CoffeeScript 小书

The Little Book on CoffeeScript Chinese version

美.Alex MacCaw 著

寸志 铁金龙 译

本书译自The Little Book on CoffeeScript，原是Github上的一个开源项目，采用MIT协议进行授权。
译者是http://island205.com/，对于中文版同样采用MIT协议进行授权。
本书的网络版在http://island205.github.com/tlboc/，在豆瓣阅读上也可以找到。
电子书由whtsky制作。

CoffeeScirpt是什么?

CoffeeScript是一门小巧的语言，会编译为JavaScript。它的语法风格受到了Ruby和Python影响，很多特性都借鉴于这两种语言。我们写作本书的目的在于帮助你学习CoffeeScript，明白最佳实践是什么，以及帮助你开始创建有意思的客户端程序。这本书很小，仅仅只有五章，但是对与CoffeeScript这门小语言来说已足够。

这本书是完全开源的，作者是Alex MacCaw (或者 @maccman)，David Griffiths、Satoshi Murakami和 Jeremy Ashkenas也做了不小的贡献。

如果你有任何勘误和建议，千万别吝啬到本书的GitHub page发个ticket。或许你们还对我的另外一本书JavaScript Web Applications by O'Reilly感兴趣，我在该书中对富JavaScript应用以及如何把状态转移到客户端进行了探索。

好了，开始我们的CoffeeScript探索之旅吧。为什么CoffeeScript要比原生的JavaScript好?首先，能够少写代码——CoffeeScript非常简洁，充分地利用空格。以我的经验来看，比起纯JavaScript的话，它能减少三分之一到一半的代码量。还有，CoffeeScript开有一些优雅的特性，比方说列表解析、原型符号别名和类等等，能够有效的减少需要你的输入。

更重要的是，JavaScript有很多不为人知的 秘密，这些秘密往往让无经验的开发者摔跤。CoffeeScript有原则地选择了一些JavaScript的特性，巧妙地避开了这些不足，解决了该语言的怪癖。

CoffeeScript不是JavaScript的超集，因此尽管你可以在CoffeeScript中的使用外部的JavaScript类库，但是如果你在没有转化之前而直接编译当前的JavaScript的话，会出现语法错误。编译器会把CoffeeScript代码转化为相对于的JavaScript，这样在运行时就不需要解释了。

首先澄清一些误解。由于处理运行时错误需要JavaScript相关的知识，要写CoffeeScript就得了解JavaScript。但是话说回来，运行时错误通常比较明显，并且到目前位置，我没觉得从JavaScript映射到CoffeeSCript会有什么问题。第二个问题是我经常听到CoffeeScript相关的话题是速度。即，CoffScript编译后的JavaScript运行起来相比与之等价的纯JavaScript代码要慢。但实际情况证明并不是问题。CoffeeScript看起来与徒手写的JavaScript代码运行速度相当，甚至更快。

CoffeeScript的劣势是什么？是的，在你和JavaScript之间介多了编译这一步。CoffeeScript也在尝试尽力通过产生优雅的可读性强的JavaScript，以及在服务器端集成自动编译来弥补这个问题。另外一个缺陷是，作为一个新的语言，事实上现阶段社区也还比较小，想找个懂这门语言的合伙人会花费你的大量你的时间。当然，CoffeeScript发展迅猛，相关的IRC列表也是人才济济，如果你有什么问题的话，都会得到迅速的解答。

CoffeeScript的用途并不仅限于浏览器，把它用在JavaScript实现的服务端也非常不错，比方说在 Node.js上。还有，CoffeeScript越来越广泛，有更多的集成，比方说它已经是Rails3.1的标配。现在正是进入CoffeeScript学习的时机。你现在为学习这门语言付出的时间在以后会以为你节约大量的时间作为回报的。

初始化安装

一种尝试这个类库最简单的方式就是直接在浏览器中使用它，访问http://coffeescript.org，点击*Try CoffeeScript*标签。这个网站使用浏览器版的CoffeeScript编译器，把在左边面板任意输入的CoffeeScript代码编译为JavaScriprt后显示在右边的面板中。

你也可以使用coffee2js项目把JavaScript转变为CoffeeScirpt。这在把JavaScript项目迁移到CoffeeScript上时尤其有用。

实际上，你自己都可以使用基于浏览器的CoffeeScirpt编译器，只需要在页面中包含这个脚本，使用正确类型（type）的标标签记CoffeeScript脚本即可。

```
    <script src="http://jashkenas.github.com/coffee-script/extras/coffee-script.js" type="text/javascript" charset="utf-8"></script>
    <script type="text/coffeescript">
      # Some CoffeeScript
    </script>
```

显然，在生产环境中，由于会减慢客户端的运行，所以没人愿意在运行时解释执行CoffeeScript。作为替代，CoffeeScript提供了一个Node.js版的编译器来对CoffeeScript文件进行预处理。

要安装该编译器，首先必须保证你已经有了稳定可用的Node.js和npm（Node程序包管理工具）。然后你就可以使用npm来安装CoffeeScirpt了：

```
    npm install -g coffee-script
```

这同时还为你提供了一个`coffee`的可执行二进制程序，如果不用任何命令行参数而直接运行该程序，它会给你一个CoffeeScript的命令行，这个命令行你可以用来快速的运行一些CoffeeScript语句。要预处理文件的话，使用`--compile`参数：

```
    coffee --compile my-script.coffee
```

如果没有指定`--output`参数，CoffeeScript会直接将编译后的代码写入到一个同名的JavaScript文件中，本例中就是`my-script.js`。已存在该文件的话会被复写掉，因此要当心你的JavaScript文件被覆盖。使用`--help`参数可以看到一个完整的可用命令行参数列表。

就如你在之前看到的一样，CoffeeScirpt文件的默认扩展名是`.coffee`，除去其他原因之外，能让像TextMate这样的编译器能够辨认出文件中包含的是什么语言的代码从而是用相对应的高亮也是其中之一。TextMate并不包含对CoffeeScript的支持，但是你可以安装这个包来提供支持。

编译看起来既不方便又很无聊。没办法，它就是这样。我们将会学习通过自动编译的方法来解决这个问题，不过首先我们先学习一下这门语言的语法。

CoffeeScript语法

首先，在开始本章之前，我还想重申下尽管很多时候CoffeeScript的语法与JavaScript相似，但是它并不是JavaScript的超集，因此，例如`function`和`var`这类JavaScript关键字并不允许在CoffeeScript中使用。如果你正在编写CoffeeScript文件，里面必须完全是纯CoffeeScript代码，你不能把这两种语言揉到一起。

为什么CoffeeScript不是超集？阻止其成为超集最直接的原因是在CoffeeScript程序中空格是有意义的。而且，既然已经这么决定了，开发团队也帮你一干到底，以精简的名字代替JavaScript的一些关键字和特性，还为避免很多常见的bug而努力。

让我极度兴奋的是，从元的角度上来说，CoffeeScript的解释器实际上就是由CoffeeScript写成的。这看起来似乎解决了先有鸡还是先有蛋的悖论！

好了，让我们从最基本的工作开始。CoffeeScript去掉了分号，它会在编译时为你自动添加。分号在JavaScript社区中引起了大量的争论，以及背后的一些解释器怪异的行为。总之，CoffeeScript为了帮你解决这个问题，简单地从语法上的移除了分号，然后在幕后更具需要添加。

注释格式与Ruby的一致，以一个哈希字符开头。

```
# A comment
```

也支持多行注释，而且还会把多行注释添加到生成的JavaScript中。使用三个哈希字符包裹即可。

```
###
  A multiline comment, perhaps a LICENSE.
###
```

正如我简单的提过，CoffeeScript对空格是敏感的。实际说来，就是你可以使用制表符来替换花括号（{}）。这受到了Python语法的影响，而且还能确保你的脚本有一个清晰的格式，否则连编译都通不过。

变量与作用域

CoffeeScript修复了JavaScript中一个最让人头疼的问题——全局变量。在JavaScript中，一不小心的话，就很容易在定义变量时遗漏`var`关键字导致产生全局变量。CoffeeScript通过简单的剔除全局变量来解决这个问题。在背后，CoffeeScript使用一个匿名函数把所有脚本都包裹起来，将其限定在局部作用域中，并且为所有的变量赋值前自动添加`var`。比如，下面是在CoffeeScript中简单的定义一个变量：

```
myVariable = "test"
```

注意示例代码右上角的深灰色小方块。单击它，代码就会在CoffeeScript和编译后的JavaScript之间来回切换。这是在页面加载是输出的，所以你放心，编译结果是准确的。

如你所见的那样，变量赋值被限定在局部作用域中，不小心创建全局变量是不可能的。CoffeeScript还更进了一步，让覆盖一个高一级的变量也很困难。这大量的减少了程序员会在JavaScript中犯的常见的错误。

然而，有时候全局变量还是有用的。你可以通过直接给全局对象（浏览器中的window）赋值来获得全局变量，也可以通过下面这种模式。

```
exports = this
exports.MyVariable = "foo-bar"
```

在顶级作用域中，this就相当于全局对象，你可以创建一个局部变量exports让阅读你代码的人能够分清楚哪个是脚本创建的全局变量。而且，这还能为支持CommonJS模块铺平了道路，这在本书的后面会做介绍。

函数

CoffeeScript移除了冗长的function语句，以瘦箭头->替之。函数可以是一行也可以是多行。函数的最后一个表达式会作为隐式的返回值。换句话说，你不再需要使用return关键字，除非你想早一点从函数中返回。

记住这点，让我们看一个例子：

```
func = -> "bar"
```

结合着编译后的JavaScript你会发现，->被转成了一个function表达式，并且"bar"被自动的返回了。

前面也说了，没有理由阻止我们使用多行的函数，只需要适当地缩进函数体即可：

```
func = ->
  # An extra line
  "bar"
```

函数参数

如何指定参数？CoffeeScript允许你通过在箭头前面的括号中指定参数。

```
times = (a, b) -> a * b
```

CoffeeScript还支持默认参数，例如：

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

你还可以使用参数槽（splats）接收多个参数，使用...表示：

```
sum = (nums...) ->
  result = 0
  nums.forEach (n) -> result += n
  result
```

在上面的例子中，nums是一个包含传递给函数全部参数的数组。它不是一个arugments对象，而是一个真实的数组对象，这样的话在你想操作它的时候就不需要先使用Array.prototype.splice或者jQuery.makeArray()了。

```
trigger = (events...) ->
  events.splice(1, 0, this)
  this.constructor.trigger.apply(events)
```

函数调用

在JavaScript中，可以通过括弧()、apply()和call()来调用函数。然而，像Ruby一样，如果函数被至少一个参数跟着的话，CoffeeScript会自动的调用这个函数。

```
a = "Howdy!"

alert a
# Equivalent to:
alert(a)

alert inspect a
# Equivalent to:
alert(inspect(a))
```

尽管括号不是必须的，但是在难以分清谁是被调用的函数哪些是参数时，我推荐还是用上括号。上一个inspect的示例中，我真心建议你至少使给inspect的调用加上括号。

```
alert inspect(a)
```

如果在调用一个函数时你没有传递参数，CoffeeScript就没有办法判断出你打算调用这个函数，还是只是把它当作一个变量。从这点来看，CoffeeScript的行为与Ruby有些差异，后者总是会调用引用函数的变量，CoffeeScript更像Python。这已经变成了我的CoffeeScript程序中常见的错误。因此，在你打算无参数调用函数时多留个心眼，别忘了加上括号。

函数上下文

在JavaScript上下文会频繁的变化。尤其是在回调函数中，CoffeeScript为此提供了一些辅助。其中之一就是->的变种胖箭头的函数=>

使用胖箭头代替普通箭头是为了确保函数的上下文可以绑定为当前的上下文。例如：

```
this.clickHandler = -> alert "clicked"
element.addEventListener "click", (e) => this.clickHandler(e)
```

你之所以要这样做的原因是，来自addEventListener的回调函数会以element为上下文被调用，也就是说，this就相当于这个元素。如果你想让this等于当前上下文，除了使用self=this，胖箭头也是一种方式。

这中绑定的思想与jQuery的 proxy()或者ES5's的bind()函数是类似的概念。

对象字面量与数组定义

就如在JavaScript中一样，可以使用一对大括号以及键/值来明确定义对象字面量。然而，与函数调用类似，CoffeeScript使得可以省略括号。事实上，你还可以使用缩进和换行来代替起分割作用的逗号。

```
object1 = {one: 1, two: 2}

# Without braces
object2 = one: 1, two: 2

# Using new lines instead of commas
object3 =
```

```
    one: 1
    two: 2

User.create(name: "John Smith")
```

同样的，数组可以使用空格来代替分隔作用的逗号，但是方括号（[]）还是需要的。

```
array1 = [1, 2, 3]

array2 = [
  1
  2
  3
]

array3 = [1,2,3,]
```

像你在上例看到的那样，CoffeeScript还能去掉array3末尾多余的逗号，这也是一个常见的跨浏览器错误源。

流程控制

这种可省略括号的便捷方式延续到了CoffeeScript中的if和else关键字。

```
if true == true
  "We're ok"

if true != true then "Panic"

# Equivalent to:
#  (1 > 0) ? "Ok" : "Y2K!"
if 1 > 0 then "Ok" else "Y2K!"
```

如你所见，在单行的if语句中，你需要使用then关键字，这样CoffeeScirpt才能明白执行体从什么地方开始。CoffeeScript并不支持条件运算符，作为替代你应该使用单行的if/else语句。

CoffeeScript还支持一项Ruby的特性，即运行在if语句前使用前缀表达式。

```
alert "It's cold!" if heat < 5
```

你还可以使用not关键字来代替感叹号（!）来做取反操作。由于很容易错过感叹号，这在某些时候能让你的代码有更强的可读性。

```
if not true then "Panic"
```

在上面的例子中，我们还可使使用CoffeeScript的unless关键字，即if的否定。

```
unless true
  "Panic"
```

与not类似的风格，CoffeeScript还为大家提供了is语句，编译过去就是===。

```
if true is 1
  "Type coercion fail!"
```

你可以使用isnt代替is not。

```
if true isnt true
  alert "Opposite day!"
```

在上面的例子中你可以已经注意到了，CoffeeScript会把==操作符转化为===,把!=转化为!==。这是这门语言中我最喜欢的一个特性，也是最简单的一个。那这背后有什么原因呢？坦白的讲JavaScript强制的类型转换有点奇怪，并且等于操作符为了比较它们会强制类型转换，这会导致很多令人迷惑的行为和很多的bug。在第7章中还是对此有有更多的讨论。

字符串插值法

CoffeeScript将Ruby风格的字符串插值法引入到了JavaScript中。在双引号的字符串中可以包含`#{}`标记，这些标记中可以包含被插入到字符串中的表达式。

```
favourite_color = "Blue. No, yel..."
question = "Bridgekeeper: What... is your favourite color?
            Galahad: #{favourite_color}
            Bridgekeeper: Wrong!
            "
```

就上例所示，多行字符串是允许的，不需要在没一行前添加+。

循环和列表解析

JavaScript中的数组迭代使用一种相当古老的语法，看上去更像一个类似于C之类的老语言，而不是现代的面向对象的语言。ES5引入`forEach()`函数来稍微改善了下这种情况，但是这样的话每次迭代都需要调用一次函数，因此运行速度会变慢。再一次，CoffeeScript给出一种漂亮的语法拯救了我们：

```
for name in ["Roger", "Roderick", "Brian"]
  alert "Release #{name}"
```

如果你需要知道当前迭代索引的话，只需要再多传一个参数：

```
for name, i in ["Roger the pickpocket", "Roderick the robber"]
  alert "#{i} - Release #{name}"
```

使用前缀的形式你可以一行代码完成迭代。

```
release prisoner for prisoner in ["Roger", "Roderick",
"Brian"]
```

就如Python的推导式一样，你可以过滤它们：

```
prisoners = ["Roger", "Roderick", "Brian"]
release prisoner for prisoner in prisoners when prisoner[0] is
"R"
```

你可以使用推导式来迭代对象的全部属性，不过要使用`of`代替`in`关键字。

```
names = sam: seaborn, donna: moss
alert("#{first} #{last}") for first, last of names
```

唯一CoffeeScript暴露出来的底层循环语法是`while`循环。它与原JavaScript中`while`循环的行为差不多，只是包含了已添加的优点，它能返回一个结果数组。看起来像`Array.prototype.map()`函数。

```
num = 6
```

```
minstrel = while num -= 1
  num + " Brave Sir Robin ran away"
```

数组

说到使用区间来分割数组，CoffeeScript是受到了Ruby的影响。使用两个数字来定义区间，分别代表区间的第一个和最后一个位置。这两个数字之间使用`..`或`...`来分隔。如果区间之前没有任何东西，CoffeeScript会将其转换为一个数组。

```
range = [1..5]
```

然而，如果区间被指定到一个变量之后，CoffeeScript则会将其转换为一个`slice()`调用。

```
firstTwo = ["one", "two", "three"][0..1]
```

在上面的例子中，区间会返回一个只包含原始数组的前两个元素的新的字符串。你也可以使用同样的语法来把数组中的某个片段替换为其他的数组。

```
numbers = [0..9]
numbers[3..5] = [-3, -4, -5]
```

更棒的是，JavaScript还能让你在字符串上调用`slice()`，因此你可以在字符串上使用区间来获得一个新的子字符串。

```
my = "my string"[0..2]
```

在JavaScript中检测数组中是否存在某个值是一件麻烦事，特别是`indexOf()`并不是所有的浏览器都支持（IE，我说的就是你！）。CoffeeScript使用`in`操作符来解决这个问题，例如：

```
words = ["rattled", "roudy", "rebbles", "ranks"]
alert "Stop wagging me" if "ranks" in words
```

别名和存在操作符

CoffeeScript采用了一些有用的别名来减少输入量。其中一个就是`@`，是`this`的别名。

```
@saviour = true
```

另外一个是`::`，`prototype`的别名。

```
User::first = -> @records[0]
```

在JavaScript中使用`if`来做`null`检查是很常见的，但是其中有几个陷阱，空字符串和零都被强制转化为`false`，这往往会让你犯错。CoffeeScript存在操作符`?`只会在变量为`null`或者`undefined`的时候会返回真，与Ruby的`nil?`类似。

```
praise if brian?
```

你还能用它来替换`||`操作符：

```
velocity = southern ? 40
```

如果你在访问属性之前进行`null`检查，你可以把存在操作符放在它左边来跳过检查。这与Actice Support的try方法比较类似。

```
blackKnight.getLegs()?.kick()
```

你能够用同样的方法检查一个属性是否是函数，是否可以调用，把存在操作符放在括号之前就行。如果属性不存在，或者不是一个函数，则就不会被调用。

```
blackKnight.getLegs().kick?()
```

类

JavaScript中的类对于纯粹主义者来说有点像大蒜对Dracula的感觉，诚实点吧，如果你喜欢那样的方式，那你应该不会想读这本关于CoffeeScript的书。可事实是类在JavaScript中就如在其他语言中一样地有用。因此，CoffeeScript为此提供了一个很棒的抽象。

在背后，CoffeeScript使用JavaScript原生的原型来产生类，为静态变量继承以及上下文持久化添加了一点语法糖，而暴露给开发者的全部只有一个`class`关键字。

```
class Animal
```

在上例中，`Animal`是类的名字，而且也是你可以用来创建实例的合成的变量的名字。CoffeeScript在背后使用了一个构造函数，这意味着你可以使用`new`关键字来实例化变量。

```
animal = new Animal
```

定义构造函数（在实例化前调用的函数）很简单，使用名为`constructor`的函数即可。这与Ruby的`initialize`或者Python的`__init__`类似。

```
class Animal
  constructor: (name) ->
    @name = name
```

实际上，CoffeeScript为设置实例属性值的常见模式提供了一种简写的方式。如果在参数前加一个`@`，CoffeeScript就会在构造函数中自动地把参数设置为实例的属性。而且，这中简写对于类之外的普通函数同样适用。下面的例子与我们手动设置实例属性的上一例子等价。

```
class Animal
  constructor: (@name) ->
```

如你所愿，每个实例化传入的参数都被代理给了构造函数。

```
animal = new Animal("Parrot")
alert "Animal is a #{animal.name}"
```

实例属性

可以非常直接地为类添加实例属性，与为对象添加属性的语法一样。只需要在类体内对属性采用合理的缩进即可。

```
class Animal
  price: 5

  sell: (customer) ->

animal = new Animal
animal.sell(new Customer)
```

在JavaScript中上下文变化很频繁，在上一章语法中我们讨论过CoffeeScript如何通过胖箭头函数（`=>`）来让`this`值锁定到某个特定的上下文中。这样无论这个函数在什么上下文中被调用，都保证该函数总是在其创建时的上下文中执行。CoffeeScript把胖箭头语法扩展到类中，因此在实例方法上使用胖箭头你就能确保方法能在正确的上下文中执行——`this`总是等于当前的实例对象。

```
class Animal
```

```
    price: 5

    sell: =>
      alert "Give me #{@price} shillings!"

animal = new Animal
$("#sell").click(animal.sell)
```

如上例所示，这在事件回调是尤其有用。正常情况下`sell()`函数会以`#sell`元素为上下文调用。然而，通过使用胖箭头来定义`sell()`，我们能保证能保持正确的上下文，所以`this.price`等于`5`。

静态变量

可以定义类（静态）变量吗？当然，事实证明在类的定义中，`this`引用的就是类。也就是说你可以通过直接在这个`this`上设置类属性。

```
class Animal
  this.find = (name) ->

Animal.find("Parrot")
```

实际上，如你所知，CoffeeScript使用`@`作为`this`的别名，这能让你更加便捷的定义静态属性：

```
class Animal
  @find: (name) ->

Animal.find("Parrot")
```

继承与Super

不支持继承不能算是一个完整的类实现。当然CoffeeScript不会让你失望，你可以使用`extends`关键字来使用继承。在下面的例子中，`Parrot`扩展自`Animal`，继承它的实例的所有属性，比方说`alive()`方法。

```
class Animal
  constructor: (@name) ->

  alive: ->
    false

class Parrot extends Animal
  constructor: ->
    super("Parrot")

  dead: ->
    not @alive()
```

在上例中，你注意到我们使用了`super()`关键字。在背后这会编译为对父类原型的一次函数调用，不过是以当前为上下文的。在本例中，就相当于`Parrot.__super__.constructor.call(this, "Parrot");`。实际上，这与Ruby或者Python中调用`super`的效果相同——调用被重写了的父类函数。

除非你重写了contructor，默认情况下，在一个实例被创建时CoffeeScript会调用其父类的构造器。

CoffeeSript使用原型继承来自动的从类实例上继承所有的属性，这保证了类的动态性。就算你给一个已经被子类继承了的父类添加属性，这些属性仍然可以被其子类继承过来。

```
class Animal
  constructor: (@name) ->

class Parrot extends Animal

Animal::rip = true

parrot = new Parrot("Macaw")
alert("This parrot is no more") if parrot.rip
```

值得一提的是，静态变量是直接拷贝给子类的，而不是像实例属性那样通过原型来实现。这都是JavaScript原型架构的实现细节所致，而且这是一个比较难解决的问题。

Mixins

CoffeeScript并不直接提供对Mixins的支持，理由是你自己完全可以很容易的实现。例如，下面有两个函数，extend()和include()分别会把类属性和实例属性添加到一个类中。

```
extend = (obj, mixin) ->
  obj[name] = method for name, method of mixin
  obj

include = (klass, mixin) ->
  extend klass.prototype, mixin

# Usage
include Parrot,
  isDeceased: true

(new Parrot).isDeceased
```

在继承不合适时Mixins是一种不错的在模块间共享通用逻辑的的模式。较之于继承只能实现从单一的父类继承，Mixins的优势是能够实现多个继承。

扩展类

Mixins很棒，只是看起来并不那么地面向对象。让我们把Mixins集成到CoffeeScript的类中吧。我们将会定义一个名为Module的类，然后可以继承这个类来获得对Mixins的支持。Module会有两个静态方法，@extend()和@include()，可以用它们来实现对类的静态属性和实例属性的扩展。

```
moduleKeywords = ['extended', 'included']

class Module
  @extend: (obj) ->
    for key, value of obj when key not in moduleKeywords
      @[key] = value
```

```
        obj.extended?.apply(@)
        this

      @include: (obj) ->
        for key, value of obj when key not in moduleKeywords
          # Assign properties to the prototype
          @::[key] = value

        obj.included?.apply(@)
        this
```

这里有个小技巧，当使用Mixins来扩展一个类时，`moduleKeywords`变量能为我们提供了回调支持。让我们实际看一下`Module`是如何工作的：

```
classProperties =
  find: (id) ->
  create: (attrs) ->

instanceProperties =
  save: ->

class User extends Module
  @extend classProperties
  @include instanceProperties

# Usage:
user = User.find(1)

user = new User
user.save()
```

如你所见，我们为`User`类添加了`find()`和`create()`静态属性，还添加了`save()`实例属性。

既然在扩展模块后我们还可以使用回调函数，于是可以快捷地处理类属性和实例属性。

```
ORM =
  find: (id) ->
  create: (attrs) ->
  extended: ->
    @include
      save: ->

class User extends Module
  @extend ORM
```

超简单也超优雅，有没有！

CoffeeScript惯用法

每个语言都有自己的惯用法和最佳实践，CoffeeScript也不例外。本章将为你揭示这些常规的东西，并且为你指出一些JavaScript到CoffeeScript的变化，以便你对这门语言有个感性的认识。

Each

在JavaScript中我们既可以使用新加入的`forEach()`也可以使用老的C语言风格的`for`循环来迭代一个数组。如果你打算使用一些在ECMAScript 5 提到的JavaScript新特性的话，我推荐你把这个 shim （薄层）来模拟这些特性以便支持老的浏览器。

```
for (var i=0; i < array.length; i++)
  myFunction(array[i]);

array.forEach(function(item, i){
  myFunction(item)
});
```

尽管`forEach()`语法非常简洁易读，但有个缺点是在每次数组迭代时都需要调用回调函数，因此它比等价的`for`循环要慢。让我看一下这CoffeeScirpt中又是什么样子。

```
myFunction(item) for item in array
```

这种语法易读简洁（我想你也这么认为），而且更棒的是这在背后会被编译为`for`循环。换句话说CoffeeScript的语法提供了`forEach()`的便捷，但是没有性能的损耗，也不需要shim的辅助。

Map

与`forEach()`相同，ES5包含了一个比经典`for`循环在语法上更加简洁的函数，名为`map()`。不过它与`forEach()`有一样需要注意的地方，其运行速度仍会受到函数调用的拖累。

```
var result = []
for (var i=0; i < array.length; i++)
  result.push(array[i].name)

var result = array.map(function(item, i){
  return item.name;
});
```

如我们在语法这一章所说，可以使用列表解析获得与`map()`同样的行为。注意最好使用括号把列表解析包裹起来，以便能够完全地确保列表解析返回你所想要的东西——映射后的数组。

```
result = (item.name for item in array)
```

筛选

ES5还提供了工具函数 `filter()` 来过滤数组：

```
    var result = []
    for (var i=0; i < array.length; i++)
      if (array[i].name == "test")
        result.push(array[i])

    result = array.filter(function(item, i){
      return item.name == "test"
    });
```

CoffeeScript的基础语法使用when关键字通过一个比较来过滤数组项。在背后会产生一个for循环，整个运行过程都包裹在一个匿名函数中，以防止作用域泄漏或变量冲突。

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

别忘了使用括号，否则result是数组的最后一项。
CoffeeScript的列表解析是如此的灵活，允许你如下例这样做出强大地选择：

```
    passed = []
    failed = []
    (if score > 60 then passed else failed).push score for score
in [49, 58, 76, 82, 88, 90]

    # Or
    passed = (score for score in scores when score > 60)
```

如果列表解析太长，你可以将它们分割成多行。

```
    passed = []
    failed = []
    for score in [49, 58, 76, 82, 88, 90]
      (if score > 60 then passed else failed).push score
```

包含

通常使用indexOf()来测试一个数组中是否包含某个值。不过真让人惊讶，Internet Explorer并没有实现该函数，这还要做一些兼容。

```
    var included = (array.indexOf("test") != -1)
```

于此CoffeeScript有一个非常给力的替代方法，Python程序员一定很熟悉，名叫in。

```
    included = "test" in array
```

在背后，CoffeeScript使用的是 Array.prototype.indexOf()，必要的话提供shim方法来检测数组中是否有某个特定值，不幸的的是同样的in语法并不能在字符串中工作。我们退回去使用indexOf()函数，查看其返回值是否是负值：

```
    included = "a long test string".indexOf("test") isnt -1
```

或者更好一点，借助于位操作符我们就不用与-1进行比较了。

```
    string   = "a long test string"
    included = !!~ string.indexOf "test"
```

属性迭代

在JavaScript中，你应该使用in操作符来迭代属性集，例如：

```
var object = {one: 1, two: 2}
for(var key in object) alert(key + " = " + object[key])
```

然而，如你在上一小节所知，CoffeeScript已把in关键字留给了数组用。作为替代，该操作符更名为of，可以像下面这样用：

```
object = {one: 1, two: 2}
alert("#{key} = #{value}") for key, value of object
```

如你所见，你可以同时指定属性名和属性值，非常方便。

Min/Max

这个技巧虽然不是CoffeeScript的专利，但是我觉得它非常有用，值得一提。Math.max 和Math.min接受多个参数，因此你可以简单的地使用...来向它们传递数组，从中检索出最大值和最小值。

```
Math.max [14, 35, -7, 46, 98]... # 98
Math.min [14, 35, -7, 46, 98]... # -7
```

请注意，这个技巧对于超大的数组也会失败，因为浏览器对传递个函数的参数数量有限制。

多个参数

在上面的Math.max示例中，我们使用...来结构数组作为多个参数传递给max方法。在背后，CoffeeScript将其转化为一个使用appply()的函数调用，以确保数组能够作为多个参数传递给max。在别的地方也可以使用这个特性，比方说代理函数：

```
Log =
  log: ->
    console?.log(arguments...)
```

或者在参数继续传递下去之前，修改参数：

```
Log =
  logPrefix: "(App)"

  log: (args...) ->
    args.unshift(@logPrefix) if @logPrefix
    console?.log(args...)
```

不过请记住，CoffeeScript会自动把函数的调用上下文设置为调用它的对象。在上例中，就是console对象，如果你想设置特殊的上下文，那你需要手动的调用apply()方法。

And/or

CoffeeScript编程风格推荐使用or代替||，使用and代替&&。我知道为什么，因为前者看起来更直观。不过，这两种编程风格产生的结果都一样。

偏爱英语风格的代码的话，也可以使用is代替==，isnt代替!=。

```
string = "migrating coconuts"
string == string # true
string is string # true
```

CoffeeScript还有另外一个非常好的扩展，Ruby程序员可将其看作像是||=这样的模式：

```
hash or= {}
```

如果hash求值为false，则把它设置为一个空对象。在这里需要注意，表达式0、""和null都会被当作false。如果这并不是你想要的，那你应该使用CoffeeScript的存在操作符，这样只有hash是undefined或者null时才会触发。

```
hash ?= {}
```

解构赋值

解构赋值对任意深度嵌套的数组或对象都适用，方便从嵌套的属性中抽取值。

```
someObject = { a: 'value for a', b: 'value for b' }
{ a, b } = someObject
console.log "a is '#{a}', b is '#{b}'"
```

这在Node程序中引入模块时尤其有用：

```
{join, resolve} = require('path')

join('/Users', 'Alex')
```

其他类库

既然所有的东西都会编译为JavaScript，那么使用其他类库与调用CoffeeScript类库的函数并没有什么差别。在CoffeeScript中使用 jQuery显得非常优雅，因为jQuery的API中有很多回调函数。

```
# Use local alias
$ = jQuery

$ ->
  # DOMContentLoaded
  $(".el").click ->
    alert("Clicked!")
```

既然CoffeeScript编译输出的所有代码都被包裹在一个匿名函数中，因此我们可以使用一个局部变量$来代替jQuery。就算在jQuery的no confict模式或者$被重定义的情况下，我们的脚本能按预想的一样正常工作。

私有变量

CoffeeScript中的do关键字能够让我们立即运行函数，这是一种非常有效的包装作用域和受保护变量的方式。在下面的例子中，我在被do立刻调用的匿名函数的上下文中定义了一个变量classToType。该匿名函数返回了另外一个匿名函数，它才是type最终的值。既然classType是在一个不保存引用的上下文中，因此在外部作用域中不可访问。

```
# Execute function immediately
type = do ->
  classToType = {}
  for name in "Boolean Number String Function Array Date
RegExp Undefined Null".split(" ")
    classToType["[object " + name + "]"] = name.toLowerCase()

  # Return a function
  (obj) ->
    strType = Object::toString.call(obj)
```

```
        classToType[strType] or "object"
```

换句话说，`classToType`是完全私有的，并且在匿名函数执行完毕之后就不能在外面作用域中引用它了。这是一种非常好的包装作用域和变量的办法。

CoffeeScript的自动编译

使用CoffeeScript的一个问题是，它会在你和JavaScript之间多加了一个层，当CoffeeScript文件频繁变更时你还需要手工编译它们。

如第一章所说，我们可以使用`coffee`命令来编译CoffeeScript文件：

```
coffee --compile --output lib src
```

在上面的例子中，事实上，在`src`中的`.coffee`文件会被编译为JavaScript并且输出到`lib`目录中。这样编译看起来有点无聊，因此让我们看一下如何自动地编译。

Cake

Cake是一个超级简单的与 Make和 Rake类似的构建工具。该库被捆绑在`coffee-script`的npm安装包中，可以通过名为`cake`的可执行程序来使用。

你可以使用在一个名为`Cakefile`的文件中使用CoffeeScript来定义任务。Cake可以去读它，而且可以在当前目录中通过运行`cake [task] [options]`来调用它。单单输入`cake`可以显示一个任务项以及参数的列表。

可以使用`task()`函数来定义任务项，给它传递一个名字、参数以及回调函数即可。例如，创建一个名为`Cakefile`的文件，还有两个目录——`lib`和`src`。把下面的代码添加到Cake文件中；

```
fs = require 'fs'

{print} = require 'sys'
{spawn} = require 'child_process'

build = (callback) ->
  coffee = spawn 'coffee', ['-c', '-o', 'lib', 'src']
  coffee.stderr.on 'data', (data) ->
    process.stderr.write data.toString()
  coffee.stdout.on 'data', (data) ->
    print data.toString()
  coffee.on 'exit', (code) ->
    callback?() if code is 0

task 'build', 'Build lib/ from src/', ->
  build()
```

在上面的例子中，我们定义了一个名为`build`的任务项，该任务项可以通过运行`cake build`来调用。这运行的是与之前例子一样的命令。将`src`内的CoffeeScript文件编译为JavaScript放到`lib`中。你现在可以如往常一样在HTML中引用`lib`中的JavaScript文件了。

```
<!DOCTYPE html>
<html>
 <head>
      <meta charset=utf-8>
      <script src="lib/app.js" type="text/javascript"
charset="utf-8"></script>
```

```
  </head>
  <body>
  </body>
</html>
```

我们在CoffeeScript代码变了之后还需要手动的运行`cake build`命令，这很不理想。幸运的是，`coffee`命令接受另外一个`--watch`参数，指示它监视一个目录的变化并且按需重新编译。让我们用它来定义另外一个任务项：

```
task 'watch', 'Watch src/ for changes', ->
  coffee = spawn 'coffee', ['-w', '-c', '-o', 'lib', 'src']
  coffee.stderr.on 'data', (data) ->
    process.stderr.write data.toString()
  coffee.stdout.on 'data', (data) ->
    print data.toString()
```

如果一个任务项依赖了另一个，你是使用`invoke(name)`来运行其他任务项。让我添加一个实用懂得任务项到我们的`Cakefile`中，该任务首先打开`index.html`然后就开始监视源文件的改变了。

```
task 'open', 'Open index.html', ->
  # First open, then watch
  spawn 'open', 'index.html'
  invoke 'watch'
```

你可以使用`option()`函数来给你的任务项定义参数，它接受一个短名、长名和描述三个参数。

```
option '-o', '--output [DIR]', 'output dir'

task 'build', 'Build lib/ from src/', ->
  # Now we have access to a `options` object
  coffee = spawn 'coffee', ['-c', '-o', options.output or
'lib', 'src']
  coffee.stderr.on 'data', (data) ->
    process.stderr.write data.toString()
  coffee.stdout.on 'data', (data) ->
    print data.toString()
```

如汝所见，任务项上下文可获一包含用户指定的所有参数的`options`对象。如若我等执行`cake`之令而不带任何参数，则所有任务项与参数列之。

Cake是一种自动化常见任务项（例如编译CoffeeScript）的有效的方式，避免了bash或者Makefiles的麻烦。 Cake的源码也值得一看，上面不但展示了CoffeeScript强大的表现力，旁边还有一份漂亮的由代码注释生成的文档。

Server端的支持

对于静态的站点来说使用Cake就已经足够，但是对于动态的站点来说，我们就需要把CoffeeScript集成到请求/回应的生命周期中，对于那些比较流行的后端语言或者开发框架（例如Rails和Django）来说已经存在各种各样的集成方案了。

Rails 3.1通过Sprockets & the asset pipeline实现了对CoffeeScript的支持。把你的CoffeeScript放到`app/assets/javascripts`中，Rails就能够智能地按照请求需要对其进行预编译。可以使用特殊的注释来指示联合和打包JavaScript和CoffeeScript文件，这就意味着你可以通过一次请求获取程序的全部JavaScript。当发布到生成环境时，Rails会把编译后的文件输出出来，以保证能够缓存文件快速提供服务。

Rack服务器是其他可选方案之一，比如说37signal的Pow 和Joshua Peek的Nack都是。如果你的程序不需要Rails其他特性或者相关东西的话我力荐你使用它们，

Django通过特殊的模板标签也能支持CoffeeScript。而且同时支持行内代码或者外联文件。

当需要编译CoffeeScript时，Ruby和Python在后台都是通过管道将其输出到Node和CoffeeScript库中，因此你在开发时需要事先安装好这些类库。如果你直接使用Node为你的站点开发后端程序，就非常容易集成CoffeeScript了，而且你还可以同时使用它来开发的前后台的代码。我们将在下一章介绍更多相关的东西，使用Stitch来为我们的客户端CoffeeScript提供服务。

创建应用程序

既然你基本已经了解了CoffeeScript的语法，那现在让我们来探索下如何使用它来真枪实弹的创建应用程序。本节内容对所有的CoffeeScript开发者都有用，无论你是新人还是老手，实际上，这对只写JavaScript的人来说也非常有用。

不知什么原因，当开发者构建JavaScript客户端程序时，往往不会使用可靠的测试过的模式或约定，这就导致大量的代码像意大利面条一样无法维护。程序架构的重要性再怎么强调都不过分。如果你所写的JavaScript或者CoffeeScript不是表单验证这么简单的话，你需要使用某种程序架构，比方说MVC。

创建可维护的大型应用程序的秘诀就是不要做大型程序。换句话说，就是创建一系列低耦和的模块。程序逻辑越通用越好，尽量抽象出来。最终，将你的逻辑分别拆分为视图（views）、数据模型（models）和控制器（controllers）（MVC）。如何实现MVC超出了本章的范围，需要的话我建议你到JavaScript Web Applications翻看我的书，或者使用像Backbone 或者 Spine这类框架。现在先不管那些，在这里我们使用CommonJS模块来构建应用程序。

结构 & CommonJS

那什么CommonJS模块到底是什么呢？好的，如果你用过NodeJS 的话，你大概没有意识到你在使用CommonJS模块。CommonJS模块起初是为了编写服务器端JavaScript类库而开发的，企图用它来解决模块加载、命名空间和作用域的问题。还有一个通用的形式用来兼容所有的JavaScript实现。目标是写一个给Rhino 的类库也可以用于Node。终于这种思想被移植到了浏览器中，而且现在我们有像RequireJS和Yabble这样好的类库来构建模块化的客户端。

实际上，模块能保证你的代码运行在局部作用域中（代码分装），你可以通过`require()`函数来载入模块，而且通过`module.exports`来暴露模块。让我们更加深入的研究下。

导入文件

你可是使用`require()`来载入其他模块和类库。只给它传递一个模块名即可，而且如果该模块正加载目录中，`require()`会返回一个代表该模块的对象。例如：

```
User = require("models/user")
```

同步的`require`是一个颇具争议的问题，但是主流的类库加载器和CommonJS的预案中已经解决了这个问题。如果你不想使用我下面推荐的Stitch而想另走一条路的话，你可能需要看一下它。

暴露属性

默认情况下，模块不会暴露任何属性，因此模块内的东西对于`require()`调用来说不可见。如果你想访问模块的某个属性，你需要将它挂到`module.exports`：

```
# random_module.js
module.exports.myFineProperty = ->
  # Some shizzle
```

现在，`require`这个模块的时候`myFineProperty`就会暴露出来：

```
    myFineProperty = require("random_module").myFineProperty
```

使用Stitch打包

把你的代码格式化为CommnonJS模块很不错，但是如何让这些模块在客户端也能工作呢？我采用Stitch这个不太有名的类库作为解决方案。Stitch的作者是Sam Stephenson，其思想来自于Prototype.js，非常优雅的解决了模块的问题，真让我兴奋呀！Stitch简单将所有的JavaScript文件打包到一起，巧妙的将它们包裹在CommonJS中，而不是尝试动态处理依赖。噢，我差点忘记说，它还能编译CoffeeScript、JS模板、LESS CSS和Sass。

首先，如果你必须安装 Node.js和npm，如果还没有安装的话。在本章中我们要用到它们。

现在，创建我们的程序结构。如果你正在使用Spine，那你可以使用Spine.App自动生成，否则你需要手动的创建。我通常把全部的程序代码放到`app`目录下，`lib`存放通常的类库，然后包括其他一些像静态资源等等放到`public`目录中。

```
    app
    app/controllers
    app/views
    app/models
    app/lib
    lib
    public
    public/index.html
```

接着，为了启动我们的Stitch服务，让我们创建一个名为`index.coffee`的文件，添加如下脚本：

```
    require("coffee-script")
    stitch  = require("stitch")
    express = require("express")
    argv    = process.argv.slice(2)

    package = stitch.createPackage(
      # Specify the paths you want Stitch to automatically bundle
up
      paths: [ __dirname + "/app" ]

      # Specify your base libraries
      dependencies: [
        # __dirname + '/lib/jquery.js'
      ]
    )
    app = express.createServer()

    app.configure ->
      app.set "views", __dirname + "/views"
      app.use app.router
      app.use express.static(__dirname + "/public")
      app.get "/application.js", package.createServer()

    port = argv[0] or process.env.PORT or 9294
```

```
    console.log "Starting server on port: #{port}"
    app.listen port
```

你会发现我们依赖了一些类库：coffee-scirpt、stitch和express。我们需要创建一个package.json文件，列出这些依赖类库以便npm可以将它们打包到一起。我们的./package.json看起来像下面这样：

```
    {
      "name": "app",
      "version": "0.0.1",
      "dependencies": {
        "coffee-script": "~1.1.2",
        "stitch": "~0.3.2",
        "express": "~2.5.0",
        "eco": "1.1.0-rc-1"
      }
    }
```

然后让我使用npm安装这些依赖：

```
    npm install .
    npm install -g coffee-script
```

好，我们就要完成了，现在运行：

```
    coffee index.coffee
```

但愿你的stitch服务器已经运行起来了，让我继续在app目录中添加一个app.coffee脚本来测试下。这个就是将要引导启动我们程序的文件。

```
    module.exports = App =
      init: ->
        # Bootstrap the app
```

现在让我们创建主页index.html，如果我们创建的是单页面程序，那它将是唯一一个我们会访问的页面。这是一个静态资源，因此将其放在public目录下。

```
    <!DOCTYPE html>
    <html>
    <head>
      <meta charset=utf-8>
      <title>Application</title>
      <!-- Require the main Stitch file -->
      <script src="/application.js" type="text/javascript"
charset="utf-8"></script>
      <script type="text/javascript" charset="utf-8">
        document.addEventListener("DOMContentLoaded", function(){
          var App = require("app");
          App.init();
        }, false);
      </script>
    </head>
    <body>
    </body>
    </html>
```

当页面加载完成，我们的*DOMContentLoaded*事件回调函数会require`app.coffee`脚本（它已经被自动编译好了），然后调用`init()`函数。其实就是这样，我们获取了一个CommonJS模块并且运行了它，就如一个HTTP服务器和CoffeeScript编译器一样。假如，我们想包含一个模块，只需调用`require()`即可。让我们创建一个新的类，`User`，在`app.coffee`中引用它：

```
# app/models/user.coffee
module.exports = class User
  constructor: (@name) ->

# app/app.coffee
User = require("models/user")
```

JavaScript模板

如果你把逻辑都放到了客户端，那你就必须使用某种模板引擎。JavaScript模板除了它运行在客户端之外,与服务器端的模板引擎非常相似，比方说Ruby的ERB或者Python的文本插值。有很多模板类库，因此我鼓励你进行研究将它们找出来。默认地，Stitch预置了对Eco的支持。

JavaScript模板与服务端模板非常相似。你可以混合使用模板标签和HTML，在模板绘制过程中这些标签会被求值被替换。Eco模板最好的地方在于，它们可以直接使用CoffeeScript。

例如：

```
<% if @projects.length: %>
  <% for project in @projects: %>
    <a href="<%= project.url %>"><%= project.name %></a>
    <p><%= project.description %></p>
  <% end %>
<% else: %>
  No projects
<% end %>
```

如你所见，模板语法非常明了。只需使用`<%`标签来执行表达式，`<%=`标签输出表达式的值。下面是一个部分模板标签的列表：

- `<% expression %>`
  对一个CoffeeScript表达式求值，但不输出其返回值

- `<%= expression %>`
  对一个CoffeeScript表达式求值，转义返回值，并将其输出。

- `<%- expression %>`
  对一个CoffeeScript表达式求值，不转义，直接输出返回值。

你可以在模板标签中使用任意的CoffeeScript表达式，但是有一点需要注意，就是虽然CoffeeScript对空格敏感，但是Eco模板不敏感。因此，Eco模板标签在开始一个CoffeeScirpt缩进块时必须使用一个冒号作为后缀。然后使用一个特殊的标签`<% end %>`来表示缩进块结束。例如：

```
<% if @project.isOnHold(): %>
  On Hold
<% end %>
```

if和end并非要写成多行：

```
<% if @project.isOnHold(): %> On Hold <% end %>
```

而且你也可以使用if单行后缀表达式来实现：

```
<%= "On Hold" if @project.isOnHold() %>
```

现在我们已经学习了语法，让我们在views/users/show.eco定义一个Eco模板吧：

```
<label>Name: <%= @name %></label>
```

Stitch会自动编译我们的模板且把它们打包到application.js中。然后，在程序控制器中我们就可以require模板，就像使用一个模块一样，传入所以的数据执行它。

```
require("views/users/show")(new User("Brian"))
```

我们的app.coffee文件现在看起来像下面这样，当文档加载完毕后，我们渲染这个模板并将其添加到页面中：

```
User = require("models/user")

App =
  init: ->
    template = require("views/users/show")
    view     = template(new User("Brian"))

    # Obviously this could be spruced up by jQuery
    element = document.createElement("div")
    element.innerHTML = view
    document.body.appendChild(element)

module.exports = App
```

访问http://localhost:9294/，然后随便到处转转！希望这个教程能够教会你如何构建一个客户端的CoffeeScript程序。下一步，我推荐研究一些客户端模块，比方说Backbone或者Spine，它们能够为你提供一个基础的MVC框架，将你解放出来，做你感兴趣的事情。

附加-使用Heroku 30秒快速发布

Heroku是一个非常棒的Web主机服务提供商。它为你管理所有的服务器和扩展，让你能够做自己感兴趣的事情（创建有意思的JavaScript程序）。要让本教程的实例工作你需要一个Heroku账户，好消息是它基本套餐是免费的。之前Heroku提供的是Ruby主机服务，不过现在最近它发布了它的Cedar栈，支持Node。

首先，我们需要创建一个`Profile`文件，用它来向Heroku提供我们的程序信息。

```
echo "web: coffee index.coffee" > Procfile
```

然后，你需要为你的程序创建一个本地的git代码仓库，如果你还没有创建的话。

```
git init
git add .
git commit -m "First commit"
```

现在发布程序，我们将使用`heroku`这个gem（如果你还没安装的话，你需要先安装好）。

```
heroku create myAppName --stack cedar
git push heroku master
heroku open
```

好了，这样做就行了。没有比这还简单的Node主机的服务了。

其他类库

Stitch和Eco并不是唯一你可用于创建CoffeeScript和Node程序的类库，还有很多可作为替代的类库。

例如，当需要使用模板时，你可以使用 Mustache、Jade或者CoffeeKup（使用纯CoffeeScript写HTML）。

说到为程序提供服务，Hem是比较好的选择，支持CommonJS和NPM模块，而且还无缝的集成了CoffeeScript MVC框架Spine。node-browsify是另外一个类似的项目。如果你想使用express集成的更底层的东西，Trevor Burnham的connect-assets不错。

你可以在项目的wiki上找到一个CoffeeScript Web框架插件的完整列表。

糟粕

JavaScript是一个让人头疼的怪物，知道你不能用什么和应该用什么同样重要。孙武有言：“知己知彼，百战百胜”，这就是本章的目的，探索JavaScript不好的一面，揭示所有暗伏着准备偷袭毫无防备的开发者的怪兽。

在第一章介绍中说过，CoffeeScript的优秀不单单在于语法上的改进，还在于弥补JavaScript缺点的能力。然而，实际上CoffeeScript语句是直接翻译为JavaScript的，而不是运行在一个虚拟机或者解析器中，它不是解决所有JavaScript怪癖的银弹，还有一些地方需要你小心的处理。

首先，让我们讨论下这个语言解决的问题。

JavaScript的子集

CoffeeScript的语法只包含了JavaScript的一个子集，众所周知的精华部分，因此这样就减少了需要处理的问题。让我们用`with`作为例子来看看。`with`被“看作有害的”好久了，应该避免使用它。`with`提供了一个频繁读写对象的快捷方式。例如，像下面这样写：

```
dataObj.users.alex.email = "info@eribium.org";
```

你可以这样写：

```
with(dataObj.users.alex) {
  email = "info@eribium.org";
}
```

首先撇开我们不应该使用如此深的对象不说，这个语法非常漂亮。但必须除掉这一点。这会让JavaScript非常迷惑，它不知道你在`with`的上下文中到底要干嘛，使得它会强迫自己在任何变量查找时总是从这个特殊的对象开始。

这对性能损耗很大，而且意味着解析器必须关闭掉全部的JIT（运行时编译执行的技术）优化。还有，`with`语句并不能被像 uglify-js这样的压缩工具压缩掉。因此它在将来的JavaScript的版本中会被弃用和移除。综合考虑，避免使用它为妙，CoffeeScript更进一步，将其从自己的语法中拿掉了。换句话说，在CoffeeScript中使用`with`语句会报错。

全局变量

默认情况下，你的JavaScript程序是在全局作用域下运行的，并且任何创建出来的变量默认的也会都在全局作用域中。如果你想创建一个局部变量的话，JavaScript要求使用`var`关键字来显式的指明。

```
usersCount = 1;         // Global
var groupsCount = 2;    // Global

(function(){
  pagesCount = 3;       // Global
```

```
    var postsCount = 4;  // Local
})()
```

这是一个比较奇怪的决定，既然大部分情况下你创建的是局部而不是全局变量，那么为什么要让它成为默认的呢？事实就是这样，开发者必须记住在定义变量之前加上`var`关键字，否则会造成冲突复写了彼此的变量，引起怪异的bug。

幸运的是，CoffeeScirpt通过彻底地剔除默认赋值给全局变量来解救你。换句话说，在CoffeeScript中`var`是保留关键字，一旦使用就会触发语法错误。默认地，会隐式地产生局部变量，而且不通过显示地给`window`的属性赋值的话，没法创建全局变量。

让我们看一个CoffeeScript变量赋值的例子：

```
outerScope = true
do ->
  innerScope = true
```

编译过来就是：

```
var outerScope;
outerScope = true;
(function() {
  var innerScope;
  return innerScope = true;
})();
```

注意CoffeeScript是如何自动的在上下文中第一次使用时自动的初始化变量（使用`var`）的。同时，我们无法覆盖外层作用域的变量，你仍然可以引用和访问它们。这你就要当心了，在你编译一个嵌套比较深的函数或者类时你就要当心不要意外的重用某个外层变量的名字。例如，这样我们就不小心在一个类函数中复写掉了`package`变量。

```
package = require('./package')

class Hem
  build: ->
    # Overwrites outer variable!
    package = @hemPackage.compile()

  hemPackage: ->
    package.create()
```

偶尔需要使用全局变量时，你需要将它们设置为`window`的属性才行：

```
class window.Asset
  constructor: ->
```

通过保证显式的全局变量而不是隐式的，CoffeeScript移除了JavaScript程序中主要的bug源之一。

分号

JavaScript并没有强制要求在源码中使用分号，因此可以省略分号。但是，在后台JavaScript还是需要分号的，因此当遇到由缺少分号造成的错误时解析器会自动的插入分号。换句话说，解析器首先对一个无分号的语句进行求值，如果失败的话，加上分号再试一次。

很不幸，这是一个非常糟糕的想法，这有可能完全改变代码的行为。看下面这个例子，这段JavaScript没有问题，对吗？

```
function() {}
(window.options || {}).property
```

不对，至少对解析器来说是这样，这会导致一个语法错误。如果以一个括号开头的话，解析器就不会插入一个分号。代码会被转化成一行：

```
function() {}(window.options || {}).property
```

现在你可以看到问题在哪里，知道为什么解析器会抱怨了。当你写JavaScript的时候，你因该在语句后面加上分号。CoffeeScript通过在其语法中不使用分号来绕靠了这些麻烦。相反会在CoffeeScript编译为JavaScript时在把分号自动的插入到正确的位置上。

保留字

某些关键字被JavaScript保留下来，以便将来新版本的JavaScript使用。比如说`const`、`enum`和`class`等。在你的JavaScript程序中使用这些保留字作为变量可能会导致不可预料的错误。有些浏览器可以很好的处理它们，而其他浏览器就会卡壳。CoffeeScript通过检测你使用的是否是保留字，在需要的时候将其转义，巧妙的绕过了这个问题。

例如，让我们把`class`保留字用作对象的属性，CoffeeScript代码可能是像下面这样：

```
myObj = {
  delete: "I am a keyword!"
}
myObj.class = ->
```

CoffeeScript解析器会注意到你使用了保留字，而为你加上引号：

```
var myObj;
myObj = {
  "delete": "I am a keyword!"
};
myObj["class"] = function() {};
```

相等比较

JavaScript的非严格等于比较会产生一些莫名其妙的行为，往往能引起bug。下面这个例子来自[JavaScript Garden's equality section](#)，对这个问题进行过深入的研究。

```
""              ==   "0"           // false
0               ==   ""            // true
0               ==   "0"           // true
false           ==   "false"       // false
false           ==   "0"           // true
false           ==   undefined     // false
false           ==   null          // false
null            ==   undefined     // true
" \t\r\n"       ==   0             // true
```

这些行为背后的原因是非严格等于会自动的强制类型转换。我相信你也同意这中方式是多么的模糊不清，而且会导致不可预料的结果和bug。

解决方案就是使用严格等于操作符来替换之，它由三个等号构成`===`。其行为与正常的等于操作符一样，但是不会进行任何的类型转换。推荐使用严格等于操作符，在需要的时候在显式地进行类型转换。

CoffeeScript通过简单的将非严格的比较替换为严格的来解决这个问题，也就是把所有的`==`比较转换为`===`。在CoffeeScript中无法使用非严格的比较，如果需要的话你应该事先显式地进行类型转换。

这并不意味着在你可以忽略CoffeeScript中所有的类型转换，尤其是在流程控制检测变量是否为“真值”时。空字符串、`null`、`undefined`和数字`0`都将转换为`false`。

```
alert("Empty Array")  unless [].length
alert("Empty String") unless ""
alert("Number 0")     unless 0
```

如果你想显式的检查`null`和`undefined`的话，你可以使用CoffeeScript的存在操作符：

```
alert("This is not called") unless ""?
```

在上例中，由于空字符串不等于`null`，所以`alert()`不会被调用。

函数定义

在JavaScript有一点非常奇怪，函数可以在定义之前调用。例如，下面的代码能够正常运行，尽管`wem`实在被调用之后才被定义的：

```
wem();
function wem() {}
```

这与函数的作用域有关。在程序运行之前函数会被提升，这样的话函数在被定义的作用域的任何地方都可用，就算在源码中也可以在明确的定义之前调用它们。可问题在于，这中提升行为在浏览器上表现不一，例如：

```
if (true) {
  function declaration() {
    return "first";
  }
} else {
  function declaration() {
    return "second";
  }
}
declaration();
```

在像Firefox这类浏览器中，`declaration()`会返回`"first"`，而在其他类似于Chrome这类浏览器上会返回`"second"`，尽管看起来`else`语句永远不会被执行。

如果你想更加深入的了解函数定义式的话，你可以访问 Juriy Zaytsev的指南，他对标准进行了深入的研究。但我只想说，它们有怪异的行为，在后面的代码中就会引起问题。综合考虑下来，使用函数表达式是避免这些问题的最佳方法。

```
var wem = function(){};
```

```
wem();
```

CoffeeScript通过彻底地移除函数定义式而只是用函数表达式来解决此问题。

读取数值对象的属性

JavaScript解释器的一个瑕疵是数值对象的点表示法会被解释为一个浮点数，而不是一个属性的查找。比如，下面的JavaScript会产生一个语法错误：

```
5.toString();
```

JavaScript解释器会在点后面查找数字，但是当它碰到的是`toString()`的话就会产生一个`Unexpected token`错误。既可以使用括号也可以多加一个点来解决这个问题。

```
(5).toString();
5..toString();
```

还好CoffeeScript解析器非常聪明，当你访问一个数字的属性时，自动地使用两个点标记来处理这个问题（如上面的例子所示）。

没有修正的部分

CoffeeScript竭尽全力解决了JavaScript设计上的一些缺陷，但是也就到这一步了。如前所述，出于性能的考虑，CoffeeScript仅严格地限于静态的分析，在运行时不做任何的检查。CoffeeScript使用一个直接的源到源的编译器，其思想就是每句CoffeeScript代码都编译为等价的JavaScript语句。CoffeeScript并不针对JavaScript的关键字提供一层抽象，比方说`typeof`,因此JavaScript设计中的一些缺陷同样被带到了CoffeeScirpt中。

在上一部分中，我们看了一些CoffeeScirpt修复的JavaScript的设计缺陷，现在我们再来讨论一些CoffeeScript没能修正的JavaScript缺陷。

使用eval

虽然CoffeeScript移除了一些JavaScript的缺陷，但是还有一些恶魔般必要的特性，你需要知道它们的短处。一个合理的例子就是`eval()`函数。毋庸置疑，这个函数自有其用武之地，但你需要知道它的阴暗面，而且需要尽量避免这些阴暗面。`eval()`函数会在局部作用域中执行JavaScript代码字符串，而且像`setTimeout()`和`setInterval()`也可以接受一个字符串作为第一个参数，对该参数求值。

然而，像`with`或`eval()`会逃脱编译器的跟踪，是一个主要的性能盲点。由于编译器不到运行的时候对其内部毫不知情，如嵌入的JavaScript代码一样，编译器不能对其做任何的优化。如果你给你的输入有误，`eval`很容易为你打开代码注入攻击之门。在你使用`eval`中的99%的时间中，应该有更好更安全的替代方式（比方说方括号）。

```
# Don't do this
model = eval(modelName)

# Use square brackets instead
model = window[modelName]
```

使用typeof

`typeof`操作符是JavaScript最坑爹的设计，因为它完全就是鸡肋。事实上，它只有一个用途，就是检测一个值是否是`undefined`。

```
typeof undefinedVar is "undefined"
```

对于其他所有的类型检测，`typeof`非常失败。依赖于浏览器和实例初始化方式不同会返回不一致的结果。于此CoffeeScript也无能为力，因为这语言用的是静态检查没有运行时的类型判断。你只能自食其力了。

为了说明问题所在，这是从JavaScript Garden拿来的表格，该表格展示了这个类型检测的关键字主要不稳定的地方。

```
Value               Class      Type
-------------------------------------
"foo"               String     string
new String("foo")   String     object
1.2                 Number     number
new Number(1.2)     Number     object
true                Boolean    boolean
```

```
new Boolean(true)   Boolean    object
new Date()          Date       object
new Error()         Error      object
[1,2,3]             Array      object
new Array(1, 2, 3)  Array      object
new Function("")    Function   function
/abc/g              RegExp     object
new RegExp("meow")  RegExp     object
{}                  Object     object
new Object()        Object     object
```

你会发现，使用引号还是`String`类来定义一个字符串会影响`typeof`的结果。`typeof`应该都返回`"string"`才符合逻辑，但是对于后者它返回的却是`"object"`。很不幸，这样的不一致性只会让事情变得更糟糕。

那在JavaScript中我们如何做类型检查呢？还好有`Object.prototype.toString()`来解围。如果我们以某个特别的对象为上下文来调用该函数，它会返回正确的类型。我们所需要做的就是手动处理其返回的字符串，这样最终我们就能获得`typeof`返回的那种字符串。下面是从jQuery的`$.type`移植过来的例子：

```
type = do ->
  classToType = {}
  for name in "Boolean Number String Function Array Date
RegExp Undefined Null".split(" ")
    classToType["[object " + name + "]"] = name.toLowerCase()

  (obj) ->
    strType = Object::toString.call(obj)
    classToType[strType] or "object"

# Returns the sort of types we'd expect:
type("")          # "string"
type(new String) # "string"
type([])          # "array"
type(/\d/)        # "regexp"
type(new Date)   # "date"
type(true)        # "boolean"
type(null)        # "null"
type({})          # "object"
```

如果你想检查某个变量是否被定义，你仍然需要使用`typeof`，否则你会得到一个`ReferenceError`的错误。

```
if typeof aVar isnt "undefined"
  objectType = type(aVar)
```

或者使用更加简洁的存在操作符：

```
objectType = type(aVar?)
```

作为类型检测的替代，你通常还可以使用鸭子类型检测结合CoffeeScript的存在操作符来避免确定一个对象的类型。例如，假设我们将向一个数组中加入一个值。我们可以这么说，既然这个类数组对象实现了`push()`方法，我们应该把它当作一个数组：

```
anArray?.push? aValue
```

如果anArray是一个对象而不是一个数组，那么存在操作符会保证push()绝不会被调用。

使用instanceof

JavaScript的instanceof关键字几乎就和typeof一样不给力。理想的instanceof将比较两个对象的构造器，看其中一个是否另外一个的实例而返回真假值。实际上instanceof只在比较自定义的对象上工作正常。如果用来比较内置类型时，就像typeof一样废。

```
new String("foo") instanceof String # true
"foo" instanceof String             # false
```

而且，instanceof当比较不用浏览器里不同frame中的对象时也不能正常工作。实际上，instanceof对于自定义的对象才会返回正确的值，比方说CoffeeScript的类型。

```
class Parent
class Child extends Parent

child = new Child
child instanceof Child  # true
child instanceof Parent # true
```

确定你只在比较你自己的对象时使用它，或者坚决不用更好。

使用delete

delete关键字只能用来移除对象内部的属性。

```
anObject = {one: 1, two: 2}
delete anObject.one
anObject.hasOwnProperty("one") # false
```

用于其他地方，比方说删除一个变量或者函数，就完全不行。

```
aVar = 1
delete aVar
typeof Var # "integer"
```

这种行为非常古怪，但是你已经知道了。如果你想移除一个变量的引用，将其赋值为null即可。

```
aVar = 1
aVar = null
```

使用parseInt

如果你给JavaScript的parseInt()函数传递一个字符串而没有指明基数的话它会返回一个让你意外的值，例如：

```
# Returns 8, not 10!
parseInt('010') is 8
```

给该函数传递一个基数让它能够正常工作：

```
# Use base 10 for the correct result
parseInt('010', 10) is 10
```

这些用法CoffeeScript并不能为你做，每次当你使用`parseInt()`函数的时候别忘了给它传递一个基数。

Strict模式

Strict模式是ECMAScript5的一个新特性，它能够让你在一种严格的上下文中运行JavaScript程序或者函数。比起普通上下文，严格上下文会抛出更多的异常和警告，当开发者不遵循最佳实践，写无优化的代码或者犯常见错误时，它能给他们一些提示。换句话说，严格模式能够减少bug，提高安全性，提升性能以及消除比较难于驾驭的语言特性。这难道不好么?

到目前为止，下面这些浏览器支持Strict模式：

- Chrome >= 13.0
- Safari >= 5.0
- Opera >= 12.0
- Firefox >= 4.0
- IE >= 10.0

话说回来，严格模式能完全兼容较老的浏览器。使用它的程序在严格或者普通模式下都能正常工作。

Strict模式的不同

严格模式的大部分改变都是与Javascript语法相关的：

* 重复属性和重复函数参数会报错

- 不合理使用`delete`会报错
- 访问`arguments.caller`或`arguments.callee`报错（与性能有关）
- 使用`with`的话会抛出语法错误
- `undefined`等之类的确定的变量不再可写
- 引入更多的保留关键字，比方说`implements`、`interface`、`let`、`package`、`private`、`protected`、`public`、`static`和`yield`

而且，严格模式还改变了一些运行时的行为：

- 全局变量必须显式申明（必须使用`var`）。全局`this`的值是`undefined`
- `eval`无法在局部作用域中引入新变量
- 函数必须在它们被使用前定义（先前的函数可以定义在任何地方）
- `arguments`不可变

CoffeeScript已经遵循了许多严格模式的要求，比方说在定义变量时一定会用`var`，但是在你的CoffeeScript程序中开启严格模式还是很有用的。实际上，在未来的版的CoffeeScript中在编译时会检查程序是否遵循严格模式。

使用Strict模式

为了开启严格的检查，你只需要在你的脚本或者函数中添加像下面这样的字符串即可：

```
->
  "use strict"
```

```
  # ... your code ...
```

就是这样，直接使用`'use strict'`字符串。没有比这更简单的了，而且它完全向后兼容。让我们实际试试严格模式。下面的代码在严格模式下会引起一个语法错误，而在普通模式下就没问题：

```
do ->
  "use strict"
  console.log(arguments.callee)
```

在Strict模式下，对`arguments.caller`和`arguments.callee`的访问已经被移除了，因为它们是性能损耗的要点。现在只要使用它们就会抛出语法错误。

在使用严格模式的时候，有个地方你需要额外小心。即使用`this`来创建全局变量。下面的例子在严格模式下会抛出一个`TypeError`的错误，但是在普通模式下能正常运行，产生一个全局变量：

```
do ->
  "use strict"
  class @Spine
```

这背后的原因是不一致性，即在严格模式下`this`是`undefined`的，而在普通模式下它指向`window`对象。解决办法就是像这样显式的在`window`对象上设置全局变量。

```
do ->
  "use strict"
  class window.Spine
```

虽然我推荐使用严格模式，但是并不值得这么做，一是严格模式并不会引入JavaScript中还没有完全准备好的特性，再者由于VM（虚拟机）在运行时会进行更多的检查从而让你的代码变慢。你应该在开发时使用严格模式，而发布时去掉它。

JavaScript Lint

JavaScript Lint是一个JavaScript代码质量检测工具，通过它运行你的代码是一种提高代码质量和最佳实践的非常好的方式。该项目基于另外一个类似的名为JSLint的工具。到JSLint的网站上查看一个非常棒的列表，该列表包含了所有它会检查的问题。包括全局变量、丢失的括号以及非严格相等比较。

好消息是CoffeeScript已经对编译输出进行过`lints`检查，因此CoffeeScript产生的JavaScript已经是与JavaScript Lint兼容的代码了。事实上，`coffee`命令工具支持`--lint`参数：

```
coffee --lint index.coffee
  index.coffee: 0 error(s), 0 warning(s)
```