窗体底端

CoffeeScript 详解

这是今年年初学习CoffeeScript的一篇个人总结，觉得也许会对社区有些帮助，就从[我的博客](http://lilulife.com/blog/2012/08/06/coffeescript/)转来了这里，因为水平有限，错漏之处在所难免，希望高手指点

**序**

Every language feature in CoffeeScript has been designed using this kind of process:  
attempt to take the beautiful dynamic semantics of JavaScript—object literals, function expressions, prototypal inheritance—and express them in a clean, readable, minimal way.  
by Jeremy Ashkenas, author of CoffeeScript

[CoffeeScript](http://coffeescript.org/)是一门简洁的，构架于JavaScript之上的预处理器语言，可以静态编译成JavaScript，语法主要受ruby和python影响，目前已经为众多rails和node项目采用。

为什么要用CoffeeScript?

* 更少，更紧凑，和更清晰的代码
* 通过规避和改变对JavaScript中不良部分的使用，只留下精华，让代码减少出错率，更容易维护
* 在很多常用模式的实现上采用了JavaScript中的最佳实践
* CoffeeScript生成的JavaScript代码都可以完全通过[JSLint](http://www.javascriptlint.com/)的检测

什么情况下不推荐使用CoffeeScript?

* CoffeeScript不是JavaScript的超集，也不是完全替代品，不应该在不会JavaScript的情况下使用CoffeeScript工作

CoffeeScript是一种需要**预编译的语言**，不能在运行时(Runtime)解释，这造成了她普遍被人质疑的一点，就是如果代码中出现运行时错误时难以调试，不过从实际使用上来看，因为CoffeeScript的编译结果大部分情况下自然而合理，至少我从来没有发现从生成的JavaScript代码回溯到对应的CoffeeScript代码有什么困难之处，我们稍后会看到这种对应关系的细节

这种静态编译还有一个额外的好处，就是CoffeeScript和现有的环境(浏览器,Node,Rhino等)与库完全兼容

最简单的安装和测试CoffeeScript的方法，是使用\*node.js\*的\*npm\*安装，然后使用命令行脚本实时编译

npm install -g coffee-script

*# watch and compile*

coffee -w --output lib --compile src

这里假设你的coffee代码在src目录下，这个daemon会自动检测文件的改变，并编译成js文件放到lib目录下

**语法**

与SASS/LESS和CSS的关系不同，CoffeeScript不是JavaScript的超集，不能在CoffeeScript程序中写JavaScript代码，比如function等关键字

**格式**

在js中，如果认为当前语句和随后语句是一个整体的话，就不会自己加;

如以下javascript代码

*//javascript code*

**var** y = x+f

(a+b).toString()

*//parsed to:*

**var** y = x+f(a+b).toString();

很多js中的问题由此引起(实际上现在把;放在哪里，在js社区内也是个争论的话题)

而CoffeeScript在编译时为每条语句加上;，因此在代码中**不需要**写;

CoffeeScript中的注释采用#

*# single line comment*

*###*

*multi line comment*

*###*

CoffeeScript中对空白敏感，这种做法来自python，任何需要({})的场合下，可以用缩进代替

**作用域**

在js中最糟糕的设计就是全局变量，当你忘记用var声明变量的时候，这个变量会成为全局对象上的一个属性

CoffeeScript避免了这点

foo = "bar"

会编译成

(**function**() {

**var** foo;

foo = "bar";

}).call(**this**);

任何的代码都会使用\*Immediate Function\*包装，这样foo成为了本地变量，并且，可以通过call指定的this引用全局对象

为了方便起见，之后的编译后代码描述不会再加上这个包装

实际上在CoffeeScript中，你也不需要再用var声明变量，编译后会自动加上var，并且将声明\*hoisting\*，即放到作用域的顶部，看一个来自官方文档的例子

outer = 1

change = ->

inner = -1

outer = 10

inner = change()

->是函数定义的简写方式，之后我们会探讨

编译后的js如下：

**var** change, inner, outer;

outer = 1;

change = **function**() {

**var** inner;

inner = -1;

**return** outer = 10;

};

inner = change();

这是类似ruby中的自然的作用域实现方式，inner在change()内定义成了局部变量，因为在代码中之前没有定义过

**赋值**

首先是字符串可以用类ruby的语法内嵌

target = "world"

alert "hello, **#{**target**}**"

其次是字面量，可以用类似\*YAML\*的方法定义对象字面量

object1 = one: 1, two: 2

object2 =

one: 1

two: 2

**class**: "numbers"

注意保留字class，现在可以直接作为对象的key了

数组也可以分行

arr = [

1

2

]

也可以解构赋值(Destructuring)

obj = {a:"foo", b:"bar"}

{a, b} = obj

arr = [1, 2]

[a, b] = arr

**数组**

数组的操作引入了来自ruby的Range概念，并且可以将字符串完全作为数组操作

numbers = [0..9]

numbers[3..5] = [-3, -4, -5]

my = "my string"[0..1]

判断一个值是否在数组内，在js中可以用Array.prototype.indexOf，不过IE8及以下不支持，CoffeeScript提供了跨浏览器的in操作符解决

arr = ["foo", "bar"]

"foo" **in** arr

具体的实现上，是一个对indexOf的Shim

**var** arr,

\_\_indexOf = [].indexOf || **function**(item) {

**for** (**var** i = 0, l = **this**.length; i < l; i++) {

**if** (i **in** **this** && **this**[i] === item)

**return** i;

}

**return** -1;

};

arr = ["foo", "bar"];

\_\_indexOf.call(arr, "foo") >= 0;

for..in语法可以用在数组上了，背后是用js的for循环实现的，这比数组的迭代器方法要效率高一些

**for** name, i **in** ["Roger", "Roderick"]

alert "**#{**i**}** - Release **#{**name**}**"

也具有过滤器when

prisoners = ["Roger", "Roderick", "Brian"]

release prisoner **for** prisoner **in** prisoners **when** prisoner[0] is "R"

看起来很像普通英语了，也可以用()收集遍历的结果

result = (item **for** item **in** array **when** item.name is "test")

遍历对象的属性可以用**of**,这是用js自己的for..in实现的

names = sam: seaborn, donna: moss

alert("**#{**first**}** **#{**last**}**") **for** first, last **of** names

**流程控制**

CoffeeScript使用来自ruby的省略语法，让控制流变得很紧凑，也引进了unless,not,then等语法糖式的关键字

result = **if** not **true** **then** "false"

result = unless **true** **then** "false"

CoffeeScript中非常好的一点，就是直接取消了js中的==判断，改成全部用===进行严格比较，js中的==会做大量诡异的类型转换，很多情况下是bug的来源

**if** "1" == 1

alert("equal")

**else**

alert("not equal")

在使用if来进行空值的判断时，js有时会让人困扰，因为""和0都会被转换成false，Coffee提供了?操作符解决这个问题，她只有在变量为null或undefined时才为false

""? *#true*

**null**? *#false*

也可以用常见的类似ruby中||=的方法，判断赋值，此外还可以用and,or,is关键字代替&&,||,==

hash or= {}

hash ?= {}

经常有当某个属性存在的时候，才会调用属性上的方法的情况，这时候也可以用?

knight.hasSword()?.poke()

只有当hasSword()返回对象不为空时，才会调用poke方法，以下是编译的js代码

**var** \_ref;

**if** ((\_ref = knight.hasSword()) != **null**) {

\_ref.poke();

}

另一种情况是当poke方法存在时才调用

knight.hasSword().poke?()

对应的js代码

**var** \_base;

**if** (**typeof** (\_base = knight.hasSword()).poke === "function") {

\_base.poke();

}

switch case语句也有了一些语法糖，并且会默认加上break

**switch** day

**when** "Sun" **then** go relax

**when** "Sat" **then** go dancing

**else** go work

**函数**

CoffeeScript对JavaScript的函数做了很大的简化，举个例子，看一个求和函数

sum = (nums...) ->

nums.reduce (x, y) -> x+y

sum 1,2,3

对应JavaScript

**var** sum,

\_\_slice = [].slice;

sum = **function**() {

**var** nums;

nums = 1 <= arguments.length ? \_\_slice.call(arguments, 0) : [];

**return** nums.reduce(**function**(x, y) {

**return** x + y;

});

};

sum(1, 2, 3);

* 可以使用和ruby 1.9类似的\*lambda函数\*写法->来代替function
* 参数列表放在->的前边，且可省略
* 取消了函数声明，只能将函数作为值定义
* 在CoffeeScript中，**任何语句都是表达式**(除了break和continue)，都有返回值，因此像ruby一样，不需要显式return
* js的函数参数有一个很讨厌的地方，就是参数对象arguments不是一个真正的数组，要使用数组方法，必须转换成数组[].slice.call(arguments, 0)这样，而在CoffeeScript中收束(加...)的参数是一个真正的数组

CoffeeScript的函数可以有默认参数，如

times = (a = 1, b = 2) -> a \* b

CoffeeScript的函数调用可以不用()语法包围参数，像ruby一样跟在函数名后面就可以，不过这也有时候会带来问题，特别是没有参数的调用

alert

对应的js

alert;

而不是alert()，这和ruby不同，需要注意

缩进的格式有时需要小心，比如用多个函数做参数的时候，需要这样写

$(".toggle").toggle ->

"on"

, ->

"off"

对应js

$(".toggle").toggle(**function**() {

**return** "on";

}, **function**() {

**return** "off";

});

**模式**

使用CoffeeScript的一个重要理由，就是她用自己的语法实现了很多很常用的js编程模式，而且，通常是在社区内广泛被承认的最佳实践，如果不熟悉JavaScript的这些模式，可能会在调试代码上遇到一些麻烦，不过，基本上来说还是比较简单易懂的，下面我们会花一些时间研究一下CoffeeScript是用什么样的方法来封装这些通用编程模式的

**闭包**

在js中，普遍会使用闭包实现各种事件的handler或封装模块，以下是CoffeeScript对这一普遍模式的实现

closure = do ->

\_private = "foo"

-> \_private

console.log(closure()) *#=> "foo"*

do关键词可以产生一个*Immediate Function*,下面是对应js代码

**var** closure;

closure = (**function**() {

**var** \_private;

\_private = "foo";

**return** **function**() {

**return** \_private;

};

})();

闭包中经常需要绑定this的值给闭包的私有变量，CoffeeScript使用特殊的=>语法省去了这个麻烦

[*@*clickHandler](https://ruby-china.org/clickHandler) = -> alert "clicked"

element.addEventListener "click", (e) => [*@*clickHandler](https://ruby-china.org/clickHandler)(e)

使用=>生成函数，可以看到生成代码中会加上对this的绑定

**var** \_this = **this**;

**this**.clickHandler = **function**() {

**return** alert("clicked");

};

element.addEventListener("click", **function**(e) {

**return** \_this.clickHandler(e);

});

这里CoffeeScript对于this有简单的别名@

**扩展**

在js中，所有的对象都是开放的，有时候会扩展原有对象的行为(比如对数组的ECMA5 shim)，这也称为Monkey patching

String::dasherize = -> [*@*replace](https://ruby-china.org/replace) /\_/g, "-"

::代表原型的引用，js代码如下

String.prototype.dasherize = **function**() {

**return** **this**.replace(/\_/g, "-");

};

**类**

在js中是否要模拟传统编程语言的类，是个一直以来都有争议的话题，不同的项目，不同的团队，在类的使用上会有不同的看法，不过，一旦决定要使用类，那么至少需要一套良好的实现，CoffeeScript在语言内部实现了类的模拟，我们来看一看一个完整的例子

**class** Gadget

[*@*CITY](https://ruby-china.org/CITY) = "beijing"

[*@*create](https://ruby-china.org/create): (name, price) ->

**new** Gadget(name, price)

\_price = 0

constructor: ([*@*name](https://ruby-china.org/name), price) ->

\_price = price

sell: =>

"Buy **#{***[@](https://ruby-china.org/name" \o "@name)*[name](https://ruby-china.org/name" \o "@name)**}** with **#{**\_price**}** in **#{**Gadget.CITY**}**"

iphone = **new** Gadget("iphone", 4999)

console.log iphone.name *#=> "iphone"*

console.log iphone.sell() *#=> "Buy iphone with 4999 in beijing"*

ipad = Gadget.create("ipad", 3999)

console.log ipad.sell() *#=> "Buy ipad with 3999 in beijing"*

这个Gadget类具有通常语言中类的功能:

* constructor是构造函数，必须用这个名称，类似ruby中的initialize
* name是实例变量,可以通过iphone.name获取
* 构造函数中如果给实例变量赋值，直接将[*@*name](https://ruby-china.org/name)写在参数中即可，等价于在函数体中的*[@](https://ruby-china.org/name" \o "@name)*[name](https://ruby-china.org/name" \o "@name) = name
* \_price是私有变量,需要赋初始值
* sell是实例方法
* create是类方法，注意这里使用了*[@](https://ruby-china.org/create" \o "@create)*[create](https://ruby-china.org/create" \o "@create)，这和ruby有些像，在定义时的this指的是这个类本身
* CITY是类变量

要注意的是，对于实例方法，要用=>来绑定this，这样可以作为闭包传递，比如

iphone = **new** Gadget("iphone", 4999)

$("#sell").click(iphone.sell())

如果不用=>，闭包被调用时就会丢失实例对象的值(iphone)

对于熟悉基于类的面向对象编程的人，CoffeeScript的类是一目了然的，下面来看看对应的js代码

**var** Gadget,

\_\_bind = **function**(fn, me){ **return** **function**(){ **return** fn.apply(me, arguments); }; };

Gadget = (**function**() {

**var** \_price;

Gadget.name = 'Gadget';

Gadget.CITY = "beijing";

Gadget.create = **function**(name, price) {

**return** **new** Gadget(name, price);

};

\_price = 0;

**function** Gadget(name, price) {

**this**.sell = \_\_bind(**this**.sell, **this**);

**this**.name = name;

\_price = price;

}

Gadget.prototype.sell = **function**() {

**return** "Buy " + **this**.name + " with " + \_price + " in " + Gadget.CITY;

};

**return** Gadget;

})();

以上的代码有很多值得注意的地方

* 整体上来说，CoffeeScript的类模拟使用的是一个\*构造函数闭包\*，这是最常用的模拟类的模式，好处是可以完整地封装内部变量，且可以使用new来生成实例对象
* \_price就是被封装在闭包内部的私有变量
* sell这样的实例方法是原型方法，并且在初始化时使用自定义的bind函数绑定实例(用=>定义的情况)
* create和CITY这样的类成员使用构造函数的属性实现，重复一下，在CoffeeScript类定义中的this指的是整个闭包Gadget
* Gadget.name是额外定义的类名属性

**类的继承**

CoffeeScript中为方便地实现类的继承也定义了自己的语法，我们把上面的类简化，来看一下如何继承：

**class** Gadget

constructor: ([*@*name](https://ruby-china.org/name)) ->

sell: =>

"Buy **#{***[@](https://ruby-china.org/name" \o "@name)*[name](https://ruby-china.org/name" \o "@name)**}**"

**class** IPhone **extends** Gadget

constructor: -> **super**("iphone")

nosell: =>

"Don't **#{***[@](https://ruby-china.org/sell" \o "@sell)*[sell](https://ruby-china.org/sell" \o "@sell)()**}**"

iphone = **new** IPhone

iphone.nosell() *#=> Don't Buy iphone*

* 使用extends关键字可以继承父类中的所有实例属性,比如sell
* super方法可以调用父类的同名方法
* 如果不覆盖constructor，则她被子类默认调用

来看一下对应的js代码，这有一些复杂，我们把和上边类定义中重复的地方去掉，只留下继承的实现部分

**var** Gadget, IPhone,

\_\_extends = **function**(child, parent) {

**for** (**var** key **in** parent) {

**if** ({}.hasOwnProperty.call(parent, key))

child[key] = parent[key];

}

**function** ctor() { **this**.constructor = child; }

ctor.prototype = parent.prototype;

child.prototype = **new** ctor;

child.\_\_super\_\_ = parent.prototype;

**return** child;

};

IPhone = (**function**(\_super) {

\_\_extends(IPhone, \_super);

IPhone.name = 'IPhone';

**function** IPhone() {

**this**.nosell = \_\_bind(**this**.nosell, **this**);

IPhone.\_\_super\_\_.constructor.call(**this**, "iphone");

}

IPhone.prototype.nosell = **function**() {

**return** "Don't " + (**this**.sell());

};

**return** IPhone;

})(Gadget);

这里重点有三个，

* \_\_extends函数使用了代理构造函数ctor来实现继承，这是非常普遍的js中对象继承的实践模式，进一步解释一下
  + 使用代理构造函数的目的是为了避免子类被更改时父类受到影响
  + 使用ctor.prototype = parent.prototype的意义是只继承定义在prototype上的公用属性
* 父类的类成员被直接引用拷贝到子类，而不是原型继承
* super的实现方法是parent.prototype.constructor.call(this)

**混入(Mixin)**

在ruby语言中的Mixin，能够让你的类获得多个模块的方法，可以说是对多重继承一种很好的实现，虽然在CoffeeScript中并没有像ruby的include一样的内置功能，但很容易实现她

**class** Module

[*@*extend](https://ruby-china.org/extend): (obj) ->

**for** key, value **of** obj

@[key] = value

[*@*include](https://ruby-china.org/include): (obj) ->

**for** key, value **of** obj

@::[key] = value

classProperties =

find: (id) ->

console.log("find **#{**id**}**")

instanceProperties =

save: ->

console.log("save")

**class** User **extends** Module

[*@*extend](https://ruby-china.org/extend) classProperties

[*@*include](https://ruby-china.org/include) instanceProperties

user = User.find(1)

user = **new** User

user.save()

* 继承了Module的类才可以Mixin，当然，这里也可以用组合或者直接为js的构造函数做Monkey patching
* classProperties是类成员模块，使用*[@](https://ruby-china.org/extend" \o "@extend)*[extend](https://ruby-china.org/extend" \o "@extend)来Mixin，实现是简单的拷贝对象的属性
* instanceProperties是实例成员模块，使用*[@](https://ruby-china.org/include" \o "@include)*[include](https://ruby-china.org/include" \o "@include)来Mixin，实现是拷贝对象原型的属性
* 需要指出的是，这里的拷贝是引用拷贝，有可能外部会更改被Mixin的模块内部值，更好的方法是深层值拷贝(clone)，包括JQuery在内的很多类库都实现了这类扩展方法

**结语**

CoffeeScript提供了一门比JavaScript更强大，优雅，表现力丰富的语言，但她毕竟架构于JavaScript之上，而且是静态地编译成JavaScript代码，也就是说，她不能完全避免对JavaScript中一些不良部分的滥用，比如eval,typeof,instanceof等，所以，在任何情况下，建议始终开启*Strict Mode*

"use strict"

严格模式是一个ECMA5标准提出的js子集，禁用了很多js设计中不好的方面，在未来会逐渐成为js的语言标准，详细介绍在[这里](https://developer.mozilla.org/en/JavaScript/Strict_mode#Changes_in_strict_mode)