来对 this 指向进行干预，那么反过来说，「不需要进行 this 指向干预的情况下，我们就不适合使用箭头函数」。总结下来，有：

- 构造函数的原型方法上

构造函数的原型方法需要通过 this 获得实例，因此箭头函数不可以出现在构造函数的原型方法上：

```
Person.prototype = () => {
 // ...
}
```

这样的做法是错误的。

- 需要获得 arguments 时

箭头函数不具有 arguments，因此在其函数体内无法访问这一特殊的伪数组，那么相关场景下也不适合使用箭头函数。

使用对象方法时

```
const person = {
 name: 'lucas',
 getName: () => {
   console.log(this.name)
 }
};
person.getName()
```

上述代码中，getName 函数体内的 this 指向 window，显然不符合其用意。

使用动态回调时

同理，类似下面这种对回调函数的 this 有特殊场景需求的用法，箭头函数的 this 无法满足要求：

```
const btn = document.getElementById('btn')

btn.addEventListener('click', () => {
   console.log(this === window)
});
```

当点击 id 为 btn 的按钮时，将会输出：true，事件绑定函数的 this 指向了 window，而无法获取事件对象。

「箭头函数」不适用的场景社区上也有相关文章分析，我个人认为这是一个很好的切入点。思考「哪些场景不适用」，不仅能够全面了解学习新特性，也能够和老知识融会贯通，可谓学习 ES Next 的正确姿势之一了。

新特性可以做些什么有趣的事儿

可能有开发者有这样的体会：「ES Next」那么多新特性，但是我使用的来来回回都是那么几项，很多感觉并用不上啊?

同时，讲了这么多细节，我们可以用新特性实现哪些很 cool 的操作呢？其实除了日常用到的新特性以外，一些不为大家所熟知的特性往往在框架开发，或者实现更深层次行为操作的场景中，应用比比皆是。比如 Proxy，它可以用来定义对象各种基本操作的自定义行为，比如 Vue 双向绑定的实现，就可以借助 Proxy 完成。

Proxy 代理

我们先来看一些简单的场景，借用上节课的例子：

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

let proxyPersonClass = new Proxy(Person, {
 apply (target, context, args) {
   throw new Error(`hello: Function ${target.name} cannot
be invoked without 'new'`)
 }
})
```

我们对 Person 构造函数进行了代理，这样就可以防止非构造函数实例化的调用：

```
proxyPersonClass('lucas')

// VM173058:9 Uncaught Error: hello: Function Person
cannot be invoked without 'new'
   at :1:1

new proxyPersonClass('lucas')
// {name: "lucas"}
```

同样道理，也可以静默处理非构造函数实例化的调用，将其强制转换为 new 调用：

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

let proxyPersonClass = new Proxy(Person, {
 apply (target, context, args) {
   return new (target.bind(context, ...args))()
 }
})
```

这样即便在不使用 new 关键字时，仍然可以得到 new 调用的实例：

```
proxyPersonClass('lucas')
// Person {name: "lucas"}
```

另外一个场景：熟悉前端测试的读者，可能对断言 assert 并不陌生，一种常用的使用方式是：

```
const lucas = {
   age: 23
}
assert['lucas is older than 22!!!'] = 22 > lucas.age

// Error: lucas is older than 22!!!
```

我们看 assert 赋值语句右侧表达式结果为一个布尔值，当表达式成立时，断言不会抛出；如果 assert 赋值语句右侧表达式不成立时，也就是断言失败时，断言抛出错误。

乍看上去这是不是很神奇？如果面试过程中，面试官要求你实现一个 assert，该怎么做呢？这样一个断言库本质上还是拦截 assert 对象的赋值（set）操作：

```
const assert = new Proxy({}, {
 set (target, warning, value) {
   if (!value) {
       console.error(warning)
   }
 }
})
```

这样我们只需要判读对 assert 的赋值值是否为 true，如果不为 true，则打印错误。

是不是很简单？这样我们就可以随意进行断言：

```
const weather = 'cold'
assert['The weather is not good!!!'] = weather === 'good'

// Error: The weather is not good!!!
```

这些只是 Proxy 实现的一些很简单的例子，这里抛砖引玉，大家可以充分发挥想象力，创造更多的玩法。

Decorator 那些事

除此之外，介绍给大家的就是 ES7 中的装饰器 Decorator。

装饰器（Decorators）让你可以在设计时对类和类的属性进行「注解」和修改。

说直白一些，Decorator 就是给类添加或者修改类的属性与方法的。这么听上去似乎跟我们刚刚介绍的 proxy 似乎有异曲同工之秒。一些开发者可能已经在使用 Decorator 了，这里我借助 autobind 这个类库的实现，介绍一下 Decorator 的玩法。

我们知道：

```
class Person {
 constructor (name) {
      this.name = name
 }
 getPersonName() {
   return this.name
 }
}

const person = new Person('lucas')

const fn = person.getPersonName

fn()

//  Cannot read property 'name' of undefined
   at getPersonName (:6:17)
   at :3:1
```

这里在执行 fn() 时，this 已经指向了 window，使用 autobind 可以完成对 this 的绑定：

```
class Person {
 constructor (name) {
      this.name = name
 }
 @autobind
 getPersonName() {
   return this.name
 }
}
```

那么 autobind 怎么实现呢？伪代码如下：

```
 function autobind(target, key, { value: fn, configurable,
enumerable }) {
```

```
 return {
   configurable,
   enumerable,
   get() {
     const boundFn = fn.bind(this);
     defineProperty(this, key, {
       configurable: true,
       writable: true,
       enumerable: false,
       value: boundFn
     });
     return boundFn;
   },
   set: createDefaultSetter(key)
 };
}
```

autobind 这个 decorator 接受以下三个参数。

target：目标对象，这里是作用于 Person 中的函数、属性的

key：属性名称

descriptor：属性原本的描述符

autobind decorator 函数最终返回描述符，这个描述符运行时相当于调用 Object.defineProperty() 修改原有属性，我们看最终修改的结果为：

```
{
   configurable,
   enumerable,
   get() {
     const boundFn = fn.bind(this);
     defineProperty(this, key, {
       configurable: true,
       writable: true,
```

```
        enumerable: false,
        value: boundFn
      });
      return boundFn;
    },
    set: createDefaultSetter(key)
}
```

这样在使用 get 赋值时（const fn = person.getPersonName），赋值结果通过 const boundFn = fn.bind(this) 进行对 this 绑定，并返回绑定 this 后的结果，因此达到了我们对 getPersonName 属性方法绑定 this 的目的。

这就是 decorator 在 autobind 这个库中的应用，这个库大家接触的不多，也许有 React 开发者使用 autobind 来对事件处理函数进行 this 绑定。总之，autobind 源码实现很好地利用了 decorator 特性。

Babel 编译对代码做了什么

为了能够使用到新鲜出炉的 ES Next 新特性，必不可少的一环就是 Babel，相信每个前端开发者都听说过它的大名。虽然 Babel 目前已经是个丰富的生态社区了，但是它刚出道时的目标，以及目前最核心的能力就是：编译 ES Next 代码，进行降级处理，进而规避了兼容性问题。

那么 Babel 编译到底是施展了什么魔法呢？它的核心原理是使用 AST（抽象语法树）将源码进行分析并转为目标代码，这中间的细节部分我们会在工程化章节中有所涉及。在上一讲中，我们已经对 ES6 class 的编译产出进行了分析，这里再分析一些比较典型的编译结果。

const、let 编译分析

简单来说，const、let 一律转成 var。为了保证 const 的不可变性：Babel 如果在编译过程中发现对 const 声明的变量进行了二次赋值，将会直接报错，这样就在编译阶段进行了处理。至于 let 的块级概念，ES5 中，我们一般通过 IIFE 实现块级作用域，但是 Babel 处理非常取巧，那就是在块内给变量换一个名字，块外自然就无法访问到。

在之前的课程中我们介绍使用 let 或者 const 声明的变量，存在暂时性死区（TDZ）现象。简单回顾下：代码声明变量所在的区块中，会形成一个封闭区域。在这个区域中，只要是在声明变量前使用这些变量，就会报错。

```
var foo = 123

{
 foo = 'abc'
 let foo
}
```

将会报错：Uncaught ReferenceError: Cannot access 'foo' before initialization。

那么 Babel 怎么编译模拟这种行为呢？其实我们提到 Babel 编译会将 let、const 变量重新命名，同时在 **JavaScript 严格模式（strict mode）不允许使用未声明的变量**，这样在声明前使用这个变量，也会报错。如下代码：

```
"use strict";
var foo = 123
{
 _foo = 'abc'
 var _foo
}
```

我们加上严格模式的标记，自然就可以实现了 TDZ 的效果。

对于经典的 for 循环问题，Babel 的处理并不让我们感到意外：

```
let array = []
for (let i = 0; i < 10; i++) {
 array[i] = function () {
   console.log(i)
 }
}
array[6]()
```

```
// 6

let array = []
for (var i = 0; i < 10; i++) {
 array[i] = function () {
   console.log(i)
 }
}
array[6]()
// 10
```

为了保存每一个循环变量 i 的值，Babel 也使用了闭包：

```
"use strict";
var array = [];

var _loop = function _loop(i) {
 array[i] = function () {
   console.log(i);
 };
};

for (var i = 0; i < 10; i++) {
 _loop(i);
}
array[6]();
```

细心的同学可能还会想到：使用 const 声明的变量一旦声明，其变量（内存地址）是不可改变的。

```
const foo = 0
foo = 1

// VM982:2 Uncaught TypeError: Assignment to constant
variable
```

对此 Babel 的处理有比较有意思：

```
"use strict";
function _readOnlyError(name) { throw new Error("\"" +
name + "\" is read-only"); }

var foo = 0;
foo = (_readOnlyError("a"), 1);
```

我们看编译结果，Babel 检测到 const 声明的变量被改变赋值，就会主动插入了一个 `_readOnlyError` 函数，并执行此函数。这个函数的执行内容就是报错，因此代码执行时就会直接抛出异常。

箭头函数的编译分析

对于箭头函数的转换，也不难理解，看代码：

```
var obj = {
   prop: 1,
   func: function() {
       var _this = this;

       var innerFunc = () => {
           this.prop = 1;
       };

       var innerFunc1 = function() {
           this.prop = 1;
       };
   },

};
```

转换为：

```
var obj = {
   prop: 1,
   func: function func() {
       var _this2 = this;

       var _this = this;

       var innerFunc = function innerFunc() {
           _this2.prop = 1;
       };

       var innerFunc1 = function innerFunc1() {
           this.prop = 1;
       };
   }

};
```

通过 `var _this2 = this;` 保存当前环境的 this 为 `_this2`，在调用 innerFunc 时，用新储存的 `_this2` 进行替换函数体内的 this 即可。

Decorators 的编译分析

上面的内容中，我们介绍了 decorators 新特性，那么 Babel 又是怎么编译 decorators 的呢?

使用方式：

```
class Person{
 @log
 say(){}
}
```

我们有一个名为 log 的 decorators，Babel 编译：

```
_applyDecoratedDescriptor(
 Person.prototype,
 'say',
 [log],
 Object.getOwnPropertyDescriptor(Person.prototype, 'say'),
 Person.prototype)
)

function _applyDecoratedDescriptor(target, property,
decorators, descriptor, context) {
 var desc = {};
 Object['ke' + 'ys'](descriptor).forEach(function (key) {
   desc[key] = descriptor[key];
 });
 desc.enumerable = !!desc.enumerable;
 desc.configurable = !!desc.configurable;

 if ('value' in desc || desc.initializer) {
   desc.writable = true;
 }

 desc = decorators.slice().reverse().reduce(function
(desc, decorator) {
   return decorator(target, property, desc) || desc;
 }, desc);

 if (context && desc.initializer !== void 0) {
   desc.value = desc.initializer ?
desc.initializer.call(context) : void 0;
   desc.initializer = undefined;
 }

 if (desc.initializer === void 0) {
   Object['define' + 'Property'](target, property, desc);
   desc = null;
 }
```

```
 return desc;
}
```

我们看这里主要依赖了 _applyDecoratedDescriptor 方法。这个方法将返回描述符 desc，具体执行逻辑为：先把所有 decorators 包装成一个数组，作为 _applyDecoratedDescriptor 方法的第三个参数传入，对于 decorators 这个数组，我们将 target、property、desc 作为参数，依次遍历执行数组中的每一个 decorator 函数。执行后返回每一个 decorator 产生的属性描述符。上述代码样例就是：decorators 这个数组只有一项：log。[log]，遍历数组时，我们将 target、property、desc 作为参数传给 log 函数并执行：log(target, property, desc)，返回结果即是新的属性描述符。

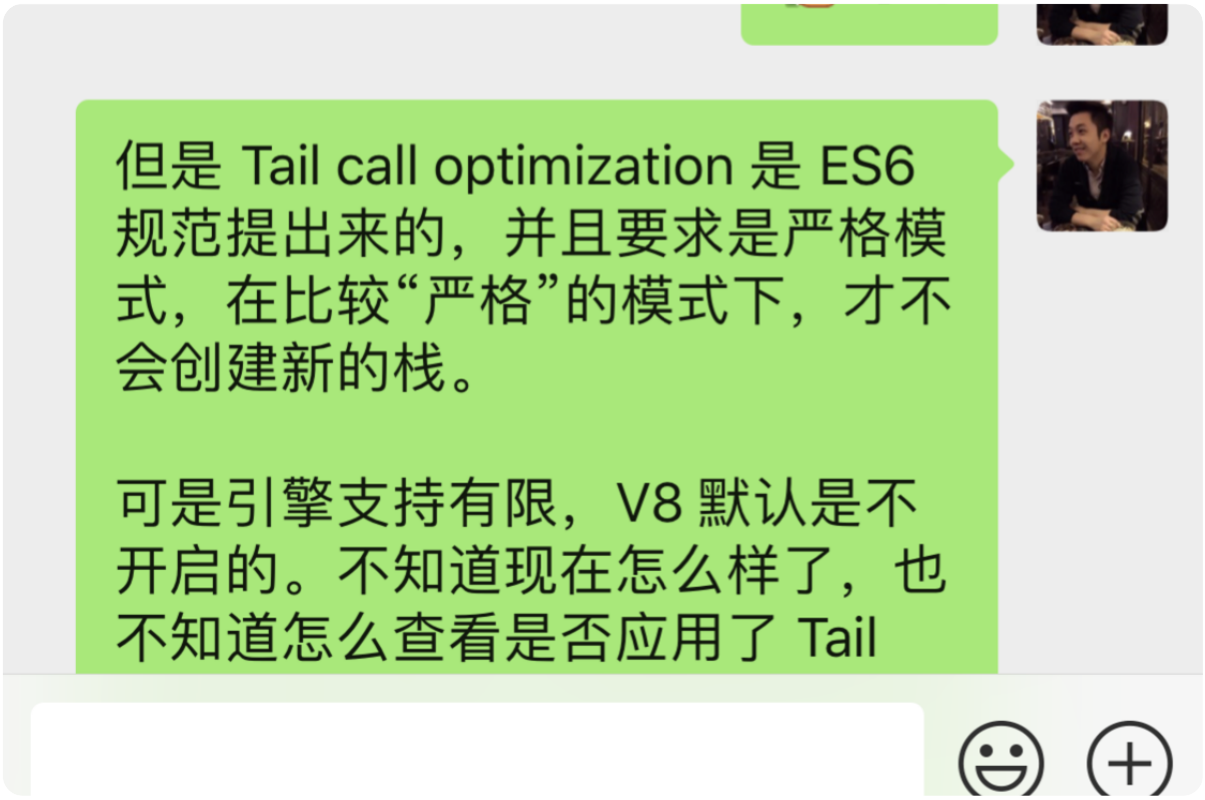
如果读者对于 decorators 特性能够熟练掌握，上述源码的理解并不困难。

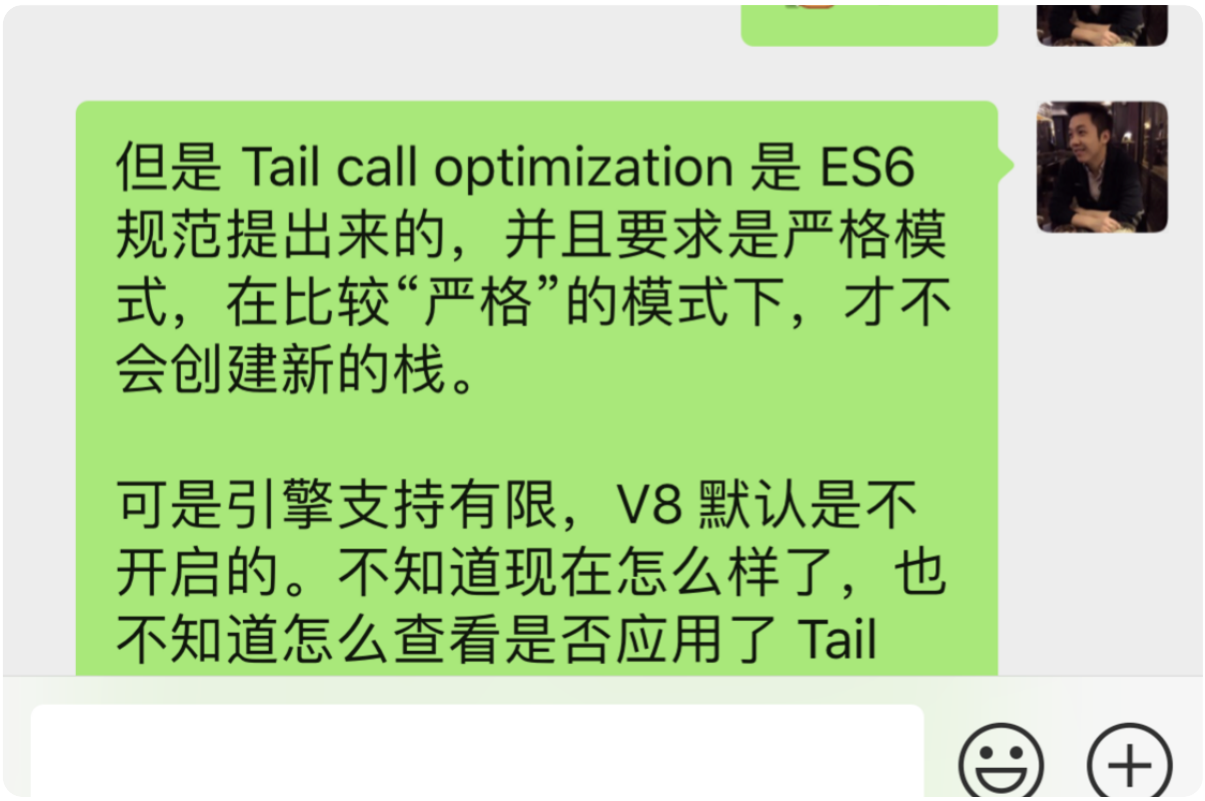
再加上上一节课对 class 编译结果的分析，我们可以知道：Babel 并没有什么「深不可测」的魔法，感兴趣的读者可以翻看各种 ES Next 的编译结果，通过对编译结果的学习，对于基础的提高，具有帮助作用。

本小节对 Babel 编译结果的进行分析，「抛砖引玉」，希望感兴趣的读者可以自行研究更多内容。值得提醒大家的一个细节是：Babel 编译产出结果主要分为两种模式，normal 模式的转换更贴近 ES Next 的写法，力求编译转换的更少，更「激进」。而另一种模式，loose 模式则更贴近 ES5 或者现有 ES 老规范的写法，也就是说在兼容性上更加有保障，因此转换代码结果也可能会更加的复杂。

来自群里一个问题的延伸

前两天本课程的核心群里，有读者问了一个关于 ES6 尾递归调用的问题。我解释了什么样的行为算是尾递归调用优化，什么行为不能算尾递归调用优化。





简而言之：递归非常耗费内存，也很容易发生「栈溢出」错误。但是对于尾递归来说，之所以可能形成优化，是因为全部执行过程中不会在调用栈上增加新的堆栈帧，而是直接更新调用栈，进而永远不会发生「栈溢出」错误。因此真正实现尾递归调用优化，最关键的是改写递归函数，确保最后只调用自身。

我们来看 fibonacci 数列求和的例子：

```
const fibonacci = n => {
 if (n === 0) return 0
 if (n === 1) return 1
 return fibonacci(n - 1) + fibonacci(n - 2)
}
```

fibonacci 数列求和非常耗费内存，如果用尾递归进行优化：

```
const fibonacciTail = (n, a = 0, b = 1) => {
 if (n === 0) return a
 return fibonacciTail(n - 1, b, a + b)
}
```

我们看，每次调用 fibonacciTail 函数后，会继续递归调用 fibonacciTail，函数的 n 会依次递减 1，它实际上是用来记录递归剩余求和的次数。而 a 和 b 两个参数在每次递归时也会在计算后再次传入 fibonacciTail 函数，最终返回值为 a，a 是上一次 a + b 的结果。这样每次递归都不会增加调用栈的长度，只是更新当前的堆栈帧而已。也就避免了内存的浪费和爆栈的危险。

然而可惜的是，据我所知，很多浏览器引擎并没有支持尾递归调用优化，即便支持，也要求代码运行环境在 strict mode 下。

那么，对于不支持尾递归调用优化的场景，我们可以做些什么实现类似的优化呢？答案一般有两个：蹦床函数和改为循环。改为循环：

```
const fibonacciLoop = (n, a = 0, b = 1) => {
 while(n--) {
   [a, b] = [b, a + b]
 }
 return a
}
```

这样一来就不存在函数的多次调用。因此，将递归改为循环，是防止递归暴栈的重要优化点之一。

另外一个优化手段是使用蹦床函数，我们来看蹦床函数：

```
const trampoline = func => {
   while(func && func instanceof Function){
       func = func()
   }
   return func
}
```

蹦床函数其实并没有实现真正的尾递归，它只是将整个执行过程拆散，还是类似循环的效果：每次产生一个结果，该结果将会对下一次执行产生影响，就像蹦床一样，越蹦越高。我们看蹦床函数接受一个函数作为参数，在蹦床函数内部执行这个函数，如果执行结果，也就是该函数的返回值还是一个函数，那么就继续执行。一直到返回值不再是一个函数时，我们返回最终的结果。

在使用蹦床函数时，我们的 fibonacci 函数需要进行一定的改动：

```
const fibonacciFunc = (n, a = 0, b = 1) => {
 if (n > 0) {
   [a, b] = [b, a + b]

   return fibonacciFunc.bind(null, n - 1, a , b)
 }
 else {
   return a
 }
}
```

在使用时：

```
trampoline(fibonacciFunc(10))
```

就能带到良好的优化效果。

这是一种比较「取巧」的方式，并不是实现了真正的尾递归调用优化。那么有没有真正实现尾递归调用优化的手段呢？答案也是有的：

```
const tailCallOpt = func => {
 let result
 let started = false

 const accumulated = []

 return function accumulator() {
   accumulated.push(arguments)
   if (!started) {
     started = true

     while (accumulated.length) {
       result = func.apply(this, accumulated.shift())
     }

     started = false

     return result
   }
 }
}
```

同样，我们改动相应的 fibonacci 函数为：

```
const fibonacciTailOpt = tailCallOpt(function (n, a = 0, b
= 1) {
 if (n === 0) return a

 return fibonacciTailOpt(n - 1, b, a + b)
})

fibonacciTailOpt(5)
```

我们观察整个实现过程，结合修改后的 fibonacciTailOpt 函数尝试理解：
tailCallOpt 接受一个待优化的函数 func，返回一个新的 accumulator 函数。执行 fibonacciTailOpt(5) 就是第一步执行 accumulator。

第一次执行 accumulator 时，先将参数推入 accumulated 数组当中，started 标记为 true。然后进入 while 循环，循环中执行待优化的 func 函数，func 这个函数执行过程中需要保证调用 tailCallOpt 函数的返回值，这里为 fibonacciTailOpt；第二次执行 accumulator，将新的参数加入 accumulated 数组；这样 accumulated 数组长度始终不为零，循环继续进行。

整个过程就是 accumulated 数组放进去一个参数，执行一次，得到结果，accumulated 清空；再放进去新的参数，执行得到结果，accumulated 再清空，以此类推。直到 func 返回了基本类型值（非函数值），这时候 accumulated 数组不会再有新的参数进来，因此返回最终结果。

这是一个通用的尾递归调用优化的轮子实现。核心原理就是不增加调用栈，拆成调用单元去分布执行。理解起来相对晦涩。不过这只是一点延伸，和 ES Next 并不太强相关，读者简单了解一下即可。

总结

JavaScript 语言、ES 规范总是在不断进步、发展，那么每个开发者都要做到时刻学习、跟进。在这个过程中，除了了解新特性之外，新老知识相结合，融会贯通，不断去思考「是什么」、「为什么」非常重要。这节课程挑选了几个典型的特性、分析了 Babel 编译结果、最后从尾调用优化展开，内容并不算太深，但却是一个很好的的切入角度。

希望大家能够掌握学习的正确「姿势」，保持好的心态，这也是进阶路上至关重要的一点。

点击查看下一节

前端面试离不开的「面子工程」