感觉es6中的箭头函数this自动绑定了外部函数，这里讨论下这个原理

es6中的箭头函数不是用类似于bind来实现的，比如下面

“箭头函数 => 所改变的并非把 this 局部化，而是完全不把 this 绑定到里面去”。

虽然 => 箭头函数没有一个自己的 this，但当你在内部使用了 this，常规的局部作用域准则就起作用了，它会指向最近一层作用域内的 this。

在创建这个函数的时候，这个函数的作用域与作用域链（函数的作用域链将会在运行时用到）就已经决定了，而是不是在调用的时候，这句话至管重要。

此函数中作用域链中每个对象被称为一个可变对象，以“键值对”表示。当一个函数创建以后，它的作用域链被填充以这些对象，它们代表创建此函数的环境中可访问的数据：

1 function add(num1, num2){

2 　　var sum = num1 + num2;

3 　　return sum;

4 }

　　当add()函数创建以后，它的作用域链中填入了一个单独可变对象，此全局对象代表了所有全局范围定义的变量。此全局对象包含诸如窗口、浏览器和文档之类的访问接口。如下图所示：（add()函数的作用域链，注意这里只画出全局变量中很少的一部分）

运行此add函数时会建立一个内部对象，称作“运行期上下文”(execution context)，一个运行期上下文定义了一个函数运行时的环境。

且对于单独的每次运行而言，每个运行期上下文都是独立的，多次调用就会产生多此创建。而当函数执行完毕，运行期上下文被销毁。

对于每个Execution Context都有三个重要的属性，变量对象（Variable object，VO），作用域链（Scope chain）和this。这三个属性跟代码运行的行为有很重要的关系

### 函数被调用发生了什么

当一段JavaScript代码**执行**的时候，JavaScript解释器会创建执行上下文Execution Context，其实这里会有两个阶段：

* 创建阶段（当函数被**调用**，但是开始执行函数内部代码之前）
  + 创建作用域链Scope chain
  + 创建变量，函数，基础变量等VO/AO（variables, functions and arguments）
  + 设置this的值 ----- 箭头函数不会做
* 激活/代码执行阶段
  + 设置变量的值、函数的引用，然后解释/执行代码

对于"创建VO/AO"这一步，JavaScript解释器主要做了下面的事情：

* 根据函数的参数，创建并初始化arguments object------箭头函数不会做
* 扫描函数内部代码，查找函数声明（Function declaration）
  + 对于所有找到的函数声明，将函数名和函数引用存入VO/AO中
  + **如果VO/AO中已经有同名的函数，那么就进行覆盖**
* 扫描函数内部代码，查找变量声明（Variable declaration）
  + 对于所有找到的变量声明，将变量名存入VO/AO中，并初始化为"undefined"
  + 如果变量名称跟已经声明的形式参数或函数相同，则变量声明不会干扰已经存在的这类属性

箭头函数 中arguments 也不存在，this 也没有创建

参考

<http://jsrocks.org/cn/2014/10/arrow-functions-and-their-scope>

箭头函数 <http://www.cnblogs.com/vajoy/p/4902935.html>

作用域链和执行上下文 <https://www.cnblogs.com/nanchen/p/6055016.html>

<http://www.cnblogs.com/wilber2013/p/4909430.html>

<http://www.cnblogs.com/coco1s/p/4017544.html>

webpack

参考

http://www.jianshu.com/p/b95bbcfc590d

1.webpack功能

可以替代部分gulp/grunt的工作，比如打包，压缩，图片转base64等；

拥有丰富的插件，比如支持React热插拔（react-hot-loader）的功能

2.安装

$npm install webpack -g //全局安装

3.使用

首先用npm(nodejs的包管理工具，有点像yum)初始化一个包项目，将相当于你要建立一个包

$npm init // 会自动生产package.json,这个文件以后用npm来管理依赖

$npm install webpack --save-dev // 将安装webpack，并增加到package.json中，后面的--save-dev 表面这个包是在开发环境下用的，所以加入到

"devDependencies": {

"webpack": "^1.14.0"

},

6.发布服务器

安装服务器

$ npm install webpack-dev-server -g

$ webpack-dev-server --progress --colors

服务器可以自动生成和刷新，修改代码保存后自动更新画面

<http://localhost:8080/webpack-dev-server/bundle>

<https://medium.com/@markocen/webpack-2%E8%AE%A9%E5%BC%82%E6%AD%A5%E5%8A%A0%E8%BD%BD%E6%9B%B4%E8%BD%BB%E6%9D%BE-d0e671f5c6b9>

<http://www.bijishequ.com/detail/413778?p>=

<https://github.com/chemdemo/chemdemo.github.io/issues/13>

1.为什么要用webpack

传统的代码引入方式

<script src="module1.js"></script>

<script src="module2.js"></script>

<script src="libraryA.js"></script>

<script src="module3.js"></script>

缺点，全是在window全局命名空间下，容易找出命名冲突，需要自己划分命令空间；代码引入和加载顺序要自己控制

2.模块化--研究下js依赖关系

2.1通过CommonsJS（通用js规范）实现----require

require("module");

require("../file.js");

exports.doStuff = function() {};

module.exports = someValue;

引入nodejs,不知道谁让js可以用作服务端语言，把js引擎添油加醋变成了nodejs，当然也有了一些新的语法，比如遵循CommonJS规范的 require方法；

CommonsJS规范的核心 是允许模块通过require方法来同步加载所要依赖的其他模块，每个定义好的模块可以通过exports 或者module.exports来导出去，暴露接口； 这也是为了像其他服务端语言一般便于模块划分和外部包的导入。

优点：服务端便于模块化，NPM中有将近20w个包

缺点：这是同步加载的方式，浏览器加载js资源是异步方式， 假设require 跑在浏览器上，会一个个慢慢请求require资源，太卡太慢；

2.2通过AMD（异步模块化定义）实现--define

define("module", ["dep1", "dep2"], function(d1, d2) {

return someExportedValue;

});

require(["module", "../file"], function(module, file) { /\* ... \*/ });

2.3通过CMD 和AMD很相似，比如sea.js 就是实现的CMD

define(function(require, exports, module) {

var $ = require('jquery');

var Spinning = require('./spinning');

exports.doSomething = ...

module.exports = ...

})

2.4UMD规范类似于兼容 CommonJS 和 AMD 的语法糖，是模块定义的跨平台解决方案。

2.5ES6模块 http://es6.ruanyifeng.com/#docs/module

设计思想是在编译的时候就确定依赖关系，而CommonJS和AMD都只能在运行的时候确定

3.期望的模块化方式

按需分块加载

4.webpack 作用

代码做静态分析，得到各个模块的类型和依赖关系；

不同类型的模块提交给适当的加载器来做处理，比如less文件先用less加载器将他转化成css模块，再通过css模块插入到<style>标签中

疑难语句

1.样式文件处理

webpack会把css文件也编译到js中，用js来实现css的功能，如下

>

/\* style.css \*/

body { background: yellow; }

/\* entry.js \*/

require("!style!css!./style.css") // 载入 style.css 这边有个!style和!css表示 依赖的style.css需要先用style-loader,再用css-loader进行处理，webpack会自动寻找style-loader

// 下面这样也可以

// require("!style-loader!css-loader!./style.css")

document.write('It works.')

document.write(require('./module.js'))

>像下面这样也可以

将 entry.js 中的 require("!style!css!./style.css") 修改为 require("./style.css") ，然后执行：

$ webpack entry.js bundle.js --module-bind 'css=style!css'

// css文件都用style-loader和css-loader来处理下

1. css-loader 处理css中路径引用等问题

查询获取样式

以后可以这样写 var css = require("css!./file.css"); 或者import ，并且对css文件中的url自动寻找解析

1. style-loader 动态把样式写入页面

插入样式到html

可以发现html中集中有你的css样式

* sass-loader scss编译器
* less-loader less编译器
* postcss-loader scss再处理