一门编译到 JavaScript 的小巧语言

CoffeeScript 实用手册

极客学院出版

前言

CoffeeScript 是一门编译到 JavaScript 的小巧语言。在 Java 般笨拙的外表下，JavaScript 其实有着一颗华丽的心脏。CoffeeScript 尝试用简洁的方式展示 JavaScript 优秀的部分。

适用人群

本书着重讲解了 CoffeeScript 的基础语法，适合初学者。

学习前提

学习本书前，你需要了解 JavaScript 这门语言，应为 CoffeeScript 是建立在此基础上的。

目录

1

语法

服务端和客户端的代码重用

问题

当你在 CoffeeScript 上创建了一个函数，并希望将它用在有网页浏览器的客户端和有 Node.js 的服务端时。

解决方案

以下列方法输出函数：

```
 # simpleMath.coffee


 # these methods are private

add = (a, b) ->
   a + b

subtract = (a, b) ->
   a - b

square = (x) ->
   x * x

 # create a namespace to export our public methods

SimpleMath = exports? and exports or @SimpleMath = {}

 # items attached to our namespace are available in Node.js as well as client browsers

class SimpleMath.Calculator
   add: add
   subtract: subtract
   square: square
```

讨论

在上面的例子中，我们创建了一个新的名为 “SimpleMath” 的命名空间。如果 “export” 是有效的，我们的类就会作为一个 Node.js 模块输出。如果 “export” 是无效的，那么 “SimpleMath” 就会被加入全局命名空间，这样就可以被我们的网页使用了。

在 Node.js 中，我们可以使用 “require” 命令包含我们的模块。

```
$ node

> var SimpleMath = require('./simpleMath');
undefined
> var Calc = new SimpleMath.Calculator();
undefined
> console.log("5 + 6 = ", Calc.add(5, 6));
5 + 6 =  11
undefined
>
```

在网页中，我们可以通过将模块作为一个脚本嵌入其中。

```
<!DOCTYPE HTML>
<html lang="en-US">
<head>
   <meta charset="UTF-8">
   <title>SimpleMath Module Example</title>
   <script src="http://ajax.googleapis.com/ajax/libs/jquery/1.7.2/jquery.min.js"></script>
   <script src="simpleMath.js"></script>
   <script>
      jQuery(document).ready(function   (){
         var Calculator = new SimpleMath.Calculator();
         var result = $('<li>').html("5 + 6 = " + Calculator.add(5, 6));
         $('#SampleResults').append(result);
      });
   </script>
</head>
<body>
   <h1>A SimpleMath Example</h1>
   <ul id="SampleResults"></ul>
</body>
</html>
```

输出结果：

A SimpleMath Example

- 5 + 6 = 11

比较范围

问题

如果你想知道某个变量是否在给定的范围内。

解决方案

使用 CoffeeScript 的连缀比较语法。

```
maxDwarfism = 147
minAcromegaly = 213

height = 180

normalHeight = maxDwarfism < height < minAcromegaly
 # => true
```

讨论

这是从 Python 中借鉴过来的一个很棒的特性。利用这个特性，不必像下面这样写出完整的比较：

```
  normalHeight = height > maxDwarfism && height < minAcromegaly
```

CoffeeScript 支持像写数学中的比较表达式一样连缀两个比较，这样更直观。

嵌入 JavaScript

问题

你想在 CoffeeScript 中嵌入找到的或预先编写的 JavaScript 代码。

解决方案

把 JavaScript 包装到撇号中：

```
`function greet(name) {
return "Hello "+name;
}`

 # Back to CoffeeScript

greet "Coffee"
 # => "Hello Coffee"
```

讨论

这是在 CoffeeScript 代码中集成少量 JavaScript 而不必用 CoffeeScript 语法转换它们的最简单的方法。正如 CoffeeScript Language Reference 中展示的，可以在一定范围内混合这两种语言的代码：

```
hello = `function (name) {
return "Hello "+name
}`
hello "Coffee"
 # => "Hello Coffee"
```

这里的变量 "hello" 还在 CoffeeScript 中，但赋给它的函数则是用 JavaScript 写的。

For 循环

问题

你想通过一个 for 循环来迭代数组、对象或范围。

解决方案

```
 # for(i = 1; i<= 10; i++)

x for x in [1..10]
 # => [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ]


 # To count by 2

 # for(i=1; i<= 10; i=i+2)

x for x in [1..10] by 2
 # => [ 1, 3, 5, 7, 9 ]


 # Perform a simple operation like squaring each item.

x * x for x in [1..10]
 # = > [1,4,9,16,25,36,49,64,81,100]
```

讨论

CoffeeScript 使用推导（comprehension）来代替 for 循环，这些推导最终会被编译成 JavaScript 中等价的 for 循环。

类和对象

对象的链式调用

问题

你想调用一个对象上的多个方法，但不想每次都引用该对象。

解决方案

在每次链式调用后返回 this（即@）对象

```
class CoffeeCup
  constructor:  ->
    @properties=
      strength: 'medium'
      cream: false
      sugar: false
  strength: (newStrength) ->
    @properties.strength = newStrength
    this
  cream: (newCream) ->
    @properties.cream = newCream
    this
  sugar: (newSugar) ->
    @properties.sugar = newSugar
    this

morningCup = new CoffeeCup()

morningCup.properties # => { strength: 'medium', cream: false, sugar: false }

eveningCup = new CoffeeCup().strength('dark').cream(true).sugar(true)

eveningCup.properties # => { strength: 'dark', cream: true, sugar: true }
```

讨论

jQuery 库使用类似的手段从每一个相似的方法中返回选择符对象，并在后续方法中通过调整选择的范围修改该对象：

```
  $('p').filter('.topic').first()
```

对我们自己对象而言，一点点元编程就可以自动设置这个过程并明确声明返回 this 的意图。

```
addChainedAttributeAccessor = (obj, propertyAttr, attr) ->
  obj[attr] = (newValues...) ->
    if newValues.length == 0
      obj[propertyAttr][attr]
    else
      obj[propertyAttr][attr] = newValues[0]
      obj

class TeaCup
  constructor:  ->
    @properties=
      size: 'medium'
      type: 'black'
      sugar: false
      cream: false
    addChainedAttributeAccessor(this, 'properties', attr) for attr of @properties

earlgrey = new TeaCup().size('small').type('Earl Grey').sugar('false')

earlgrey.properties # => { size: 'small', type: 'Earl Grey', sugar: false }

earlgrey.sugar true

earlgrey.sugar() # => true
```

类方法和实例方法

问题

你想创建类和实例的方法。

解决方案

类方法

```
class Songs
  @_titles: 0    # Although it's directly accessible, the leading _ defines it by convention as private property.

  @get_count: ->
    @_titles

  constructor: (@artist, @title) ->
    @constructor._titles++     # Refers to <Classname>._titles, in this case Songs.titles

Songs.get_count()
 # => 0


song = new Songs("Rick Astley", "Never Gonna Give You Up")
Songs.get_count()
 # => 1


song.get_count()
 # => TypeError: Object <Songs> has no method 'get_count'
```

实例方法

```
class Songs
  _titles: 0    # Although it's directly accessible, the leading _ defines it by convention as private property.

  get_count: ->
    @_titles
```

```
  constructor: (@artist, @title) ->
    @_titles++

song = new Songs("Rick Astley", "Never Gonna Give You Up")
song.get_count()
 # => 1


Songs.get_count()
 # => TypeError: Object function Songs(artist, title) ... has no method 'get_count'
```

讨论

Coffeescript 会在对象本身中保存类方法（也叫静态方法），而不是在对象原型中（以及单一的对象实例），在保存了记录的同时也将类级的变量保存在中心位置。

类变量和实例变量

问题

你想创建类变量和实例变量（属性）。

解决方案

类变量

```
class Zoo
  @MAX_ANIMALS: 50
  MAX_ZOOKEEPERS: 3

  helpfulInfo: =>
    "Zoos may contain a maximum of #{@constructor.MAX_ANIMALS} animals and #{@MAX_ZOOKEEPERS} zoo keepe

Zoo.MAX_ANIMALS
 # => 50


Zoo.MAX_ZOOKEEPERS
 # => undefined (it is a prototype member)


Zoo::MAX_ZOOKEEPERS
 # => 3


zoo = new Zoo
zoo.MAX_ZOOKEEPERS
 # => 3

zoo.helpfulInfo()
 # => "Zoos may contain a maximum of 50 animals and 3 zoo keepers."


zoo.MAX_ZOOKEEPERS = "smelly"
zoo.MAX_ANIMALS = "seventeen"
```

```
zoo.helpfulInfo()
 # => "Zoos may contain a maximum of 50 animals and smelly zoo keepers."
```

实例变量

你必须在一个类的方法中才能定义实例变量（例如属性），在 constructor 结构中初始化你的默认值。

```
class Zoo
  constructor: ->
    @animals = [] # Here the instance variable is defined

  addAnimal: (name) ->
    @animals.push name


zoo = new Zoo()
zoo.addAnimal 'elephant'

otherZoo = new Zoo()
otherZoo.addAnimal 'lion'

zoo.animals
 # => ['elephant']


otherZoo.animals
 # => ['lion']
```

> **警告!**

不要试图在 constructor 外部添加变量（即使在 elsewhere 中提到了，由于潜在的 JavaScript 的原型概念，这不会像预期那样运行正确）。

```
class BadZoo
  animals: []         # Translates to BadZoo.prototype.animals = []; and is thus shared between instances

  addAnimal: (name) ->
    @animals.push name  # Works due to the prototype concept of Javascript


zoo = new BadZoo()
zoo.addAnimal 'elephant'
```

```
otherZoo = new BadZoo()
otherZoo.addAnimal 'lion'

zoo.animals
 # => ['elephant','lion'] # Oops...


otherZoo.animals
 # => ['elephant','lion'] # Oops...


BadZoo::animals
 # => ['elephant','lion'] # The value is stored in the prototype
```

讨论

Coffeescript 会将类变量的值保存在类中而不是它定义的原型中。这在定义类中的变量时是十分有用的，因为这不会被实体属性变量重写。

克隆对象（深度复制）

问题

你想复制一个对象，包含其所有子对象。

解决方案

```
clone = (obj) ->
  if not obj? or typeof obj isnt 'object'
    return obj

  if obj instanceof Date
    return new Date(obj.getTime())

  if obj instanceof RegExp
    flags = ''
    flags += 'g' if obj.global?
    flags += 'i' if obj.ignoreCase?
    flags += 'm' if obj.multiline?
    flags += 'y' if obj.sticky?
    return new RegExp(obj.source, flags)

  newInstance = new obj.constructor()

  for key of obj
    newInstance[key] = clone obj[key]

  return newInstance

x =
  foo: 'bar'
  bar: 'foo'

y = clone(x)

y.foo = 'test'

console.log x.foo isnt y.foo, x.foo, y.foo
 # => true, bar, test
```

讨论

通过赋值来复制对象与通过克隆函数来复制对象的区别在于如何处理引用。赋值只会复制对象的引用，而克隆函数则会:

- 创建一个全新的对象
- 这个新对象会复制原对象的所有属性，
- 并且对原对象的所有子对象，也会递归调用克隆函数，复制每个子对象的所有属性。

下面是一个通过赋值来复制对象的例子:

```
x =
  foo: 'bar'
  bar: 'foo'

y = x

y.foo = 'test'

console.log x.foo isnt y.foo, x.foo, y.foo
 # => false, test, test
```

显然，复制之后修改 y 也就修改了 x。

类的混合

问题

你有一些通用方法，你想把他们包含到很多不同的类中。

解决方案

使用 mixOf 库函数，它会生成一个混合父类。

```
mixOf = (base, mixins...) ->
  class Mixed extends base
  for mixin in mixins by -1 #earlier mixins override later ones
    for name, method of mixin::
      Mixed::[name] = method
  Mixed

...

class DeepThought
  answer: ->
    42

class PhilosopherMixin
  pontificate: ->
    console.log "hmm..."
    @wise = yes

class DeeperThought extends mixOf DeepThought, PhilosopherMixin
  answer: ->
    @pontificate()
    super()

earth = new DeeperThought
earth.answer()
 # hmm...

 # => 42
```

讨论

这适用于轻量级的混合。因此你可以从基类和基类的祖先中继承方法，也可以从混合类的基类和祖先中继承，但是不能从混合类的祖先中继承。与此同时，在声明了一个混合类后，此后的对这个混合类进行的改变是不会反应出来的。

创建一个不存在的对象字面值

问题

你想初始化一个对象字面值，但如果这个对象已经存在，你不想重写它。

解决方案

使用存在判断运算符（existential operator）。

```
window.MY_NAMESPACE ?= {}
```

讨论

这行代码与下面的 JavaScript 代码等价：

```
window.MY_NAMESPACE = window.MY_NAMESPACE || {};
```

这是 JavaScript 中一个常用的技巧，即使用对象字面值来定义命名空间。这样先判断是否存在同名的命名空间然后再创建，可以避免重写已经存在的命名空间。

CoffeeScrip 的 type 函数

问题

你想在不使用 typeof 的情况下知道一个函数的类型。（要了解为什么 typeof 不靠谱，请参见 http://javascript.crockford.com/remedial.html。）

解决方案

使用下面这个type函数

```
type = (obj) ->
  if obj == undefined or obj == null
    return String obj
  classToType = {
    '[object Boolean]': 'boolean',
    '[object Number]': 'number',
    '[object String]': 'string',
    '[object Function]': 'function',
    '[object Array]': 'array',
    '[object Date]': 'date',
    '[object RegExp]': 'regexp',
    '[object Object]': 'object'
  }
  return classToType[Object.prototype.toString.call(obj)]
```

讨论

这个函数模仿了 jQuery 的 $.type函数。

需要注意的是，在某些情况下，只要使用鸭子类型检测及存在运算符就可以不必检测对象的类型了。例如，下面这行代码不会发生异常，它会在 myArray 的确是数组（或者一个带有 push 方法的类数组对象）的情况下向其中推入一个元素，否则什么也不做。

```
  myArray?.push? myValue
```

3 字符串

大写单词首字母

问题

你想把字符串中每个单词的首字母转换为大写形式。

解决方案

使用“拆分-映射-拼接”模式：先把字符串拆分成单词，然后通过映射来大写单词第一个字母小写其他字母，最后再将转换后的单词拼接成字符串。

```
("foo bar baz".split(' ').map (word) -> word[0].toUpperCase() + word[1..-1].toLowerCase()).join ' '
 # => 'Foo Bar Baz'
```

或者使用列表推导（comprehension），也可以实现同样的结果：

```
(word[0].toUpperCase() + word[1..-1].toLowerCase() for word in "foo   bar   baz".split /\s+/).join ' '
 # => 'Foo Bar Baz'
```

讨论

“拆分-映射-拼接”是一种常用的脚本编写模式，可以追溯到Perl语言。如果能把这个功能直接通过“扩展类”放到String类里，就更方便了。

需要注意的是，“拆分-映射-拼接”模式存在两个问题。第一个问题，只有在文本形式统一的情况下才能有效拆分文本。如果来源字符串中有分隔符包含多个空白符，就需要考虑怎么过滤掉多余的空单词。一种解决方案是使用正则表达式来匹配空白符的串，而不是像前面那样只匹配一个空格：

```
("foo   bar   baz".split(/\s+/).map (word) -> word[0].toUpperCase() + word[1..-1].toLowerCase()).join ' '
 # => 'Foo Bar Baz'
```

但这样做又会导致第二个问题：在结果字符串中，原来的空白符串经过拼接就只剩下一个空格了。

不过，一般来说，这两个问题还是可以接受的。所以，“拆分-映射-拼接”仍然是一种有效的技术。

查找子字符串

问题

你想在一条消息中查找某个关键字第一次或最后一次出现的位置。

解决方案

分别使用 JavaScript 的 indexOf() 和 lastIndexOf() 方法查找字符串第一次和最后一次出现的位置。语法: string.indexOf searchstring, start

```
message = "This is a test string. This has a repeat or two. This might even have a third."
message.indexOf "This", 0
 # => 0


 # Modifying the start parameter

message.indexOf "This", 5
 # => 23


message.lastIndexOf "This"
 # => 49
```

讨论

还需要想办法统计出给定字符串在一条消息中出现的次数。

生成唯一ID

问题

你想随机生成一个唯一的标识符。

解决方案

可以根据一个随机数值生成一个 Base 36 编码的字符串。

```
uniqueId = (length=8) ->
  id = ""
  id += Math.random().toString(36).substr(2) while id.length < length
  id.substr 0, length

uniqueId()   # => n5yjla3b
uniqueId(2)  # => 0d
uniqueId(20) # => ox9eo7rt3ej0pb9kqlke
uniqueId(40) # => xu2vo4xjn4g0t3xr74zmndshrqlivn291d584alj
```

讨论

使用其他技术也可以，但这种方法相对来说性能更高，也更灵活。

字符串插值

问题

你想创建一个字符串，让它包含体现某个 CoffeeScript 变量的文本。

解决方案

使用 CoffeeScript 中类似 Ruby 的字符串插值，而不是 JavaScript 的字符串拼接。

插值：

```
muppet = "Beeker"
favorite = "My favorite muppet is #{muppet}!"

 # => "My favorite muppet is Beeker!"
```

```
square = (x) -> x * x
message = "The square of 7 is #{square 7}."

 # => "The square of 7 is 49."
```

讨论

CoffeeScript 的插值与 Ruby 类似，多数表达式都可以用在 #{ ... } 插值结构中。

CoffeeScript 支持在插值结构中放入多个有副作用的表达式，但建议大家不要这样做。因为只有表达式的最后一个值会被插入。

```
 # 可以这样做，但不要这样做。否则，你会疯掉。

square = (x) -> x * x
muppet = "Beeker"
message = "The square of 10 is #{muppet='Animal'; square 10}. Oh, and your favorite muppet is now #{muppet}."

 # => "The square of 10 is 100. Oh, and your favorite muppet is now Animal."
```

把字符串转换为小写形式

问题

你想把字符串转换成小写形式。

解决方案

使用 JavaScript 的 String 的 toLowerCase() 方法：

```
"ONE TWO THREE".toLowerCase()
 # => 'one two three'
```

讨论

toLowerCase() 是一个标准的 JavaScript 方法。不要忘了带圆括号。

语法块

通过下面的快捷方式可以添加某种类似 Ruby 的语法块：

```
String::downcase = -> @toLowerCase()
"ONE TWO THREE".downcase()
 # => 'one two three'
```

上面的代码演示了 CoffeeScript 的两个特性:

- 双冒号 :: 是引用 `.prototype` 的快捷方式；
- “at” 字符 @ 是引用 this 的快捷方式。

上面的代码会编译成如下 JavaScript 代码：

```
String.prototype.downcase = function() {
  return this.toLowerCase();
};
"ONE TWO THREE".downcase();
```

提示 尽管上面的用法在类似 Ruby 的语言中很常见，但在 JavaScript 中对本地对象的扩展经常被视为不好的。（请看：Maintainable JavaScript: Don’t modify objects you don’t own;Extending built-in native objects. Evil or not? ）

匹配字符串

问题

你想要匹配两个或多个字符串。

解决方案

计算把一个字符串转换成另一个字符串所需的编辑距离或操作数。

```
levenshtein = (str1, str2) ->

  l1 = str1.length
  l2 = str2.length
  prevDist = [0..l2]
  nextDist = [0..l2]

  for i in [1..l1] by 1
    nextDist[0] = i
    for j in [1..l2] by 1
      if (str1.charAt i-1) == (str2.charAt j-1)
        nextDist[j] = prevDist[j-1]
      else
        nextDist[j] = 1 + Math.min prevDist[j], nextDist[j-1], prevDist[j-1]
      [prevDist,nextDist]=[nextDist, prevDist]

  prevDist[l2]
```

讨论

可以使用赫斯伯格（ Hirschberg ）或瓦格纳菲舍尔（ Wagner – Fischer ）的算法来计算来文史特（ Levenshtein ）距离。这个例子用的是瓦格纳菲舍尔算法。

这个版本的文史特算法和内存呈线性关系，和时间呈二次方关系。

在这里我们使用 str.charAt i 这种表示法而不用 str[i] 这种方式，是因为后者在某些浏览器（ 如 IE7 ）中不支持。

起初，"by 1" 在两次循环中看起来似乎是没用的。它在这里是用来避免一个 coffeescript [i..j] 语法的常见错误。如果 str1 或 str2 为空字符串，那么 [1..l1] 或 [1..l2] 将会返回 [1,0] 。添加了 "by 1" 的循环也能编译出更加简洁高效的 javascript 。

最后，循环结尾处对回收数组的优化在这里主要是为了演示 coffeescript 中交换两个变量的语法。

重复字符串

问题

你想重复一个字符串。

解决方案

创建一个包含 n+1 个空元素的数组，然后用要重复的字符串作为连接字符将数组元素拼接到一起：

```
# 创建包含10个foo的字符串

Array(11).join 'foo'

# => "foofoofoofoofoofoofoofoofoofoo"
```

为字符串重复方法

你也可以在字符串的原型中为其创建方法。它十分简单：

```
# 为所有的字符串添加重复方法，这会重复返回 n 次字符串

String::repeat = (n) -> Array(n+1).join(this)
```

讨论

JavaScript 缺少字符串重复函数，CoffeeScript 也没有提供。虽然在这里也可以使用列表推导（ comprehensions ），但对于简单的字符串重复来说，还是像这样先构建一个包含 n+1 个空元素的数组，然后再把它拼接起来更方便。

拆分字符串

问题

你想拆分一个字符串。

解决方案

使用 JavaScript 字符串的 split() 方法：

```
"foo bar baz".split " "
 # => [ 'foo', 'bar', 'baz' ]
```

讨论

String 的这个 split() 方法是标准的 JavaScript 方法。可以用来基于任何分隔符——包括正则表达式来拆分字符串。这个方法还可以接受第二个参数，用于指定返回的子字符串数目。

```
"foo-bar-baz".split "-"
 # => [ 'foo', 'bar', 'baz' ]
```

```

"foo bar \t baz".split /\s+/ # => [ 'foo', 'bar', 'baz' ]

```

"the sun goes down and I sit on the old broken-down river pier".split " ", 2 # => ['the', 'sun']

清理字符串前后的空白符

问题

你想清理字符串前后的空白符。

解决方案

使用 JavaScript 的正则表达式来替换空白符。

要清理字符串前后的空白符，可以使用以下代码：

```
" padded string ".replace /^\s+|\s+$/g, "" # => 'padded string'
```

如果只想清理字符串前面的空白符，使用以下代码：

```
" padded string ".replace /^\s+/g, "" # => 'padded string '
```

如果只想清理字符串后面的空白符，使用以下代码：

```
" padded string ".replace /\s+$/g, "" # => ' padded string'
```

讨论

Opera、Firefox 和 Chrome 中 String 的原型都有原生的 trim 方法，其他浏览器也可以添加一个。对于这个方法而言，还是尽可能使

unless String::trim then String::trim = -> @replace /^\s+|\s+$/g, ""

" padded string ".trim() # => 'padded string'

语法块

还可以添加一些类似Ruby中的语法块，定义如下快捷方法：

String::strip = -> if String::trim? then @trim() else @replace /^\s+|\s+$/g, "" String::lstrip = -> @replac e /^\s+/g, "" String::rstrip = -> @replace /\s+$/g, ""

" padded string ".strip() # => 'padded string'

" padded string ".lstrip() # => 'padded string '

" padded string ".rstrip() # => ' padded string'

要想深入了解 JavaScript 执行 trim 操作时的性能，请参见 Steve Levithan 的这篇博客文章。

把字符串转换为大写形式

问题

你想把字符串转换成大写形式。

解决方案

使用 JavaScript 的 String 的 toUpperCase() 方法：

```
"one two three".toUpperCase() # => 'ONE TWO THREE'
```

讨论

toUpperCase() 是一个标准的 JavaScript 方法。不要忘了带圆括号。

语法块

通过下面的快捷方式可以添加某种类似 Ruby 的语法块：

String::upcase = -> @toUpperCase() "one two three".upcase() # => 'ONE TWO THREE'

上面的代码演示了 CoffeeScript 的两个特性:

* 双冒号 :: 是引用 .prototype 的快捷方式；
* “at” 字符 @ 是引用 this 的快捷方式。

上面的代码会编译成如下 JavaScript 代码：

String.prototype.upcase = function() { return this.toUpperCase(); }; "one two three".upcase(); ```

提示 尽管上面的用法在类似 Ruby 的语言中很常见，但在 JavaScript 中对本地对象的扩展经常被视为不好的。（请看：Maintainable JavaScript: Don’t modify objects you don’t own;Extending built-in native objects. Evil or not? ）

4

数组

检查变量的类型是否为数组

问题

你希望检查一个变量是否为一个数组。

```
myArray = []
console.log typeof myArray // outputs 'object'
```

“typeof” 运算符为数组输出了一个错误的结果。

解决方案

使用下面的代码：

```
typeIsArray = Array.isArray || ( value ) -> return {}.toString.call( value ) is '[object Array]'
```

为了使用这个，像下面这样调用 typeIsArray 就可以了。

```
myArray = []
typeIsArray myArray // outputs true
```

讨论

上面方法取自 "the Miller Device"。另外一个方式是使用 Douglas Crockford 的片段。

```
typeIsArray = ( value ) ->
  value and
    typeof value is 'object' and
    value instanceof Array and
    typeof value.length is 'number' and
    typeof value.splice is 'function' and
    not ( value.propertyIsEnumerable 'length' )
```

将数组连接

问题

你希望将两个数组连接到一起。

解决方案

在 JavaScript 中，有两个标准方法可以用来连接数组。

第一种是使用 JavaScript 的数组方法 concat()：

```
array1 = [1, 2, 3]
array2 = [4, 5, 6]
array3 = array1.concat array2
 # => [1, 2, 3, 4, 5, 6]
```

需要指出的是 array1 没有被运算修改。连接后形成的新数组的返回值是一个新的对象。

如果你希望在连接两个数组后不产生新的对象，那么你可以使用下面的技术：

```
array1 = [1, 2, 3]
array2 = [4, 5, 6]
Array::push.apply array1, array2
array1
 # => [1, 2, 3, 4, 5, 6]
```

在上面的例子中，Array.prototype.push.apply(a, b) 方法修改了 array1 而没有产生一个新的数组对象。

在 CoffeeScript 中，我们可以简化上面的方式，通过给数组创建一个新方法 merge()：

```
Array::merge = (other) -> Array::push.apply @, other

array1 = [1, 2, 3]
array2 = [4, 5, 6]
array1.merge array2
array1
 # => [1, 2, 3, 4, 5, 6]
```

另一种方法，我可以直接将一个 CoffeeScript splat(array2) 放入 push() 中，避免了使用数组原型。

```
array1 = [1, 2, 3]
array2 = [4, 5, 6]
array1.push array2...
array1
 # => [1, 2, 3, 4, 5, 6]
```

一个更加符合语言习惯的方法是在一个数组语言中直接使用 splat 运算符(...)。这可以用来连接任意数量的数组。

```
array1 = [1, 2, 3]
array2 = [4, 5, 6]
array3 = [array1..., array2...]
array3
 # => [1, 2, 3, 4, 5, 6]
```

讨论

CoffeeScript 缺少了一种用来连接数组的特殊语法，但是 concat() 和 push() 是标准的 JavaScript 方法。

由数组创建一个对象词典

问题

你有一组对象，例如：

```
cats = [
  {
    name: "Bubbles"
    age: 1
  },
  {
    name: "Sparkle"
    favoriteFood: "tuna"
  }
]
```

但是你想让它像词典一样，可以通过关键字访问它，就像使用 cats["Bubbles"]。

解决方案

你需要将你的数组转换为一个对象。通过这样使用 reduce：

```
 # key = The key by which to index the dictionary

Array::toDict = (key) ->
  @reduce ((dict, obj) -> dict[ obj[key] ] = obj if obj[key]?; return dict), {}
```

使用它时像下面这样：

```
catsDict = cats.toDict('name')
  catsDict["Bubbles"]
  # => { age: 1, name: "Bubbles" }
```

讨论

另一种方法是使用数组推导：

```
Array::toDict = (key) ->
  dict = {}
```

```
  dict[obj[key]] = obj for obj in this when obj[key]?
  dict
```

如果你使用 Underscore.js，你可以创建一个 mixin：

```
_.mixin toDict: (arr, key) ->
    throw new Error('_.toDict takes an Array') unless _.isArray arr
    _.reduce arr, ((dict, obj) -> dict[ obj[key] ] = obj if obj[key]?; return dict), {}
catsDict = _.toDict(cats, 'name')
catsDict["Sparkle"]
 # => { favoriteFood: "tuna", name: "Sparkle" }
```

由数组创建一个字符串

问题

你想由数组创建一个字符串。

解决方案

使用 JavaScript 的数组方法 toString()：

```
["one", "two", "three"].toString()
 # => 'one,two,three'
```

讨论

toString() 是一个标准的 JavaScript 方法。不要忘记圆括号。

定义数组范围

问题

你想定义一个数组的范围。

解决方案

在 CoffeeScript 中，有两种方式定义数组元素的范围。

```
myArray = [1..10]
 # => [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ]
```

```
myArray = [1...10]
 # => [ 1, 2, 3, 4, 5, 6, 7, 8, 9 ]
```

要想反转元素的范围，则可以写成下面这样。

```
myLargeArray = [10..1]
 # => [ 10, 9, 8, 7, 6, 5, 4, 3, 2, 1 ]
```

```
myLargeArray = [10...1]
 # => [ 10, 9, 8, 7, 6, 5, 4, 3, 2 ]
```

讨论

包含范围以 “..” 运算符定义，包含最后一个值。 排除范围以 “...” 运算符定义，并且通常忽略最后一个值。

筛选数组

问题

你想要根据布尔条件来筛选数组。

解决方案

使用 `Array.filter (ECMAScript 5): array = [1..10]`

```
array.filter (x) -> x > 5
 # => [6,7,8,9,10]
```

在 EC5 之前的实现中，可以通过添加一个筛选函数扩展 Array 的原型，该函数接受一个回调并对自身进行过滤，将回调函数返回 true 的元素收集起来。

```
 # 扩展 Array 的原型

Array::filter = (callback) ->
  element for element in this when callback(element)

array = [1..10]

 # 筛选偶数

filtered_array = array.filter (x) -> x % 2 == 0
 # => [2,4,6,8,10]


 # 过滤掉小于或等于5的元素

gt_five = (x) -> x > 5
filtered_array = array.filter gt_five
 # => [6,7,8,9,10]
```

讨论

这个方法与 Ruby 的 Array 的 #select 方法类似。

列表推导

问题

你有一个对象数组，想将它们映射到另一个数组，类似于 Python 的列表推导。

解决方案

使用列表推导，但不要忘记还有 mapping-arrays 。

```
electric_mayhem = [ { name: "Doctor Teeth", instrument: "piano" },
                    { name: "Janice", instrument: "lead guitar" },
                    { name: "Sgt. Floyd Pepper", instrument: "bass" },
                    { name: "Zoot", instrument: "sax" },
                    { name: "Lips", instrument: "trumpet" },
                    { name: "Animal", instrument: "drums" } ]

names = (muppet.name for muppet in electric_mayhem)
 # => [ 'Doctor Teeth', 'Janice', 'Sgt. Floyd Pepper', 'Zoot', 'Lips', 'Animal' ]
```

讨论

因为 CoffeeScript 直接支持列表推导，在你使用一个 Python 的语句时，他们会很好地起到作用。对于简单的映射，列表推导具有更好的可读性。但是对于复杂的转换或链式映射，映射数组可能更合适。

映射数组

问题

你有一个对象数组，想把这些对象映射到另一个数组中，就像 Ruby 的映射一样。

解决方案

使用 map() 和匿名函数，但不要忘了还有列表推导。

```
electric_mayhem = [ { name: "Doctor Teeth", instrument: "piano" },
                    { name: "Janice", instrument: "lead guitar" },
                    { name: "Sgt. Floyd Pepper", instrument: "bass" },
                    { name: "Zoot", instrument: "sax" },
                    { name: "Lips", instrument: "trumpet" },
                    { name: "Animal", instrument: "drums" } ]

names = electric_mayhem.map (muppet) -> muppet.name
 # => [ 'Doctor Teeth', 'Janice', 'Sgt. Floyd Pepper', 'Zoot', 'Lips', 'Animal' ]
```

讨论

因为 CoffeeScript 支持匿名函数，所以在 CoffeeScript 中映射数组就像在 Ruby 中一样简单。 映射在 CoffeeScript 中是处理复杂转换和连缀映射的好方法。如果你的转换如同上例中那么简单，那可能将它当成列表推导 看起来会清楚一些。

数组最大值

问题

你需要找出数组中包含的最大的值。

解决方案

你可以使用 JavaScript 实现，在列表推导基础上使用 Math.max()：

```
Math.max [12, 32, 11, 67, 1, 3]...
 # => 67
```

另一种方法，在 ECMAScript 5 中，可以使用 Array 的 reduce 方法，它与旧的 JavaScript 实现兼容。

```
 # ECMAScript 5

[12,32,11,67,1,3].reduce (a,b) -> Math.max a, b
 # => 67
```

讨论

Math.max 在这里比较两个数值，返回其中较大的一个。省略号 (...) 将每个数组价值转化为给函数的参数。你还可以使用它与其他带可变数量的参数进行讨论，如执行 console.log 。

归纳数组

问题

你有一个对象数组，想要把它们归纳为一个值，类似于 Ruby 中的 reduce() 和 reduceRight() 。

解决方案

可以使用一个匿名函数包含 Array 的 reduce() 和 reduceRight() 方法，保持代码清晰易懂。这里归纳可能会像对数值和字符串应用 + 运算符那么简单。

```
[1,2,3,4].reduce (x,y) -> x + y
 # => 10


["words", "of", "bunch", "A"].reduceRight (x, y) -> x + " " + y
 # => 'A bunch of words'
```

或者，也可能更复杂一些，例如把列表中的元素聚集到一个组合对象中。

```
people =
   { name: 'alec', age: 10 }
   { name: 'bert', age: 16 }
   { name: 'chad', age: 17 }

people.reduce (x, y) ->
   x[y.name]= y.age
   x
, {}
 # => { alec: 10, bert: 16, chad: 17 }
```

讨论

Javascript 1.8 中引入了 reduce 和 reduceRight ，而 Coffeescript 为匿名函数提供了简单自然的表达语法。二者配合使用，可以把集合的项合并为组合的结果。

删除数组中的相同元素

问题

你想从数组中删除相同元素。

解决方案

```
Array::unique = ->
  output = {}
  output[@[key]] = @[key] for key in [0...@length]
  value for key, value of output

[1,1,2,2,2,3,4,5,6,6,6,"a","a","b","d","b","c"].unique()
 # => [ 1, 2, 3, 4, 5, 6, 'a', 'b', 'd', 'c' ]
```

讨论

在 JavaScript 中有很多的独特方法来实现这一功能。这一次是基于“最快速的方法来查找数组的唯一元素”，出自这里 。

> 注意: 延长本机对象通常被认为是在 JavaScript 不好的做法，即便它在 Ruby 语言中相当普遍，（参考:Maintainable JavaScript: Don’t modify objects you don’t own

反转数组

问题

你想要反转数组元素。

解决方案

使用 JavaScript Array 的 reverse() 方法：

```
["one", "two", "three"].reverse()
 # => ["three", "two", "one"]
```

讨论

reverse() 是标准的 JavaScript 方法，别忘了带圆括号。

打乱数组中的元素

问题

你想打乱数组中的元素。

解决方案

Fisher-Yates shuffle 是一种高效、公正的方式来让数组中的元素随机化。这是一个相当简单的方法：在列表的结尾处开始，用一个随机元素交换最后一个元素列表中的最后一个元素。继续下一个并重复操作，直到你到达列表的起始端，最终列表中所有的元素都已打乱。这 Fisher-Yates shuffle Visualization 可以帮助你理解算法。

```
shuffle = (source) ->
  # Arrays with < 2 elements do not shuffle well. Instead make it a noop.
  return source unless source.length >= 2
  # From the end of the list to the beginning, pick element `index`.
  for index in [source.length-1..1]
    # Choose random element `randomIndex` to the front of `index` to swap with.
    randomIndex = Math.floor Math.random() * (index + 1)
    # Swap `randomIndex` with `index`, using destructured assignment
    [source[index], source[randomIndex]] = [source[randomIndex], source[index]]
  source

shuffle([1..9])
 # => [ 3, 1, 5, 6, 4, 8, 2, 9, 7 ]
```

讨论

一种错误的方式

有一个很常见但是错误的打乱数组的方式：通过随机数。

```
shuffle = (a) -> a.sort -> 0.5 - Math.random()
```

如果你做了一个随机的排序，你应该得到一个序列随机的顺序，对吧？甚至微软也用这种随机排序算法 。原来，这种随机排序算法产生有偏差的结果 ，因为它存在一种洗牌的错觉。随机排序不会导致一个工整的洗牌，它会导致序列排序质量的参差不齐。

速度和空间的优化

以上的解决方案处理速度是不一样的。该列表，当转换成 JavaScript 时，比它要复杂得多，变性分配比处理裸变量的速度要慢得多。以下代码并不完善，并且需要更多的源代码空间 … 但会编译量更小，运行更快：

```
shuffle = (a) ->
  i = a.length
  while --i > 0
    j = ~~(Math.random() * (i + 1)) # ~~ is a common optimization for Math.floor
    t = a[j]
    a[j] = a[i]
    a[i] = t
  a
```

扩展 Javascript 来包含乱序数组

下面的代码将乱序功能添加到数组原型中，这意味着你可以在任何希望的数组中运行它，并以更直接的方式来运行它。

```
Array::shuffle ?= ->
  if @length > 1 then for i in [@length-1..1]
    j = Math.floor Math.random() * (i + 1)
    [@[i], @[j]] = [@[j], @[i]]
  this

[1..9].shuffle()
 # => [ 3, 1, 5, 6, 4, 8, 2, 9, 7 ]
```

> 注意: 虽然它像在 Ruby 语言中相当普遍，但是在 JavaScript 中扩展本地对象通常被认为是不太好的做法 (参考: Maintainable JavaScript: Don’t modify objects you don’t own
> 正如提到的，以上的代码的添加是十分安全的。它仅仅需要添 Array :: shuffle 如果它不存在，就要添加赋值运算符 (? =) 。这样，我们就不会重写到别人的代码，或是本地浏览器的方式。
>
> 同时，如果你认为你会使用很多的实用功能，可以考虑使用一个工具库，像 Lo-dash 。他们有很多功能，像跨浏览器的简洁高效的地图。 Underscore 也是一个不错的选择。

检测每个元素

问题

你希望能够在特定的情况下检测出在数组中的每个元素。

解决方案

使用 Array.every (ECMAScript 5)：

```
evens = (x for x in [0..10] by 2)

evens.every (x)-> x % 2 == 0
 # => true
```

Array.every 被加入到 Mozilla 的 Javascript 1.6 ，ECMAScript 5 标准。如果你的浏览器支持，但仍无法实施 EC5 ，那么请检查 _.all from underscore.js 。

对于一个真实例子，假设你有一个多选择列表，如下：

```
<select multiple id="my-select-list">
  <option>1</option>
  <option>2</option>
  <option>Red Car</option>
  <option>Blue Car</option>
</select>
```

现在你要验证用户只选择了数字。让我们利用 array.every：

```
validateNumeric = (item)->
  parseFloat(item) == parseInt(item) && !isNaN(item)

values = $("#my-select-list").val()

values.every validateNumeric
```

讨论

这与 Ruby 中的 Array #all? 的方法很相似。

使用数组来交换变量

问题

你想通过数组来交换变量。

解决方案

使用 CoffeeScript 的解构赋值语法：

```
a = 1
b = 3

[a, b] = [b, a]

a
 # => 3


b
 # => 1
```

讨论

解构赋值可以不依赖临时变量实现变量值的交换。

这种语法特别适合在遍历数组的时候只想迭代最短数组的情况：

```
ray1 = [ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 ]
ray2 = [ 5, 9, 14, 20 ]

intersection = (a, b) ->
  [a, b] = [b, a] if a.length > b.length
  value for value in a when value in b

intersection ray1, ray2
 # => [ 5, 9 ]
```

```
intersection ray2, ray1
 # => [ 5, 9 ]
```

对象数组

问题

你想要得到一个与你的某些属性匹配的数组对象。

你有一系列的对象，如：

```
cats = [
  {
    name: "Bubbles"
    favoriteFood: "mice"
    age: 1
  },
  {
    name: "Sparkle"
    favoriteFood: "tuna"
  },
  {
    name: "flyingCat"
    favoriteFood: "mice"
    age: 1
  }
]
```

你想用某些特征来滤出想要的对象。例如：猫的位置 ({ 年龄: 1 }) 或者猫的位置 ({ 年龄: 1 , 最爱的食物: "老鼠" })

解决方案

你可以像这样来扩展数组：

```
Array::where = (query) ->
    return [] if typeof query isnt "object"
    hit = Object.keys(query).length
    @filter (item) ->
        match = 0
        for key, val of query
            match += 1 if item[key] is val
        if match is hit then true else false

cats.where age:1
```

```
 # => [ { name: 'Bubbles', favoriteFood: 'mice', age: 1 },{ name: 'flyingCat', favoriteFood: 'mice', age: 1 } ]


cats.where age:1, name: "Bubbles"
 # => [ { name: 'Bubbles', favoriteFood: 'mice', age: 1 } ]


cats.where age:1, favoriteFood:"tuna"
 # => []
```

讨论

这是一个确定的匹配。我们能够让匹配函数更加灵活：

```
Array::where = (query, matcher = (a,b) -> a is b) ->
    return [] if typeof query isnt "object"
    hit = Object.keys(query).length
    @filter (item) ->
        match = 0
        for key, val of query
            match += 1 if matcher(item[key], val)
        if match is hit then true else false

cats.where name:"bubbles"
 # => []

 # it's case sensitive


cats.where name:"bubbles", (a, b) -> "#{ a }".toLowerCase() is "#{ b }".toLowerCase()
 # => [ { name: 'Bubbles', favoriteFood: 'mice', age: 1 } ]

 # now it's case insensitive
```

处理收集的一种方式可以被叫做 “find” ，但是像 underscore 或者 lodash 这些库把它叫做 “where” 。

类似 Python 的 zip 函数

问题

你想把多个数组连在一起，生成一个数组的数组。换句话说，你需要实现与 Python 中的 zip 函数类似的功能。Python 的 zip 函数返回的是元组的数组，其中每个元组中包含着作为参数的数组中的第 i 个元素。

解决方案

使用下面的 CoffeeScript 代码：

```
 # Usage: zip(arr1, arr2, arr3, ...)

zip = () ->
  lengthArray = (arr.length for arr in arguments)
  length = Math.max.apply(Math, lengthArray)
  argumentLength = arguments.length
  results = []
  for i in [0...length]
    semiResult = []
    for arr in arguments
      semiResult.push arr[i]
    results.push semiResult
  return results

zip([0, 1, 2, 3], [0, -1, -2, -3])
 # => [[0, 0], [1, -1], [2, -2], [3, -3]]
```

5

日期和时间

计算复活节的日期

问题

你需要在给出的年份中找到复活节的月份和日期。

解决方案

下面的函数返回数组有两个要素：复活节的月份（ 1-12 ）和日期。如果没有给出任何参数，给出的结果是当前的一年。这是 在CoffeeScript 的匿名公历算法实现的。

```
gregorianEaster = (year = (new Date).getFullYear()) ->
  a = year % 19
  b = ~~(year / 100)
  c = year % 100
  d = ~~(b / 4)
  e = b % 4
  f = ~~((b + 8) / 25)
  g = ~~((b - f + 1) / 3)
  h = (19 * a + b - d - g + 15) % 30
  i = ~~(c / 4)
  k = c % 4
  l = (32 + 2 * e + 2 * i - h - k) % 7
  m = ~~((a + 11 * h + 22 * l) / 451)
  n = h + l - 7 * m + 114
  month = ~~(n / 31)
  day = (n % 31) + 1
  [month, day]
```

讨论

Javascript 中的月份是 0-11 。getMonth() 查找的是三月的话将返回数字 2 ，这个函数会返回 3 。如果你想要这个功能是一致的，你可以修改这个函数。

该函数使用 `~~` 符号代替来 Math.floor() 。

```
gregorianEaster()    # => [4, 24] (April 24th in 2011)
gregorianEaster 1972 # => [4, 2]
```

计算（美国和加拿大的）感恩节日期

问题

你需要在给出的年份中找到感恩节的月份和日期。

解决方案

下面的函数返回给出年份的感恩节的日期。如果没有给出任何参数，给出的结果是当前年份。

美国的感恩节是十一月的第四个星期四。

```
thanksgivingDayUSA = (year = (new Date).getFullYear()) ->
  first = new Date year, 10, 1
  day_of_week = first.getDay()
  22 + (11 - day_of_week) % 7
```

加拿大的感恩节是在十月的第二个周一。

```
thanksgivingDayCA = (year = (new Date).getFullYear()) ->
    first = new Date year, 9, 1
    day_of_week = first.getDay()
    8 + (8 - day_of_week) % 7
```

讨论

```
thanksgivingDayUSA() #=> 24 (November 24th, 2011)

thanksgivingDayCA() # => 10 (October 10th, 2011)

thanksgivingDayUSA(2012) # => 22 (November 22nd)

thanksgivingDayCA(2012) # => 8 (October 8th)
```

这个想法很简单：

1. 找出哪一天是以下各月份的第一天（美国十一月，加拿大十月）。
2. 计算从那天起偏移到下一个工作日的量（美国星期四，加拿大星期一）。

3. 将这个偏移量加上第一个可能的假期日期（第二十二个美国感恩节，第八个加拿大感恩节）。

计算两个日期中间的天数

问题

你需要找出两个日期间隔了几年，几个月，几天，几个小时，几分钟，几秒。

解决方案

利用 JavaScript 的日期计算函数 getTime() 。它提供了从 1970 年 1 月 1 日开始经过了多少毫秒。

```
DAY = 1000 * 60 * 60  * 24

d1 = new Date('02/01/2011')
d2 = new Date('02/06/2011')

days_passed = Math.round((d2.getTime() - d1.getTime()) / DAY)
```

讨论

使用毫秒，使计算时间跨度更容易，以避免日期的溢出错误。所以我们首先计算一天有多少毫秒。然后，给出了 2 个不同的日期，只须知道在 2 个日期之间的毫秒数，然后除以一天的毫秒数，这将得到 2 个不同的日期之间的天数。

如果你想计算出 2 个日期对象的小时数，你可以用毫秒的时间间隔除以一个小时有多少毫秒来得到。同样的可以得到几分钟和几秒。

```
HOUR = 1000 * 60 * 60

d1 = new Date('02/01/2011 02:20')
d2 = new Date('02/06/2011 05:20')

hour_passed = Math.round((d2.getTime() - d1.getTime()) / HOUR)
```

找到一个月中的最后一天

问题

你需要去找出一个月的最后一天，但是一年中的各月并没有一个固定时间表。

解决方案

利 用JavaScript 的日期下溢来找到给出月份的第一天：

```
now = new Date
lastDayOfTheMonth = new Date(1900+now.getYear(), now.getMonth()+1, 0)
```

讨论

JavaScript 的日期构造函数成功地处理溢出和下溢情况，使日期的计算变得很简单。鉴于这种简单操作，不需要担心一个给定的月份里有多少天；只需要用数学稍加推导。在十二月，以上的解决方案就是寻找当前年份的第十三个月的第 0 天日期，那么它就是下一年的一月一日，也计算出来今年十二月份 31 号的日期。

找到上一个月(或下一个月)

问题

你需要计算相关日期范围例如“上一个月”，“下一个月”。

解决方案

添加或减去当月的数字，JavaScript 的日期构造函数会修复数学知识。

```
 # these examples were written in GMT-6

 # Note that these examples WILL work in January!

now = new Date
 # => "Sun, 08 May 2011 05:50:52 GMT"


lastMonthStart = new Date 1900+now.getYear(), now.getMonth()-1, 1
 # => "Fri, 01 Apr 2011 06:00:00 GMT"


lastMonthEnd = new Date 1900+now.getYear(), now.getMonth(), 0
 # => "Sat, 30 Apr 2011 06:00:00 GMT"
```

讨论

JavaScript 的日期对象会处理下溢和溢出的月和日，并将相应调整日期对象。例如，你可以要求寻找三月的第 42 天，你将获得 4 月 11 日。

JavaScript 对象存储日期为从 1900 开始的每年的年份数，月份为一个 0 到 11 的整数，日期为从 1 到 31 的一个整数。在上述解决方案中，上个月的起始日是要求在本年度某一个月的第一天，但月是从 -1 至 10 。如果月是 -1 的日期对象将实际返回为前一年的十二月：

```
lastNewYearsEve = new Date 1900+now.getYear(), -1, 31
 # => "Fri, 31 Dec 2010 07:00:00 GMT"
```

对于溢出是同样的：

```
thirtyNinthOfFourteember = new Date 1900+now.getYear(), 13, 39
 # => "Sat, 10 Mar 2012 07:00:00 GMT"
```

计算月球的相位

问题

你想找出月球的相位。

解决方案

以下代码提供了一种计算给出日期的月球相位计算方案：

```
# moonPhase.coffee


# Moon-phase calculator

# Roger W. Sinnott, Sky & Telescope, June 16, 2006

# http://www.skyandtelescope.com/observing/objects/javascript/moon_phases

#

# Translated to CoffeeScript by Mike Hatfield @WebCoding4Fun


proper_ang = (big) ->
  tmp = 0
  if big > 0
    tmp = big / 360.0
    tmp = (tmp - (~~tmp)) * 360.0
  else
    tmp = Math.ceil(Math.abs(big / 360.0))
    tmp = big + tmp * 360.0

  tmp

jdn = (date) ->
  month = date.getMonth()
  day = date.getDate()
  year = date.getFullYear()
  zone = date.getTimezoneOffset() / 1440
```

```
    mm = month
    dd = day
    yy = year

    yyy = yy
    mmm = mm
    if mm < 3
        yyy = yyy - 1
        mmm = mm + 12

    day = dd + zone + 0.5
    a = ~~( yyy / 100 )
    b = 2 - a + ~~( a / 4 )
    jd = ~~( 365.25 * yyy ) + ~~( 30.6001 * ( mmm+ 1 ) ) + day + 1720994.5
    jd + b if jd > 2299160.4999999

moonElong = (jd) ->
    dr    = Math.PI / 180
    rd    = 1 / dr
    meeDT = Math.pow((jd - 2382148), 2) / (41048480 * 86400)
    meeT  = (jd + meeDT - 2451545.0) / 36525
    meeT2 = Math.pow(meeT, 2)
    meeT3 = Math.pow(meeT, 3)
    meeD  = 297.85 + (445267.1115 * meeT) - (0.0016300 * meeT2) + (meeT3 / 545868)
    meeD  = (proper_ang meeD) * dr
    meeM1 = 134.96 + (477198.8676 * meeT) + (0.0089970 * meeT2) + (meeT3 / 69699)
    meeM1 = (proper_ang meeM1) * dr
    meeM  = 357.53 + (35999.0503 * meeT)
    meeM  = (proper_ang meeM) * dr

    elong = meeD * rd + 6.29 * Math.sin( meeM1 )
    elong = elong     - 2.10 * Math.sin( meeM )
    elong = elong     + 1.27 * Math.sin( 2*meeD - meeM1 )
    elong = elong     + 0.66 * Math.sin( 2*meeD )
    elong = proper_ang elong
    elong = Math.round elong

    moonNum = ( ( elong + 6.43 ) / 360 ) * 28
    moonNum = ~~( moonNum )

    if moonNum is 28 then 0 else moonNum

getMoonPhase = (age) ->
    moonPhase = "new Moon"
    moonPhase = "first quarter" if age > 3 and age < 11
```

```
    moonPhase = "full Moon"     if age > 10 and age < 18
    moonPhase = "last quarter"  if age > 17 and age < 25

    if ((age is 1) or (age is 8) or (age is 15) or (age is 22))
        moonPhase = "1 day past " + moonPhase

    if ((age is 2) or (age is 9) or (age is 16) or (age is 23))
        moonPhase = "2 days past " + moonPhase

    if ((age is 3) or (age is 1) or (age is 17) or (age is 24))
        moonPhase = "3 days past " + moonPhase

    if ((age is 4) or (age is 11) or (age is 18) or (age is 25))
        moonPhase = "3 days before " + moonPhase

    if ((age is 5) or (age is 12) or (age is 19) or (age is 26))
        moonPhase = "2 days before " + moonPhase

    if ((age is 6) or (age is 13) or (age is 20) or (age is 27))
        moonPhase = "1 day before " + moonPhase

    moonPhase

MoonPhase = exports? and exports or @MoonPhase = {}

class MoonPhase.Calculator
    getMoonDays: (date) ->
        jd = jdn date
        moonElong jd

    getMoonPhase: (date) ->
        jd = jdn date
        getMoonPhase( moonElong jd )
```

讨论

此代码显示一个月球相位计算器对象的方法有两种。计算器 -> getmoonphase 将返回一用个文本表示的日期的月球相位。

这可以用在浏览器和 Node.js 中。

```
$ node
> var MoonPhase = require('./moonPhase.js');
 undefined
```

```
> var calc = new MoonPhase.Calculator();
 undefined
> calc.getMoonPhase(new Date());
 'full moon'
> calc.getMoonPhase(new Date(1972, 6, 30));
 '3 days before last quarter'
```

6 数学

数学常数

问题

你需要使用常见的数学常数，比如 π 或者 e 。

解决方案

使用 Javascript 的 Math object 来提供通常需要的数学常数。

```
Math.PI
 # => 3.141592653589793


 # Note: Capitalization matters! This produces no output, it's undefined.

Math.Pi
 # =>


Math.E
 # => 2.718281828459045


Math.SQRT2
 # => 1.4142135623730951


Math.SQRT1_2
 # => 0.7071067811865476


 # Natural log of 2. ln(2)

Math.LN2
 # => 0.6931471805599453


Math.LN10
 # => 2.302585092994046
```

```
Math.LOG2E
 # => 1.4426950408889634


Math.LOG10E
 # => 0.4342944819032518
```

讨论

另外一个例子是关于一个数学常数用于真实世界的问题，是数学章节有关弧度和角度的部分。

更快的 Fibonacci 算法

问题

你想计算出 Fibonacci 数列中的数值 N，但需迅速地算出结果。

解决方案

下面的方案（仍有需改进的地方）最初在 Robin Houston 的博客上被提出来。

这里给出一些关于该算法和改进方法的链接：

- ? http://bosker.wordpress.com/2011/04/29/the-worst-algorithm-in-the-world/
- ? http://www.math.rutgers.edu/~erowland/fibonacci
- ? http://jsfromhell.com/classes/bignumber
- ? http://www.math.rutgers.edu/~erowland/fibonacci
- ? http://bigintegers.blogspot.com/2010/11/square-division-power-square-root
- ? http://bugs.python.org/issue3451

以下的代码来源于：https://gist.github.com/1032685

```
###

Author: Jason Giedymin <jasong _a_t_ apache -dot- org>
      http://www.jasongiedymin.com
      https://github.com/JasonGiedymin

CoffeeScript Javascript 的快速 Fibonacci 代码是基于 Robin Houston 博客里的 python 代码。
见下面的链接。

我要介绍一下 Newtonian，Burnikel / Ziegle 和Binet 关于大数目框架算法的实现。

Todo:
- https://github.com/substack/node-bigint
- BZ and Newton mods.
- Timing

###
```

```
MAXIMUM_JS_FIB_N = 1476

fib_bits = (n) ->
  #代表一个作为二进制数字阵列的整数

  bits = []
  while n > 0
    [n, bit] = divmodBasic n, 2
    bits.push bit

  bits.reverse()
  return bits

fibFast = (n) ->
  #快速 Fibonacci

  if n < 0
    console.log "Choose an number >= 0"
    return

  [a, b, c] = [1, 0, 1]

  for bit in fib_bits n
    if bit
      [a, b] = [(a+c)*b, b*b + c*c]
    else
      [a, b] = [a*a + b*b, (a+c)*b]

    c = a + b
    return b

divmodNewton = (x, y) ->
  throw new Error "Method not yet implemented yet."

divmodBZ = () ->
  throw new Error "Method not yet implemented yet."

divmodBasic = (x, y) ->
  ###
  这里并没有什么特别的。如果可能的话，也许以后的版本将是Newtonian 或者 Burnikel / Ziegler 的。
  ###

  return [(q = Math.floor x/y), (r = if x < y then x else x % y)]
```

```
start = (new Date).getTime();
calc_value = fibFast(MAXIMUM_JS_FIB_N)
diff = (new Date).getTime() - start;
console.log "[#{calc_value}] took #{diff} ms."
```

平方根倒数快速算法

问题

你想快速计算某数的平方根倒数。

解决方案

在 Quake Ⅲ Arena 的源代码中，这个奇怪的算法对一个幻数进行整数运算，来计算平方根倒数的浮点近似值。

在 CoffeeScript 中，他使用经典原始的变量，以及由 Chris Lomont 发现的新的最优 32 位幻数。除此之外，还使用 64 位大小的幻数。

另一特征是可以通过控制牛顿迭代法的迭代次数来改变其精确度。

相比于传统的，该算法在性能上更胜一筹，这归功于使用的机器及其精确度。

运行的时候使用 coffee -c script.coffee 来编译 script：

然后复制粘贴编译的 JS 代码到浏览器的 JavaScript 控制台。

注意：你需要一个支持类型数组的浏览器

参考：

1. ftp://ftp.idsoftware.com/idstuff/source/quake3-1.32b-source.zip
2. http://www.lomont.org/Math/Papers/2003/InvSqrt.pdf
3. http://en.wikipedia.org/wiki/Newton%27s_method
4. https://developer.mozilla.org/en/JavaScripttypedarrays
5. http://en.wikipedia.org/wiki/Fastinversesquare_root

以下的代码来源于：https://gist.github.com/1036533

```
###



Author: Jason Giedymin <jasong _a_t_ apache -dot- org>
      http://www.jasongiedymin.com
```

```
    https://github.com/JasonGiedymin

在 Quake Ⅲ Arena 的源代码 [1](ftp://ftp.idsoftware.com/idstuff/source/quake3-1.32b-source.zip) 中，这个奇怪的算法对一个

在 CoffeeScript 中，我使用经典原始的变量，以及由 Chris Lomont [2](http://www.lomont.org/Math/Papers/2003/InvSqrt.pdf)

另一特征是可以通过控制牛顿迭代法 [3](http://en.wikipedia.org/wiki/Newton%27s_method) 的迭代次数来改变其精确度。

相比于传统的，该算法在性能上更胜一筹，归功于使用的机器及其精确度。

运行的时候使用 coffee -c script.coffee 来编译 script：


然后复制粘贴编译的 JS 代码到浏览器的 JavaScript 控制台。

注意：你需要一个支持类型数组 [4](https://developer.mozilla.org/en/JavaScript_typed_arrays) 的浏览器

 ###


approx_const_quake_32 = 0x5f3759df # See [1]
approx_const_32 = 0x5f375a86 # See [2]
approx_const_64 = 0x5fe6eb50c7aa19f9 # See [2]

fastInvSqrt_typed = (n, precision=1) ->
  # 使用类型数组。现在只能在浏览器中操作。
  # Node.JS 的版本即将推出。

  y = new Float32Array(1)
  i = new Int32Array(y.buffer)

  y[0] = n
  i[0] = 0x5f375a86 - (i[0] >> 1)

  for iter in [1...precision]
    y[0] = y[0] * (1.5 - ((n * 0.5) * y[0] * y[0]))

  return y[0]

 ### 单次运行示例


testSingle = () ->
  example_n = 10
```

```
    console.log("Fast InvSqrt of 10, precision 1: #{fastInvSqrt_typed(example_n)}")
    console.log("Fast InvSqrt of 10, precision 5: #{fastInvSqrt_typed(example_n, 5)}")
    console.log("Fast InvSqrt of 10, precision 10: #{fastInvSqrt_typed(example_n, 10)}")
    console.log("Fast InvSqrt of 10, precision 20: #{fastInvSqrt_typed(example_n, 20)}")
    console.log("Classic of 10: #{1.0 / Math.sqrt(example_n)}")

testSingle()
```

生成可预测的随机数

问题

你需要生成在一定范围内的随机数，但你也需要对发生器进行“生成种子”操作来提供可预测的值。

解决方案

编写你自己的随机数生成器。当然有很多方法可以做到这一点，这里给出一个简单的示例。 *该发生器绝对不可以以加密为目的！*

```
class Rand
  # 如果没有种子创建，使用当前时间作为种子
  constructor: (@seed) ->
    # Knuth and Lewis' improvements to Park and Miller's LCPRNG
    @multiplier = 1664525
    @modulo = 4294967296 # 2**32-1;
    @offset = 1013904223
    unless @seed? && 0 <= seed < @modulo
      @seed = (new Date().valueOf() * new Date().getMilliseconds()) % @modulo

  # 设置新的种子值
  seed: (seed) ->
    @seed = seed

  # 返回一个随机整数满足 0 <= n < @modulo
  randn: ->
    # new_seed = (a * seed + c) % m
    @seed = (@multiplier*@seed + @offset) % @modulo

 # 返回一个随机浮点满足 0 <= f < 1.0
  randf: ->
    this.randn() / @modulo

  # 返回一个随机的整数满足 0 <= f < n
  rand: (n) ->
    Math.floor(this.randf() * n)

  #返回一个随机的整数满足min <= f < max
```

```
rand2: (min, max) ->
  min + this.rand(max-min)
```

讨论

JavaScript 和 CoffeeScript 都不提供可产生随机数的发生器。编写发生器对于我们来说将是一个挑战，在于权衡量的随机性与发生器的简单性。对随机性的全面讨论已超出了本书的范围。如需进一步阅读，可参考 Donald Kunth 的 *The Art of Computer Programming* 第 II 卷第 3 章的 “ Random Numbers ”，以及 *Numeric al Recipes in C* 第二版本第 7 章的“ Random Numbers ”。

但是，对于这个随机数发生器只有简单的解释。这是一个线性同余伪随机数发生器，其运行源于一条数学公式 $I_{j+1} = (aI_j+c) \% m$，其中 a 是乘数，c 是加法偏移量，m 是模数。每次请求随机数时就会执行很大的乘法和加法运算——这里的“很大”与密钥空间有关——得到的结果将以模数的形式被返回密钥空间。

这个发生器的周期为 232。虽然它绝对不能以加密为目的，但是对于最简单的随机性要求来说，它是相当足够的。randn() 在循环之前将遍历整个密钥空间，下一个数由上一个来确定。

如果你想修补这个发生器，强烈建议你去阅读 Knuth 的 * The Art of Computer Programming * 中的第 3 章。随机数生成是件很容易弄糟的事情，然而 Knuth 会解释如何区分好的和坏的随机数生成。

不要把发生器的输出结果变成模数。如果你需要一个整数的范围，应使用分割的方法。线性同余发生器的低位是不具有随机性的。特别的是，它总是从偶数种子产生奇数，反之亦然。所以如果你需要一个随机的 0 或者 1，不要使用：

```
 # NOT random! Do not do this!

r.randn() % 2
```

因为你肯定得不到随机数字。反而，你应该使用 r.rand(2)。

生成随机数

问题

你需要生成在一定范围内的随机数。

解决方案

使用 JavaScript 的 Math.random() 来获得浮点数，满足 0<=X<1.0 。使用乘法和 Math.floor 得到在一定范围内的数字。

```
probability = Math.random()
0.0 <= probability < 1.0
 # => true


 # 注意百分位数不会达到 100。从 0 到 100 的范围实际上是 101 的跨度。

percentile = Math.floor(Math.random() * 100)
0 <= percentile < 100
 # => true


dice = Math.floor(Math.random() * 6) + 1
1 <= dice <= 6
 # => true


max = 42
min = -13
range = Math.random() * (max - min) + min
-13 <= range < 42
 # => true
```

讨论

对于 JavaScript 来说，它更直接更快。

需要注意到 JavaScript 的 Math.random() 不能通过发生器生成随机数种子来得到特定值。详情可参考产生可预测的随机数。

产生一个从 0 到 n（不包括在内）的数，乘以 n。
产生一个从 1 到 n（包含在内）的数，乘以 n 然后加上 1。

转换弧度和度

问题

你需要实现弧度和度之间的转换。

解决方案

使用 JavaScript 的 Math.PI 和一个简单的公式来转换两者。

```
 # 弧度转换成度

radiansToDegrees = (radians) ->
   degrees = radians * 180 / Math.PI

radiansToDegrees(1)
 # => 57.29577951308232


 # 度转换成弧度

degreesToRadians = (degrees) ->
   radians = degrees * Math.PI / 180

degreesToRadians(1)
 # => 0.017453292519943295
```

一个随机整数函数

问题

你想要获得两个整数（包含在内）之间的一个随机整数。

解决方案

使用以下的函数。

```
randomInt = (lower, upper) ->
  [lower, upper] = [0, lower]    unless upper?          # 用一个参数调用
  [lower, upper] = [upper, lower] if lower > upper      # Lower 必须小于 upper
  Math.floor(Math.random() * (upper - lower + 1) + lower) # 最后一条语句是一个返回值

(randomInt(1) for i in [0...10])
 # => [0,1,1,0,0,0,1,1,1,0]


(randomInt(1, 10) for i in [0...10])
 # => [7,3,9,1,8,5,4,10,10,8]
```

指数对数运算

问题

你需要进行包含指数和对数的运算。

解决方案

使用 JavaScript 的 Math 对象来提供常用的数学函数。

```
 # Math.pow(x, y) 返回 x^y

Math.pow(2, 4)
 # => 16


 # Math.exp(x) 返回 E^x，被简写为 Math.pow(Math.E, x)

Math.exp(2)
 # => 7.38905609893065


 # Math.log returns the natural (base E) log

Math.log(5)
 # => 1.6094379124341003

Math.log(Math.exp(42))
 # => 42


 # To get a log with some other base n, divide by Math.log(n)

Math.log(100) / Math.log(10)
 # => 2
```

讨论

若想了解关于数学对象的更多信息，请参阅 Mozilla 开发者网络上的文档。另可参阅数学常量关于数学对象中各种常量的讨论。

7

方法

去抖动函数

问题

你想只执行某个函数一次，在开始或结束时把多个连续的调用合并成一个简单的操作。

解决方案

使用一个命名函数：

```
debounce: (func, threshold, execAsap) ->
  timeout = null
  (args...) ->
    obj = this
    delayed = ->
      func.apply(obj, args) unless execAsap
      timeout = null
    if timeout
      clearTimeout(timeout)
    else if (execAsap)
      func.apply(obj, args)
    timeout = setTimeout delayed, threshold || 100
mouseMoveHandler: (e) ->
  @debounce((e) ->
    # 只能在鼠标光标停止 300 毫秒后操作一次。
  300)

someOtherHandler: (e) ->
  @debounce((e) ->
    # 只能在初次执行 250 毫秒后操作一次。
  250, true)
```

讨论

可参阅 John Hann 的博客文章，了解 JavaScript 去抖动方法。

当函数括号不可选

问题

你想要调用一个没有参数的函数，但不希望使用括号。

解决方案

不管怎样都使用括号。

另一个方法是使用 do 表示法，如下：

```
notify = -> alert "Hello, user!"
do notify if condition
```

编译成 JavaScript 则可表示为：

```
var notify;
notify = function() {
  return alert("Hello, user!");
};
if (condition) {
  notify();
}
```

讨论

这个方法与 Ruby 类似，在于都可以不使用括号来完成方法的调用。而不同点在于，CoffeeScript 把空的函数名作为函数的指针。这样以来，如果你不赋予一个方法任何参数，那么 CoffeeScript 将无法分辨你是想要调用函数还是把它作为引用。

这是好是坏呢？其实只是有所不同。它创造了一个意想不到的语法实例——括号并不*总是*可选的——但是它能让你流利地使用名字来传递和接收函数，这对于 Ruby 来说是难以实现的。

对于 CoffeeScript 来说，使用 do 表示法是一个巧妙的方法来克服括号使用恐惧症。尽管有部分人宁愿在函数调用中写出所有括号。

递归函数

问题

你想在一个函数中调用相同的函数。

解决方案

使用一个命名函数：

```
ping = ->
  console.log "Pinged"
  setTimeout ping, 1000
```

若为未命名函数，则使用 @arguments.callee@：

```
delay = 1000

setTimeout((->
  console.log "Pinged"
  setTimeout arguments.callee, delay
  ), delay)
```

讨论

虽然 arguments.callee 允许未命名函数的递归，在内存密集型应用中占有一定优势，但是命名函数相对来说目的更加明确，也更易于代码的维护。

提示参数

问题

你的函数将会被可变数量的参数所调用。

解决方案

使用 *splat* 。

```
loadTruck = (firstDibs, secondDibs, tooSlow...) ->
  truck:
    driversSeat: firstDibs
    passengerSeat: secondDibs
    trunkBed: tooSlow

loadTruck("Amanda", "Joel")
 # => { truck: { driversSeat: "Amanda", passengerSeat: "Joel", trunkBed: [] } }


loadTruck("Amanda", "Joel", "Bob", "Mary", "Phillip")
 # => { truck: { driversSeat: "Amanda", passengerSeat: "Joel", trunkBed: ["Bob", "Mary", "Phillip"] } }
```

使用尾部参数：

```
loadTruck = (firstDibs, secondDibs, tooSlow..., leftAtHome) ->
  truck:
    driversSeat: firstDibs
    passengerSeat: secondDibs
    trunkBed: tooSlow
  taxi:
    passengerSeat: leftAtHome

loadTruck("Amanda", "Joel", "Bob", "Mary", "Phillip", "Austin")
 # => { truck: { driversSeat: 'Amanda', passengerSeat: 'Joel', trunkBed: [ 'Bob', 'Mary', 'Phillip' ] }, taxi: { passengerSeat: 'Au


loadTruck("Amanda")
 # => { truck: { driversSeat: "Amanda", passengerSeat: undefined, trunkBed: [] }, taxi: undefined }
```

讨论

通过在函数其中的（不多于）一个参数之后添加一个省略号（...），CoffeeScript 能把所有不被其他命名参数采用的参数值整合进一个列表中。就算并没有提供命名参数，它也会制造一个空列表。

8

元编程

检测与构建丢失的函数

问题

你想要检测一个函数是否存在，如果不存在则构建该函数。（比如 Internet Explorer 8 的 ECMAScript 5 函数）。

解决方案

使用存在赋值运算符（?=）来把函数分配给类库的原型（使用 :: 简写），然后把它放于一个立即执行函数表达式中（do ->）使其含有所有变量。

```
do -> Array::filter ?= (callback) ->
  element for element in this when callback element

array = [1..10]

array.filter (x) -> x > 5
 # => [6,7,8,9,10]
```

讨论

在 JavaScript（同样地，在 CoffeeScript）中，对象都有一个原型成员，它定义了什么成员函数能够适用于基于该原型的所有对象。
在CoffeeScript中，你可以使用 :: 捷径来访问这个原型。所以如果你想要把过滤函数添加至数组类中，就执行 Array::filter = ... 语句。它能把过滤函数加至所有数组中。

但是，不要去覆盖一个在第一时间还没有构造的原型。比如，如果 Array::filter = ... 已经以快速本地形式存在于浏览器中，或者库制造者拥有其对于 Array::filter = ... 的独特版本，这样以来，你要么换一个慢速的 JavaScript 版本，要么打破这种依赖于其自身 Array::shuffle 的库。
你需要做的仅仅是在函数不存在的时候添加该函数。这就是存在赋值运算符（?=）的意义。如果我们执行 Array::filter = ... 语句，它会首先判断 Array::filter 是否已经存在。如果存在的话，它就会使用现在的版本。否则，它会添加你的版本。

最后，由于存在的赋值运算符在编译时会创建一些变量，我们会通过把它们封装在立即调用函数表达式（IIFE）中来简化代码。这将隐藏那些内部专用的变量，以防止泄露。所以假如我们写的函数已经存在，那么它将运

行，基本上什么都没做然后退出，绝对不会对你的代码造成影响。但是假如我们写的函数并不存在，那么我们发送出去的仅是一个作为闭包的函数。所以只有你写的函数能够对代码产生影响。无论哪种方式，?= 的内部运行都会被隐藏。

举例

接下来，我们用上述的方法编译了 CoffeeScript 并附加了说明：

```
// (function(){ ... })() 是一个 IIFE， 使用 `do ->` 来编译它。
(function() {

  // 它来自 `?=` 运算符，用来检查 Array.prototype.filter (`Array::filter`) 是否存在。
  // 如果确实存在，我们把它设置给其自身，并返回。如果不存在，则把它设置给函数，并返回函数。
  // The IIFE is only used to hide _base and _ref from the outside world.
  var _base, _ref;
  return (_ref = (_base = Array.prototype).filter) != null ? _ref : _base.filter = function(callback) {

    // `element for element in this when callback element`
    var element, _i, _len, _results;
    _results = [];
    for (_i = 0, _len = this.length; _i < _len; _i++) {
      element = this[_i];
      if (callback(element)) {
        _results.push(element);
      }
    }
    return _results;

  };
// The end of the IIFE from `do ->`
})();
```

扩展内置对象

问题

你想要扩展一个类来增加新的函数或者替换旧的。

解决方案

使用 :: 把你的新函数分配到对象或者类的原型中。

```
String::capitalize = () ->
  (this.split(/\s+/).map (word) -> word[0].toUpperCase() + word[1..-1].toLowerCase()).join ' '

"foo bar    baz".capitalize()
 # => 'Foo Bar Baz'
```

讨论

在 JavaScript（同样地，在 CoffeeScript ）中，对象都有一个原型成员，它定义了什么成员函数能够适用于基于该原型的所有对象。在 CoffeeScript 中，你可以使用 :: 捷径来直接访问这个原型。

> 注意：虽然这种做法在很多种语言中相当普遍，比如 Ruby，但是在 JavaScript 中，扩展本地对象通常被认为是不好的做法（可参考：可维护的 JavaScript：不要修改你不拥有的对象；扩展内置的本地对象。对还是错? 。）

9

jQuery

AJAX

问题

你想要使用 jQuery 来调用 AJAX。

解决方案

```
$ ?= require 'jquery' # 由于 Node.js 的兼容性

$(document).ready ->
  # 基本示例
  $.get '/', (data) ->
    $('body').append "Successfully got the page."

  $.post '/',
    userName: 'John Doe'
    favoriteFlavor: 'Mint'
    (data) -> $('body').append "Successfully posted to the page."

  # 高级设置
  $.ajax '/',
    type: 'GET'
    dataType: 'html'
    error: (jqXHR, textStatus, errorThrown) ->
      $('body').append "AJAX Error: #{textStatus}"
    success: (data, textStatus, jqXHR) ->
      $('body').append "Successful AJAX call: #{data}"
```

jQuery 1.5 和更新版本都增加了一种新补充的 API ， 用于处理不同的回调。

```
request = $.get '/'
  request.success (data) -> $('body').append "Successfully got the page again."
  request.error (jqXHR, textStatus, errorThrown) -> $('body').append "AJAX Error: ${textStatus}."
```

讨论

其中的 jQuery 和 $ 变量可以互换使用。另请参阅 Callback bindings 。

回调绑定

问题

你想要把一个回调与一个对象绑定在一起。

解决方案

```
$ ->
  class Basket
    constructor: () ->
      @products = []

      $('.product').click (event) =>
        @add $(event.currentTarget).attr 'id'

    add: (product) ->
      @products.push product
      console.log @products

  new Basket()
```

讨论

通过使用等号箭头（ => ）取代正常箭头（ -> ），函数将会自动与对象绑定，并可以访问 @- 可变量。

创建 jQuery 插件

问题

你想用 CoffeeScript 来创建 jQuery 插件。

解决方案

```
 # 参考 jQuery

$ = jQuery

 # 给 jQuery 添加插件对象

$.fn.extend
 # 把 pluginName 改成你的插件名字。
 pluginName: (options) ->
  # 默认设置
  settings =
   option1: true
   option2: false
   debug: false

  # 合并选项与默认设置。
  settings = $.extend settings, options

  # Simple logger.
  log = (msg) ->
   console?.log msg if settings.debug

  # _Insert magic here._
  return @each ()->
   log "Preparing magic show."
   # 你可以使用你的设置了。
   log "Option 1 value: #{settings.option1}"
```

讨论

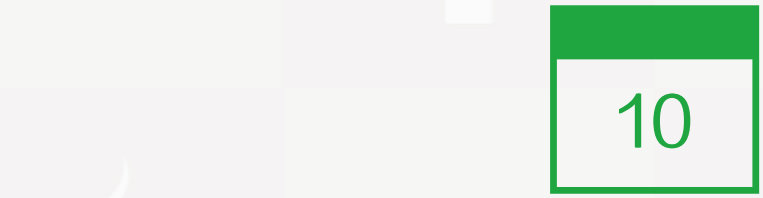
这里有几个关于如何使用新插件的例子。

JavaScript

```
$("body").pluginName({
  debug: true
});
```

CoffeeScript

```
$("body").pluginName
  debug: true
```



10

Ajax

不使用 jQuery 的 Ajax 请求

问题

你想要通过 AJAX 来从你的服务器加载数据，而不使用 jQuery 库。

解决方案

你将使用本地的 XMLHttpRequest 对象。

通过一个按钮来打开一个简单的测试 HTML 页面。

```
<!DOCTYPE HTML>
<html lang="en-US">
<head>
  <meta charset="UTF-8">
  <title>XMLHttpRequest Tester</title>
</head>
<body>
  <h1>XMLHttpRequest Tester</h1>
  <button id="loadDataButton">Load Data</button>

  <script type="text/javascript" src="XMLHttpRequest.js"></script>
</body>
</html>
```

当单击该按钮时，我们想给服务器发送 Ajax 请求以获取一些数据。对于该例子，我们使用一个 JSON 小文件。

```
// data.json
{
 message: "Hello World"
}
```

然后，创建 CoffeeScript 文件来保存页面逻辑。此文件中的代码创建了一个函数，当点击加载数据按钮时将会调用该函数。

```
1 # XMLHttpRequest.coffee
2 loadDataFromServer = ->
3   req = new XMLHttpRequest()
4
5   req.addEventListener 'readystatechange', ->
```

```
6    if req.readyState is 4                    # ReadyState Complete
7      successResultCodes = [200, 304]
8      if req.status in successResultCodes
9        data = eval '(' + req.responseText + ')'
10         console.log 'data message: ', data.message
11     else
12       console.log 'Error loading data...'
13
14   req.open 'GET', 'data.json', false
15   req.send()
16
17 loadDataButton = document.getElementById 'loadDataButton'
18 loadDataButton.addEventListener 'click', loadDataFromServer, false
```

讨论

在以上代码中，我们对 HTML 中按键进行了处理（第 16 行）以及添加了一个*单击*事件监听器（第 17 行）。在事件监听器中，我们把回调函数定义为 loadDataFromServer。

我们在第 2 行定义了 loadDataFromServer 回调的开头。

我们创建了一个 XMLHttpRequest 请求对象（第 3 行），并添加了一个 *readystatechange* 事件处理器。请求的 readyState 发生改变的那一刻，它就会被触发。

在事件处理器中，我们会检查判断是否满足 readyState=4，若等于则说明请求已经完成。然后检查请求的状态值。状态值为 200 或者 304 都代表着请求成功，其它则表示发生错误。

如果请求确实成功了，那我们就会对从服务器返回的 JSON 重新进行运算，然后把它分配给一个数据变量。此时，我们可以在需要的时候使用返回的数据。

在最后我们需要提出请求。

在第 13 行打开了一个 “GET” 请求来读取 data.json 文件。

在第 14 行把我们的请求发送至服务器。

旧版服务器支持

如果你的应用需要使用旧版本的 Internet Explorer ，你需确保 XMLHttpRequest 对象存在。为此，你可以在创建 XMLHttpRequest 实例之前输入以下代码。

```
if (typeof @XMLHttpRequest == "undefined")
  console.log 'XMLHttpRequest is undefined'
  @XMLHttpRequest = ->
    try
      return new ActiveXObject("Msxml2.XMLHTTP.6.0")
    catch error
    try
      return new ActiveXObject("Msxml2.XMLHTTP.3.0")
    catch error
    try
      return new ActiveXObject("Microsoft.XMLHTTP")
    catch error
    throw new Error("This browser does not support XMLHttpRequest.")
```

这段代码确保了 XMLHttpRequest 对象在全局命名空间中可用。

11

正则表达式

使用 Heregexes

问题

你需要写一个复杂的正则表达式。

解决方案

使用 CoffeeScript 的 “heregexes” ——可以忽视内部空白字符并可以包含注释的扩展正则表达式。

```
pattern = ///
  ^\(?(\d{3})\)? # 采集区域代码，忽略可选的括号
  [-\s]?(\d{3})  # 采集前缀，忽略可选破折号或空格
  -?(\d{4})      # 采集行号，忽略可选破折号
///
[area_code, prefix, line] = "(555)123-4567".match(pattern)[1..3]
 # => ['555', '123', '4567']
```

讨论

通过打破复杂的正则表达式和注释重点部分，它们变得更加容易去辨认和维护。例如，现在这是一个相当明显的做法去改变正则表达式以容许前缀和行号之间存在可选的空间。

heregexes 中的空白字符

空白字符在 heregexes 中是被忽视的——所以如果要为 ASCII 空格匹配字符，你应该怎么做呢?

我们的解决方案是使用 @\s@ 字符组，它能够匹配空格，制表符和换行符。假如你只想匹配一个空格，你需要使用 \X20 来表示字面上的 ASCII 空格。

使用 HTML 命名实体替换 HTML 标签

问题

你需要使用命名实体来替代 HTML 标签：

`<br/> => <br/>`

解决方案

```
htmlEncode = (str) ->
  str.replace /[&<>"']/g, ($0) ->
    "&" + {"&":"amp", "<":"lt", ">":"gt", '"':"quot", "'":"#39"}[$0] + ";"

htmlEncode('<a href="http://bn.com">Barnes & Noble</a>')
 # => '&lt;a href=&quot;http://bn.com&quot;&gt;Barnes &amp; Noble&lt;/a&gt;'
```

讨论

可能有更好的途径去执行上述方法。

替换子字符串

问题

你需要用另一个值替换字符串的一部分。

解决方案

使用 JavaScript 的 **replace** 方法。它与给定字符串匹配，并返回已编辑的字符串。

第一个版本需要 2 个参数：*模式*和*字符串替换*

```
"JavaScript is my favorite!".replace /Java/, "Coffee"
 # => 'CoffeeScript is my favorite!'


"foo bar baz".replace /ba./, "foo"
 # => 'foo foo baz'


"foo bar baz".replace /ba./g, "foo"
 # => 'foo foo foo'
```

第二个版本需要 2 个参数：*模式*和*回调函数*

```
"CoffeeScript is my favorite!".replace /(\w+)/g, (match) ->
  match.toUpperCase()
 # => 'COFFEESCRIPT IS MY FAVORITE!'
```

每次匹配需要调用回调函数，并且匹配值作为参数传给回调函数。

讨论

正则表达式是一种强有力的方式来匹配和替换字符串。

查找子字符串

问题

你需要搜索一个字符串，并返回匹配的起始位置或匹配值本身。

解决方案

有几种使用正则表达式的方法来实现这个功能。其中一些方法被称为 RegExp 模式或对象还有一些方法被称为 String 对象。

RegExp 对象

第一种方式是在 RegExp 模式或对象中调用 test 方法。test 方法返回一个布尔值：

```
match = /sample/.test("Sample text")
 # => false


match = /sample/i.test("Sample text")
 # => true
```

下一种方式是在 RegExp 模式或对象中调用 exec 方法。exec 方法返回一个匹配信息的数组或空值：

```
match = /s(amp)le/i.exec "Sample text"
 # => [ 'Sample', 'amp', index: 0, input: 'Sample text' ]


match = /s(amp)le/.exec "Sample text"
 # => null
```

String 对象

match 方法使给定的字符串与表达式对象匹配。有 “g” 标识的返回一个包含匹配项的数组，没有 “g” 标识的仅返回第一个匹配项或如果没有找到匹配项则返回 null 。

```
"Watch out for the rock!".match(/r?or?/g)
 # => [ 'o', 'or', 'ro' ]
```

```
"Watch out for the rock!".match(/r?or?/)
 # => [ 'o', index: 6, input: 'Watch out for the rock!' ]


"Watch out for the rock!".match(/ror/)
 # => null
```

search 方法以字符串匹配正则表达式，且如果找到的话返回匹配的起始位置，未找到的话则返回 −1 。

```
"Watch out for the rock!".search /for/
 # => 10


"Watch out for the rock!".search /rof/
 # => −1
```

讨论

正则表达式是一种可用来测试和匹配子字符串的强大的方法。

12

网络

客户端

问题

你想使用网络上提供的服务。

解决方案

创建一个基本的 TCP 客户机。

在 Node.js 中

```
net = require 'net'

domain = 'localhost'
port = 9001

connection = net.createConnection port, domain

connection.on 'connect', () ->
  console.log "Opened connection to #{domain}:#{port}."

connection.on 'data', (data) ->
  console.log "Received: #{data}"
  connection.end()
```

使用示例

可访问 Basic Server：

```
$ coffee basic-client.coffee
Opened connection to localhost:9001
Received: Hello, World!
```

讨论

最重要的工作发生在 *connection.on 'data'* 处理过程中，客户端接收到来自服务器的响应并最有可能安排对它的应答。

另请参阅 Basic Server，Bi-Directional Client 和 Bi-Directional Server 。

练习

？ 根据命令行参数或配置文件为选定的目标域和端口添加支持。

HTTP 客户端

问题

你想创建一个 HTTP 客户端。

解决方案

在这个方法中,我们将使用 node.js's HTTP 库。我们将从一个简单的客户端 GET 请求示例返回计算机的外部 IP 。

关于 GET

```
http = require 'http'

http.get { host: 'www.google.com' }, (res) ->
  console.log res.statusCode
```

get 函数，从 node.js's http 模块，发出一个 GET 请求到一个 http 服务器。响应是以回调的形式，我们可以在一个函数中处理。这个例子仅仅输出响应状态代码。检查一下：

```
$ coffee http-client.coffee
200
```

我的 IP 是什么?

如果你是在一个类似局域网的依赖于 NAT 的网络中,你可能会面临找出外部 IP 地址的问题。让我们为这个问题写一个小的 coffeescript 。

```
http = require 'http'

http.get { host: 'checkip.dyndns.org' }, (res) ->
  data = ''
  res.on 'data', (chunk) ->
    data += chunk.toString()
  res.on 'end', () ->
    console.log data.match(/([0-9]+\.){3}[0-9]+/)[0]
```

我们可以从监听 'data' 事件的结果对象中得到数据，知道它结束了一次 'end' 的触发事件。当这种情况发生时，我们可以做一个简单的正则表达式来匹配我们提取的 IP 地址。试一试：

```
$ coffee http-client.coffee
123.123.123.123
```

讨论

请注意 http.get 是 http.request 的快捷方式。后者允许您使用不同的方法发出 HTTP 请求，如 POST 或 PUT。

在这个问题上的 API 和整体信息，检查 node.js's http 和 https 文档页面。此外, HTTP spec 可能派上用场。

练习

？ 为键值存储 HTTP 服务器创建一个客户端，使用基本的 HTTP 服务器方法。

基本的 HTTP 服务器

问题

你想在网络上创建一个 HTTP 服务器。在这个方法中，我们将逐步从最小的服务器成为一个功能键值存储。

解决方案

我们将使用 node.js HTTP 库并在 Coffeescript 中创建最简单的 web 服务器。

开始 'hi\n'

我们可以通过导入 node.js HTTP 模块开始。这会包含 createServer ，一个简单的请求处理程序返回 HTTP 服务器。我们可以使用该服务器监听 TCP 端口。

```
http = require 'http'
server = http.createServer (req, res) -> res.end 'hi\n'
server.listen 8000
```

要运行这个例子，只需放在一个文件中并运行它。你可以用 ctrl-c 终止它。我们可以使用 curl 命令测试它,可用在大多数 *nix 平台：

```
$ curl -D - http://localhost:8000/
HTTP/1.1 200 OK
Connection: keep-alive
Transfer-Encoding: chunked

hi
```

发生什么了?

让我们一点点来反馈服务器上发生的事情。这时，我们可以友好的对待用户并提供他们一些 HTTP 头文件。

```
http = require 'http'

server = http.createServer (req, res) ->
  console.log req.method, req.url
  data = 'hi\n'
```

```
  res.writeHead 200,
    'Content-Type':   'text/plain'
    'Content-Length':  data.length
  res.end data

server.listen 8000
```

再次尝试访问它，但是这一次使用不同的 URL 路径，比如 `http://localhost:8000/coffee` 。你会看到这样的服务器控制台：

```
$ coffee http-server.coffee
GET /
GET /coffee
GET /user/1337
```

得到的东西

假如我们的网络服务器能够保存一些数据会怎么样？我们将在通过 GET 方法 请求检索的元素中设法想出一个简单的键值存储。并提供一个关键路径，服务器将请求返回相应的值,如果不存在则返回 404 错误。

```
http = require 'http'

store = # we'll use a simple object as our store
  foo:   'bar'
  coffee: 'script'

server = http.createServer (req, res) ->
  console.log req.method, req.url

  value = store[req.url[1..]]

  if not value
    res.writeHead 404
  else
    res.writeHead 200,
      'Content-Type': 'text/plain'
      'Content-Length': value.length + 1
    res.write value + '\n'

  res.end()

server.listen 8000
```

我们可以试试几种 url，看看它们如何回应：

```
$ curl -D - http://localhost:8000/coffee
HTTP/1.1 200 OK
Content-Type: text/plain
Content-Length: 7
Connection: keep-alive

script

$ curl -D - http://localhost:8000/oops
HTTP/1.1 404 Not Found
Connection: keep-alive
Transfer-Encoding: chunked
```

使用你的头文件

text/plain 是站不住脚的。如果我们使用 application/json 或 text/xml 会怎么样？同时,我们的存储检索过程也可以用一点重构——一些异常的抛出 & 处理怎么样？来看看我们能想出什么：

```
http = require 'http'

 # known mime types

[any, json, xml] = ['*/*', 'application/json', 'text/xml']

 # gets a value from the db in format [value, contentType]

get = (store, key, format) ->
  value = store[key]
  throw 'Unknown key' if not value
  switch format
    when any, json then [JSON.stringify({ key: key, value: value }), json]
    when xml then ["<key>#{ key }</key>\n<value>#{ value }</value>", xml]
    else throw 'Unknown format'

store =
  foo:    'bar'
  coffee: 'script'

server = http.createServer (req, res) ->
  console.log req.method, req.url

  try
    key = req.url[1..]
    [value, contentType] = get store, key, req.headers.accept
```

```
      code = 200
    catch error
      contentType = 'text/plain'
      value = error
      code = 404

    res.writeHead code,
      'Content-Type': contentType
      'Content-Length': value.length + 1
    res.write value + '\n'
    res.end()

server.listen 8000
```

这个服务器仍然会返回一个匹配给定键的值,如果不存在则返回 404 错误。但它根据标头 Accept 将响应在 JSON 或 XML 结构中。可亲眼看一下:

```
$ curl http://localhost:8000/
Unknown key

$ curl http://localhost:8000/coffee
{"key":"coffee","value":"script"}

$ curl -H "Accept: text/xml" http://localhost:8000/coffee
<key>coffee</key>
<value>script</value>

$ curl -H "Accept: image/png" http://localhost:8000/coffee
Unknown format
```

你需要有所返回

我们的最后一步是提供客户端存储数据的能力。我们将通过监听 POST 请求来保持 RESTiness。

```
http = require 'http'

 # known mime types

[any, json, xml] = ['*/*', 'application/json', 'text/xml']

 # gets a value from the db in format [value, contentType]

get = (store, key, format) ->
  value = store[key]
```

```
  throw 'Unknown key' if not value
  switch format
    when any, json then [JSON.stringify({ key: key, value: value }), json]
    when xml then ["<key>#{ key }</key>\n<value>#{ value }</value>", xml]
    else throw 'Unknown format'

# puts a value in the db

put = (store, key, value) ->
  throw 'Invalid key' if not key or key is ''
  store[key] = value

store =
  foo:    'bar'
  coffee: 'script'

# helper function that responds to the client

respond = (res, code, contentType, data) ->
  res.writeHead code,
    'Content-Type': contentType
    'Content-Length': data.length
  res.write data
  res.end()

server = http.createServer (req, res) ->
  console.log req.method, req.url
  key = req.url[1..]
  contentType = 'text/plain'
  code = 404

  switch req.method
    when 'GET'
      try
        [value, contentType] = get store, key, req.headers.accept
        code = 200
      catch error
        value = error
      respond res, code, contentType, value + '\n'

    when 'POST'
      value = ''
      req.on 'data', (chunk) -> value += chunk
      req.on 'end', () ->
        try
```

```
            put store, key, value
            value = ''
            code = 200
        catch error
            value = error + '\n'
        respond res, code, contentType, value

server.listen 8000
```

在一个 POST 请求中注意数据是如何接收的。通过在“数据”和“结束”请求对象的事件中附上一些处理程序，我们最终能够从客户端缓冲和保存数据。

```
$ curl -D - http://localhost:8000/cookie
HTTP/1.1 404 Not Found # ...
Unknown key

$ curl -D - -d "monster" http://localhost:8000/cookie
HTTP/1.1 200 OK # ...

$ curl -D - http://localhost:8000/cookie
HTTP/1.1 200 OK # ...
{"key":"cookie","value":"monster"}
```

讨论

给 http.createServer 一个函数 (request，response) - >…… 它将返回一个服务器对象，我们可以用它来监听一个端口。让服务器与 request 和 response 对象交互。使用 server.listen 8000 监听端口 8000。

在这个问题上的 API 和整体信息，参考 node.js http 和 https 文档页面。此外，HTTP spec 可能派上用场。

练习

在服务器和开发人员之间创建一个层，允许开发人员做类似的事情：

```
server = layer.createServer
  'GET /': (req, res) ->
    ...
  'GET /page': (req, res) ->
    ...
  'PUT /image': (req, res) ->
    ...
```

服务器

问题

你想在网络上提供一个服务器。

解决方案

创建一个基本的 TCP 服务器。

在 Node.js 中

```
net = require 'net'

domain = 'localhost'
port = 9001

server = net.createServer (socket) ->
  console.log "Received connection from #{socket.remoteAddress}"
  socket.write "Hello, World!\n"
  socket.end()

console.log "Listening to #{domain}:#{port}"
server.listen port, domain
```

使用示例

可访问 Basic Client：

```
$ coffee basic-server.coffee
Listening to localhost:9001
Received connection from 127.0.0.1
Received connection from 127.0.0.1
[...]
```

讨论

函数将为每个客户端新连接的新插口传递给 @net.createServer@ 。基本的服务器与访客只进行简单地交互，但是复杂的服务器会将插口连上一个专用的处理程序,然后返回等待下一个用户的任务。

另请参阅 Basic Client，Bi-Directional Server 和 Bi-Directional Client。

练习

？ 为选定的目标域和基于命令行参数或配置文件的端口添加支持。

双向客户端

问题

你想通过网络提供持续的服务,与客户保持持续的联系。

解决方案

创建一个双向 TCP 客户机。

在 Node.js 中

```
net = require 'net'

domain = 'localhost'
port = 9001

ping = (socket, delay) ->
  console.log "Pinging server"
  socket.write "Ping"
  nextPing = -> ping(socket, delay)
  setTimeout nextPing, delay

connection = net.createConnection port, domain

connection.on 'connect', () ->
  console.log "Opened connection to #{domain}:#{port}"
  ping connection, 2000

connection.on 'data', (data) ->
  console.log "Received: #{data}"

connection.on 'end', (data) ->
  console.log "Connection closed"
  process.exit()
```

使用示例

可访问 Bi-Directional Server：

```
$ coffee bi-directional-client.coffee
Opened connection to localhost:9001
Pinging server
Received: You have 0 peers on this server
Pinging server
Received: You have 0 peers on this server
Pinging server
Received: You have 1 peer on this server
[...]
Connection closed
```

讨论

这个特殊示例发起与服务器联系并在 @connection.on 'connect'@ 处理程序中开启对话。大量的工作在一个真正的用户中，然而 @connection.on 'data'@ 处理来自服务器的输出。@ping@ 函数递归是为了说明连续与服务器通信可能被真实的用户移除。

另请参阅 Bi-Directional Server，Basic Client 和 Basic Server。

练习

？ 为选定的目标域和基于命令行参数或配置文件的端口添加支持。

双向服务器

问题

你想通过网络提供持续的服务，与客户保持持续的联系。

解决方案

创建一个双向 TCP 服务器。

在 Node.js 中

```
net = require 'net'

domain = 'localhost'
port = 9001

server = net.createServer (socket) ->
  console.log "New connection from #{socket.remoteAddress}"

  socket.on 'data', (data) ->
    console.log "#{socket.remoteAddress} sent: #{data}"
    others = server.connections - 1
    socket.write "You have #{others} #{others == 1 and "peer" or "peers"} on this server"

console.log "Listening to #{domain}:#{port}"
server.listen port, domain
```

使用示例

可访问 Bi-Directional Client：

```
$ coffee bi-directional-server.coffee
Listening to localhost:9001
New connection from 127.0.0.1
127.0.0.1 sent: Ping
127.0.0.1 sent: Ping
```

```
127.0.0.1 sent: Ping
[...]
```

讨论

大部分工作在 @socket.on 'data'@ 中，处理所有的输入端。真正的服务器可能会将数据传给另一个函数处理并生成任何响应以便源程序处理。

练习

？ 为选定的目标域和基于命令行参数或配置文件的端口添加支持。

13 设计模式

适配器模式

问题

想象你去国外旅行，一旦你意识到你的电源线插座与酒店房间墙上的插座不兼容时，幸运的是你记得带你的电源适配器。它将一边连接你的电源线插座另一边连接墙壁插座，允许它们之间进行通信。

同样的情况也可能会出现在代码中，当两个（或更多）实例（类、模块等）想跟对方通信，但其通信协议（例如，他们所使用的语言交流）不同。在这种情况下，Adapter 模式更方便。它会充当翻译，从一边到另一边。

解决方案

```
 # a fragment of 3-rd party grid component

class AwesomeGrid
   constructor: (@datasource)->
      @sort_order = 'ASC'
      @sorter = new NullSorter # in this place we use NullObject pattern (another useful pattern)
   setCustomSorter: (@customSorter) ->
      @sorter = customSorter
   sort: () ->
      @datasource = @sorter.sort @datasource, @sort_order
      # don't forget to change sort order


class NullSorter
   sort: (data, order) -> # do nothing; it is just a stub

class RandomSorter
   sort: (data)->
      for i in [data.length-1..1] #let's shuffle the data a bit
            j = Math.floor Math.random() * (i + 1)
            [data[i], data[j]] = [data[j], data[i]]
      return data

class RandomSorterAdapter
   constructor: (@sorter) ->
   sort: (data, order) ->
      @sorter.sort data
```

```
agrid = new AwesomeGrid ['a','b','c','d','e','f']
agrid.setCustomSorter new RandomSorterAdapter(new RandomSorter)
agrid.sort() # sort data with custom sorter through adapter
```

讨论

当你要组织两个具有不同接口的对象之间的交互时，适配器是有用的。它可以当你使用第三方库或者使用遗留代码时使用。在任何情况下小心使用适配器：它可以是有用的，但它也可以导致设计错误。

桥接模式

问题

你需要为代码保持一个可靠的接口，可以经常变化或者在多种实现间转换。

解决方案

使用桥接模式作为不同的实现和剩余代码的中间体。

假设你开发了一个浏览器的文本编辑器保存到云。然而，现在你需要通过独立客户端的端口将其在本地保存。

```
class TextSaver
  constructor: (@filename, @options) ->
  save: (data) ->

class CloudSaver extends TextSaver
  constructor: (@filename, @options) ->
    super @filename, @options
  save: (data) ->
    # Assuming jQuery
    # Note the fat arrows
    $( =>
      $.post "#{@options.url}/#{@filename}", data, =>
        alert "Saved '#{data}' to #{@filename} at #{@options.url}."
    )

class FileSaver extends TextSaver
  constructor: (@filename, @options) ->
    super @filename, @options
    @fs = require 'fs'
  save: (data) ->
    @fs.writeFile @filename, data, (err) => # Note the fat arrow
      if err? then console.log err
      else console.log "Saved '#{data}' to #{@filename} in #{@options.directory}."

filename = "temp.txt"
data = "Example data"

saver = if window?
  new CloudSaver filename, url: 'http://localhost' # => Saved "Example data" to temp.txt at http://localhost
```

```
else if root?
    new FileSaver filename, directory: './' # => Saved "Example data" to temp.txt in ./

saver.save data
```

讨论

桥接模式可以帮助你将特定实现的代码置于看不见的地方，这样你就可以专注于你的程序中的具体代码。在上面的示例中,应用程序的其余部分可以称为 saver.save data ，不考虑文件的最终结束。

生成器模式

问题

你需要准备一个复杂的、多部分的对象，你希望操作不止一次或有不同的配置。

解决方案

创建一个生成器封装对象的产生过程。

Todo.txt 格式提供了一个先进的但还是纯文本的方法来维护待办事项列表。手工输入每个项目有损耗且容易出错，然而 TodoTxtBuilder 类可以解决我们的麻烦：

```
class TodoTxtBuilder
    constructor: (defaultParameters={ }) ->
        @date = new Date(defaultParameters.date) or new Date
        @contexts = defaultParameters.contexts or [ ]
        @projects = defaultParameters.projects or [ ]
        @priority =  defaultParameters.priority or undefined
    newTodo: (description, parameters={ }) ->
        date = (parameters.date and new Date(parameters.date)) or @date
        contexts = @contexts.concat(parameters.contexts or [ ])
        projects = @projects.concat(parameters.projects or [ ])
        priorityLevel = parameters.priority or @priority
        createdAt = [date.getFullYear(), date.getMonth()+1, date.getDate()].join("-")
        contextNames = ("@#{context}" for context in contexts when context).join(" ")
        projectNames = ("+#{project}" for project in projects when project).join(" ")
        priority = if priorityLevel then "(#{priorityLevel})" else ""
        todoParts = [priority, createdAt, description, contextNames, projectNames]
        (part for part in todoParts when part.length > 0).join " "

builder = new TodoTxtBuilder(date: "10/13/2011")

builder.newTodo "Wash laundry"

 # => '2011-10-13 Wash laundry'


workBuilder = new TodoTxtBuilder(date: "10/13/2011", contexts: ["work"])
```

```
workBuilder.newTodo "Show the new design pattern to Lucy", contexts: ["desk", "xpSession"]
 # => '2011-10-13 Show the new design pattern to Lucy @work @desk @xpSession'

workBuilder.newTodo "Remind Sean about the failing unit tests", contexts: ["meeting"], projects: ["compilerRefactor"], prio
 # => '(A) 2011-10-13 Remind Sean about the failing unit tests @work @meeting +compilerRefactor'
```

讨论

TodoTxtBuilder 类负责所有文本的生成，让程序员关注每个工作项的独特元素。此外，命令行工具或 GUI 可以插入这个代码且之后仍然保持支持，提供轻松、更高版本的格式。

前期建设

并不是每次创建一个新的实例所需的对象都要从头开始，我们将负担转移到一个单独的对象，可以在对象创建过程中进行调整。

```
builder = new TodoTxtBuilder(date: "10/13/2011")
builder.newTodo "Order new netbook"
 # => '2011-10-13 Order new netbook'

builder.projects.push "summerVacation"
builder.newTodo "Buy suntan lotion"
 # => '2011-10-13 Buy suntan lotion +summerVacation'

builder.contexts.push "phone"
builder.newTodo "Order tickets"
 # => '2011-10-13 Order tickets @phone +summerVacation'

delete builder.contexts[0]
```

```
builder.newTodo "Fill gas tank"

 # => '2011-10-13 Fill gas tank +summerVacation'
```

练习

? 扩大 project- 和 context-tag 生成代码来过滤掉重复的条目。

? 一些 Todo.txt 用户喜欢在任务描述中插入项目和上下文的标签。添加代码来识别这些标签和过滤器的结束标记。

命令模式

问题

你需要让另一个对象处理你自己的可执行的代码。

解决方案

使用 Command pattern 传递函数的引用。

```
 # Using a private variable to simulate external scripts or modules

incrementers = (() ->
   privateVar = 0

   singleIncrementer = () ->
      privateVar += 1

   doubleIncrementer = () ->
      privateVar += 2

   commands =
      single: singleIncrementer
      double: doubleIncrementer
      value: -> privateVar
)()

class RunsAll
   constructor: (@commands...) ->
   run: -> command() for command in @commands

runner = new RunsAll(incrementers.single, incrementers.double, incrementers.single, incrementers.double)
runner.run()
incrementers.value() # => 6
```

讨论

以函数作为一级的对象且从 Javascript 函数的变量范围中继承，CoffeeScript 使语言模式几乎看不出来。事实上，任何函数传递回调函数可以作为一个*命令*。

jqXHR 对象返回 jQuery AJAX 方法使用此模式。

```
jqxhr = $.ajax
    url: "/"

logMessages = ""

jqxhr.success -> logMessages += "Success!\n"
jqxhr.error -> logMessages += "Error!\n"
jqxhr.complete -> logMessages += "Completed!\n"

 # On a valid AJAX request:

 # logMessages == "Success!\nCompleted!\n"
```

修饰模式

问题

你有一组数据，需要在多个过程、可能变换的方式下处理。

解决方案

使用修饰模式来构造如何更改应用。

```
miniMarkdown = (line) ->
  if match = line.match /^(#+)\s*(.*)$/
    headerLevel = match[1].length
    headerText = match[2]
    "<h#{headerLevel}>#{headerText}</h#{headerLevel}>"
  else
    if line.length > 0
      "<p>#{line}</p>"
    else
      ''

stripComments = (line) ->
  line.replace /\s*\/\/.*$/, '' # Removes one-line, double-slash C-style comments

class TextProcessor
  constructor: (@processors) ->

  reducer: (existing, processor) ->
    if processor
      processor(existing or '')
    else
      existing
  processLine: (text) ->
    @processors.reduce @reducer, text
  processString: (text) ->
    (@processLine(line) for line in text.split("\n")).join("\n")

exampleText = '''
          # A level 1 header
          A regular line
          // a comment
```

```
        ## A level 2 header
        A line // with a comment
        '''

processor = new TextProcessor [stripComments, miniMarkdown]

processor.processString exampleText

 # => "<h1>A level 1 header</h1>\n<p>A regular line</p>\n\n<h2>A level 2 header</h2>\n<p>A line</p>"
```

结果

```
<h1>A level 1 header</h1>
<p>A regular line</p>

<h2>A level 1 header</h2>
<p>A line</p>
```

讨论

TextProcessor 服务有修饰的作用，可将个人、专业文本处理器绑定在一起。这使 miniMarkdown 和 stripComments 组件只专注于处理一行文本。未来的开发人员只需要编写函数返回一个字符串，并将它添加到阵列的处理器即可。

我们甚至可以修改现有的修饰对象动态：

```
smilies =
  ':)' : "smile"
  ':D' : "huge_grin"
  ':(' : "frown"
  ';)' : "wink"

smilieExpander = (line) ->
  if line
    (line = line.replace symbol, "<img src='#{text}.png' alt='#{text}' />") for symbol, text of smilies
  line

processor.processors.unshift smilieExpander

processor.processString "# A header that makes you :) // you may even laugh"

 # => "<h1>A header that makes you <img src='smile.png' alt='smile' /></h1>"
```

```
processor.processors.shift()

 # => "<h1>A header that makes you :)</h1>"
```

工厂方法模式

问题

直到开始运行你才知道需要的是什么种类的对象。

解决方案

使用 工厂方法（Factory Method）模式和选择对象都是动态生成的。

你需要将一个文件加载到编辑器，但是直到用户选择文件时你才知道它的格式。一个类使用工厂方法（Factory Method）模式可以根据文件的扩展名提供不同的解析器。

```
class HTMLParser
  constructor: ->
    @type = "HTML parser"
class MarkdownParser
  constructor: ->
    @type = "Markdown parser"
class JSONParser
  constructor: ->
    @type = "JSON parser"

class ParserFactory
  makeParser: (filename) ->
    matches = filename.match /\.(\w*)$/
    extension = matches[1]
    switch extension
      when "html" then new HTMLParser
      when "htm" then new HTMLParser
      when "markdown" then new MarkdownParser
      when "md" then new MarkdownParser
      when "json" then new JSONParser

factory = new ParserFactory

factory.makeParser("example.html").type # => "HTML parser"

factory.makeParser("example.md").type # => "Markdown parser"
```

```
factory.makeParser("example.json").type # => "JSON parser"
```

讨论

在这个示例中，你可以关注解析的内容，忽略细节文件的格式。更先进的工厂方法，例如，搜索版本控制文件中的数据本身，然后返回一个更精确的解析器(例如，返回一个 HTML5 解析器而不是 HTML v4 解析器)。

解释器模式

问题

其他人需要以控制方式运行你的一部分代码。相对地，你选择的语言不能以一种简洁的方式表达问题域。

解决方案

使用解释器模式来创建一个你翻译为特定代码的领域特异性语言（ domain-specific language ）。

我们来做个假设，例如用户希望在你的应用程序中执行数学运算。你可以让他们正向运行代码来演算指令（ eval ）但这会让他们运行任意代码。相反，你可以提供一个小型的“堆栈计算器（ stack calculator ）”语言，用来做单独分析，以便只运行数学运算，同时报告更有用的错误信息。

```
class StackCalculator
  parseString: (string) ->
    @stack = [ ]
    for token in string.split /\s+/
      @parseToken token

    if @stack.length > 1
      throw "Not enough operators: numbers left over"
    else
      @stack[0]

  parseToken: (token, lastNumber) ->
    if isNaN parseFloat(token) # Assume that anything other than a number is an operator
      @parseOperator token
    else
      @stack.push parseFloat(token)

  parseOperator: (operator) ->
    if @stack.length < 2
      throw "Can't operate on a stack without at least 2 items"

    right = @stack.pop()
    left = @stack.pop()

    result = switch operator
      when "+" then left + right
```

```
            when "-" then left - right
            when "*" then left * right
            when "/"
              if right is 0
                throw "Can't divide by 0"
              else
                left / right
            else
              throw "Unrecognized operator: #{operator}"

        @stack.push result

calc = new StackCalculator

calc.parseString "5 5 +" # => { result: 10 }

calc.parseString "4.0 5.5 +" # => { result: 9.5 }

calc.parseString "5 5 + 5 5 + *" # => { result: 100 }

try
  calc.parseString "5 0 /"
catch error
  error # => "Can't divide by 0"

try
  calc.parseString "5 -"
catch error
  error # => "Can't operate on a stack without at least 2 items"

try
  calc.parseString "5 5 5 -"
catch error
  error # => "Not enough operators: numbers left over"

try
  calc.parseString "5 5 5 foo"
catch error
  error # => "Unrecognized operator: foo"
```

讨论

作为一种替代编写我们自己的解释器的选择，你可以将现有的 CoffeeScript 解释器与更自然的（更容易理解的）表达自己的算法的正常方式相结合。

```
class Sandwich
  constructor: (@customer, @bread='white', @toppings=[], @toasted=false)->

white = (sw) ->
  sw.bread = 'white'
  sw

wheat = (sw) ->
  sw.bread = 'wheat'
  sw

turkey = (sw) ->
  sw.toppings.push 'turkey'
  sw

ham = (sw) ->
  sw.toppings.push 'ham'
  sw

swiss = (sw) ->
  sw.toppings.push 'swiss'
  sw

mayo = (sw) ->
  sw.toppings.push 'mayo'
  sw

toasted = (sw) ->
  sw.toasted = true
  sw

sandwich = (customer) ->
  new Sandwich customer

to = (customer) ->
  customer

send = (sw) ->
  toastedState = sw.toasted and 'a toasted' or 'an untoasted'

  toppingState = ''
  if sw.toppings.length > 0
    if sw.toppings.length > 1
      toppingState = " with #{sw.toppings[0..sw.toppings.length-2].join ', '} and #{sw.toppings[sw.toppings.length-1]}"
    else
```

```
        toppingState = " with #{sw.toppings[0]}"
    "#{sw.customer} requested #{toastedState}, #{sw.bread} bread sandwich#{toppingState}"

send sandwich to 'Charlie' # => "Charlie requested an untoasted, white bread sandwich"
send turkey sandwich to 'Judy' # => "Judy requested an untoasted, white bread sandwich with turkey"
send toasted ham turkey sandwich to 'Rachel' # => "Rachel requested a toasted, white bread sandwich with turkey and h
send toasted turkey ham swiss sandwich to 'Matt' # => "Matt requested a toasted, white bread sandwich with swiss, ham
```

这个实例可以允许功能层实现返回修改后的对象，从而外函数可以依次修改它。示例通过借用动词和介词的用法，把自然语法提供给结构，当被正确使用时，会像自然语句一样结束。这样，利用 CoffeeScript 语言技能和你现有的语言技能可以帮助你关于捕捉代码的问题。

备忘录模式

问题

你想预测对一个对象做出改变后的反应。

解决方案

使用备忘录模式（Memento Pattern）来跟踪一个对象的变化。使用这个模式的类会输出一个存储在其他地方的备忘录对象。

如果你的应用程序可以让用户编辑文本文件，例如，他们可能想要撤销上一个动作。你可以在用户改变文件之前保存文件现有的状态，然后回滚到上一个位置。

```
class PreserveableText
  class Memento
    constructor: (@text) ->

  constructor: (@text) ->
  save: (newText) ->
    memento = new Memento @text
    @text = newText
    memento
  restore: (memento) ->
    @text = memento.text

pt = new PreserveableText "The original string"
pt.text # => "The original string"

memento = pt.save "A new string"
pt.text # => "A new string"

pt.save "Yet another string"
pt.text # => "Yet another string"

pt.restore memento
pt.text # => "The original string"
```

讨论

备忘录对象由 PreserveableText#save 返回，为了安全保护，分别地存储着重要的状态信息。你可以序列化备忘录以便来保证硬盘中的“撤销”缓冲或者是那些被编辑的图片等数据密集型对象。

观察者模式

问题

当一个事件发生时你不得不向一些对象发布公告。

解决方案

使用观察者模式（Observer Pattern）。

```
class PostOffice
    constructor: () ->
        @subscribers = []
    notifyNewItemReleased: (item) ->
        subscriber.callback(item) for subscriber in @subscribers when subscriber.item is item
    subscribe: (to, onNewItemReleased) ->
        @subscribers.push {'item':to, 'callback':onNewItemReleased}

class MagazineSubscriber
    onNewMagazine: (item) ->
        alert "I've got new "+item

class NewspaperSubscriber
    onNewNewspaper: (item) ->
        alert "I've got new "+item

postOffice = new PostOffice()
sub1 = new MagazineSubscriber()
sub2 = new NewspaperSubscriber()
postOffice.subscribe "Mens Health", sub1.onNewMagazine
postOffice.subscribe "Times", sub2.onNewNewspaper
postOffice.notifyNewItemReleased "Times"
postOffice.notifyNewItemReleased "Mens Health"
```

讨论

这里你有一个观察者对象（ PostOffice ）和可观察对象(MagazineSubscriber, NewspaperSubscriber ）。为了通报发布新的周期性可观察对象的事件，应该对 PostOffice 进行订阅。每一个被订阅的对象都存储在 PostOffice 的内部订阅数组中。当新的实体周期发布时每一个订阅者都会收到通知。

单件模式

问题

许多时候你想要一个，并且只要一个类的实例。比如，你可能需要一个创建服务器资源的类，并且你想要保证使用一个对象就可以控制这些资源。但是使用时要小心，因为单件模式可以很容易被滥用来模拟不必要的全局变量。

解决方案

公有类只包含获得一个实例的方法。实例被保存在该公共对象的闭包中，并且总是有返回值。

这很奏效因为 CoffeeScript 允许你在一个类的声明中定义可执行的状态。但是，因为大多数 CoffeeScript 编译成一个 IIFE 包，如果这个方式适合你，你就不需要在类的声明中放置私有的类。之后的内容可能对开发模块化代码有所帮助，例如 CommonJS（Node.js）或 Require.js 中可见（见实例讨论）。

```
class Singleton
  # You can add statements inside the class definition
  # which helps establish private scope (due to closures)
  # instance is defined as null to force correct scope
  instance = null
  # Create a private class that we can initialize however
  # defined inside this scope to force the use of the
  # singleton class.
  class PrivateClass
    constructor: (@message) ->
    echo: -> @message
  # This is a static method used to either retrieve the
  # instance or create a new one.
  @get: (message) ->
    instance ?= new PrivateClass(message)

a = Singleton.get "Hello A"
a.echo() # => "Hello A"

b = Singleton.get "Hello B"
b.echo() # => "Hello A"

Singleton.instance # => undefined
```

```
a.instance # => undefined
Singleton.PrivateClass # => undefined
```

讨论

通过上面的实例我们可以看到，所有的实例是如何从同一个 Singleton 类的实例中输出的。你也可以看到，私有类和实例变量都无法在 Singleton class 外被访问到。 Singleton class 的本质是提供一个静态方法得到只返回一个私有类的实例。它也对外界也隐藏私有类，因此你无法创建一个自己的私有类。

隐藏或使私有类在内部运作的想法是更受偏爱的。尤其是由于缺省的 CoffeeScript 将编译的代码封装在自己的 IIFE（闭包）中，你可以定义类而无须担心会被文件外部访问到。在这个实例中，注意，用惯用的模块导出特点来强调模块中可被公共访问的部分。（请看 “导出到全局命名空间” 中对此理解更深入的讨论）。

```
root = exports ? this

 # Create a private class that we can initialize however

 # defined inside the wrapper scope.

class ProtectedClass
  constructor: (@message) ->
  echo: -> @message

class Singleton
  # You can add statements inside the class definition
  # which helps establish private scope (due to closures)
  # instance is defined as null to force correct scope
  instance = null
  # This is a static method used to either retrieve the
  # instance or create a new one.
  @get: (message) ->
    instance ?= new ProtectedClass(message)

 # Export Singleton as a module

root.Singleton = Singleton
```

我们可以注意到 coffeescript 是如此简单地实现这个设计模式。为了更好地参考和讨论 JavaScript 的实现，请看初学者必备 JavaScript 设计模式。

策略模式

问题

解决问题的方式有多种，但是你需要在程序运行时选择（或是转换）这些方法。

解决方案

在策略对象（Strategy objects）中封装你的算法。

例如，给定一个未排序的列表，我们可以在不同情况下改变排序算法。

基类

```
StringSorter = (algorithm) ->
  sort: (list) -> algorithm list
```

策略

```
bubbleSort = (list) ->
  anySwaps = false
  swapPass = ->
    for r in [0..list.length-2]
      if list[r] > list[r+1]
        anySwaps = true
        [list[r], list[r+1]] = [list[r+1], list[r]]

  swapPass()
  while anySwaps
    anySwaps = false
    swapPass()
  list

reverseBubbleSort = (list) ->
  anySwaps = false
  swapPass = ->
    for r in [list.length-1..1]
      if list[r] < list[r-1]
```

```
            anySwaps = true
            [list[r], list[r-1]] = [list[r-1], list[r]]

    swapPass()
    while anySwaps
        anySwaps = false
        swapPass()
    list
```

使用策略

```
sorter = new StringSorter bubbleSort

unsortedList = ['e', 'b', 'd', 'c', 'x', 'a']

sorter.sort unsortedList

 # => ['a', 'b', 'c', 'd', 'e', 'x']


unsortedList.push 'w'

 # => ['a', 'b', 'c', 'd', 'e', 'x', 'w']


sorter.algorithm = reverseBubbleSort

sorter.sort unsortedList

 # => ['a', 'b', 'c', 'd', 'e', 'w', 'x']
```

讨论

“没有作战计划在第一次接触敌人时便能存活下来。” 用户如是，但是我们可以运用从变化的情况中获得的知识来做出适应改变。在示例末尾，例如，数组中的最新项是乱序排列的，知道了这个细节，我们便可以通过切换算法来加速排序，只要简单地重赋值就可以了。

练习

？ 将 StringSorter 扩展为 AlwaysSortedArray 类来实现规则序列的所有功能，但是要基于插入方法自动分类新的项（例如 push 对比 shift）。

模板方法模式

问题

定义一个算法的结构，作为一系列的高层次的步骤，使每一个步骤的行为可以指定，使属于一个族的算法都具有相同的结构但是有不同的行为。

解决方案

使用模板方法（ Template Method ）在父类中描述算法的结构，再授权一个或多个具体子类来具体地进行实现。

例如，想象你希望模拟各种类型的文件的生成，并且每个文件要包含一个标题和正文。

```
class Document
  produceDocument: ->
    @produceHeader()
    @produceBody()

  produceHeader: ->
  produceBody: ->

class DocWithHeader extends Document
  produceHeader: ->
    console.log "Producing header for DocWithHeader"

  produceBody: ->
    console.log "Producing body for DocWithHeader"

class DocWithoutHeader extends Document
  produceBody: ->
    console.log "Producing body for DocWithoutHeader"

docs = [new DocWithHeader, new DocWithoutHeader]
doc.produceDocument() for doc in docs
```

讨论

在这个实例中，算法用两个步骤来描述文件的生成：其一是产生文件的标题，另一步是生成文件的正文。父类中是实现每一个步骤的空的方法，多态性使得每一个具体的子类可以通过重写一步步的方法来实现对方法不同的利用。在本实例中，DocWithHeader 实现了正文和标题的步骤， DocWithoutHeader 只是实现了正文的步骤。

不同类型文件的生成就是简单的将文档对象存储在一个数组中，简单的遍历每个文档对象并调用其 produceDocument 方法的问题。

14

数据库

MongoDB

问题

你需要与一个 MongoDB 数据库连接的接口。

解决方案

对于 Node.js

安装

？ 如果你的计算机中还没有 MongoDB ，需要安装。

？ 安装本地 MongoDB 模块。

保存记录

```
mongo = require 'mongodb'

server = new mongo.Server "127.0.0.1", 27017, {}

client = new mongo.Db 'test', server, {w:1}

 # save() updates existing records or inserts new ones as needed

exampleSave = (dbErr, collection) ->
   console.log "Unable to access database: #{dbErr}" if dbErr
   collection.save { _id: "my_favorite_latte", flavor: "honeysuckle" }, (err, docs) ->
      console.log "Unable to save record: #{err}" if err
      client.close()

client.open (err, database) ->
   client.collection 'coffeescript_example', exampleSave
```

查找记录

```
mongo = require 'mongodb'

server = new mongo.Server "127.0.0.1", 27017, {}

client = new mongo.Db 'test', server, {w:1}
```

```
exampleFind = (dbErr, collection) ->
  console.log "Unable to access database: #{dbErr}" if dbErr
  collection.find({ _id: "my_favorite_latte" }).nextObject (err, result) ->
    if err
      console.log "Unable to find record: #{err}"
    else
      console.log result # => {  id: "my_favorite_latte", flavor: "honeysuckle" }
    client.close()

client.open (err, database) ->
  client.collection 'coffeescript_example', exampleFind
```

对于浏览器

一个基于 REST 的接口在工程中，会提供基于 AJAX 的访问通道。

讨论

这个方法将 save 和 find 分开进单独的实例，其目的是分散 MongoDB 指定的连接任务的关注点以及回收任务。async 模块可以帮助这样的异步调用。

SQLite

问题

你需要 Node.js 内部与 SQLite 数据库连接的接口。

解决方案

使用 SQLite 模块。

```
sqlite = require 'sqlite'

db = new sqlite.Database

 # The module uses asynchronous methods,

 # so we chain the calls the db.execute

exampleCreate = ->
   db.execute "CREATE TABLE snacks (name TEXT(25), flavor TEXT(25))",
      (exeErr, rows) ->
         throw exeErr if exeErr
         exampleInsert()

exampleInsert = ->
   db.execute "INSERT INTO snacks (name, flavor) VALUES ($name, $flavor)",
      { $name: "Potato Chips", $flavor: "BBQ" },
      (exeErr, rows) ->
         throw exeErr if exeErr
         exampleSelect()

exampleSelect = ->
   db.execute "SELECT name, flavor FROM snacks",
      (exeErr, rows) ->
         throw exeErr if exeErr
         console.log rows[0] # => { name: 'Potato Chips', flavor: 'BBQ' }

 # :memory: creates a DB in RAM

 # You can supply a filepath (like './example.sqlite') to create/open one on disk
```

```
db.open ":memory:", (openErr) ->
  throw openErr if openErr
  exampleCreate()
```

讨论

你也可以提前准备你的 SQL 查询语句。

```
sqlite = require 'sqlite'
async = require 'async' # Not required but added to make the example more concise

db = new sqlite.Database

createSQL = "CREATE TABLE drinks (name TEXT(25), price NUM)"

insertSQL = "INSERT INTO drinks (name, price) VALUES (?, ?)"

selectSQL = "SELECT name, price FROM drinks WHERE price < ?"

create = (onFinish) ->
  db.execute createSQL, (exeErr) ->
    throw exeErr if exeErr
    onFinish()

prepareInsert = (name, price, onFinish) ->
  db.prepare insertSQL, (prepErr, statement) ->
    statement.bindArray [name, price], (bindErr) ->
      statement.fetchAll (fetchErr, rows) -> # Called so that it executes the insert
        onFinish()

prepareSelect = (onFinish) ->
  db.prepare selectSQL, (prepErr, statement) ->
    statement.bindArray [1.00], (bindErr) ->
      statement.fetchAll (fetchErr, rows) ->
        console.log rows[0] # => { name: "Mia's Root Beer", price: 0.75 }
        onFinish()

db.open ":memory:", (openErr) ->
  async.series([
    (onFinish) -> create onFinish,
    (onFinish) -> prepareInsert "LunaSqueeze", 7.95, onFinish,
    (onFinish) -> prepareInsert "Viking Sparkling Grog", 4.00, onFinish,
    (onFinish) -> prepareInsert "Mia's Root Beer", 0.75, onFinish,
```

```
    (onFinish) -> prepareSelect onFinish
])
```

SQL 的 SQLite 版本的以及 node-SQLite 模块文档提供了更完整的信息。

15

测试

使用 Jasmine 测试

问题

假如你正在使用 CoffeeScript 写一个简单地计算器，并且想要验证其功能是否与预期一致。可以使用 Jasmine 测试框架。

讨论

在使用 Jasmine 测试框架时，你要在一个参数（spec）文档中写测试，文档描述的是代码需要测试的预期功能。

例如，我们希望计算器可以实现加法和减法的功能，并且可以正确进行正数和负数的运算。我们的 spec 文档如下列所示。

```
 # calculatorSpec.coffee

describe 'Calculator', ->
  it 'can add two positive numbers', ->
    calculator = new Calculator()
    result = calculator.add 2, 3
    expect(result).toBe 5

  it 'can handle negative number addition', ->
    calculator = new Calculator()
    result = calculator.add -10, 5
    expect(result).toBe -5

  it 'can subtract two positive numbers', ->
    calculator = new Calculator()
    result = calculator.subtract 10, 6
    expect(result).toBe 4

  it 'can handle negative number subtraction', ->
    calculator = new Calculator()
    result = calculator.subtract 4, -6
    expect(result).toBe 10
```

配置 Jasmine

在你运行测试之前，必须要先下载并配置 Jasmine。包括：1.下载最新的 Jasmine 压缩文件；2.在你的项目工程中创建一个 spec 以及一个 spec/jasmine 目录；3.将下载的 Jasmine 文件解压到 spec/jasmine 目录中；4.创建一个测试流

创建测试流

Jasmine 可以使用 spec runner 的 HTML 文档在 web 浏览器中运行你的测试。 spec runner 是一个简单地 HTML 页面，连接着 Jasmine 以及你的代码所需要的必要的 JavaScript 和 CSS 文件。示例如下。

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
 "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
  <title>Jasmine Spec Runner</title>
  <link rel="shortcut icon" type="image/png" href="spec/jasmine/jasmine_favicon.png">
  <link rel="stylesheet" type="text/css" href="spec/jasmine/jasmine.css">
  <script src="http://code.jquery.com/jquery.min.js"></script>
  <script src="spec/jasmine/jasmine.js"></script>
  <script src="spec/jasmine/jasmine-html.js"></script>
  <script src="spec/jasmine/jasmine-jquery-1.3.1.js"></script>

  <!-- include source files here... -->
  <script src="js/calculator.js"></script>

  <!-- include spec files here... -->
  <script src="spec/calculatorSpec.js"></script>

</head>

<body>
  <script type="text/javascript">
    (function() {
      var jasmineEnv = jasmine.getEnv();
      jasmineEnv.updateInterval = 1000;

      var trivialReporter = new jasmine.TrivialReporter();

      jasmineEnv.addReporter(trivialReporter);

```

```
31      jasmineEnv.specFilter = function(spec) {
32        return trivialReporter.specFilter(spec);
33      };
34
35      var currentWindowOnload = window.onload;
36
37      window.onload = function() {
38        if (currentWindowOnload) {
39          currentWindowOnload();
40        }
41        execJasmine();
42      };
43
44      function execJasmine() {
45        jasmineEnv.execute();
46      }
47
48    })();
49   </script>
50 </body>
51 </html>
```

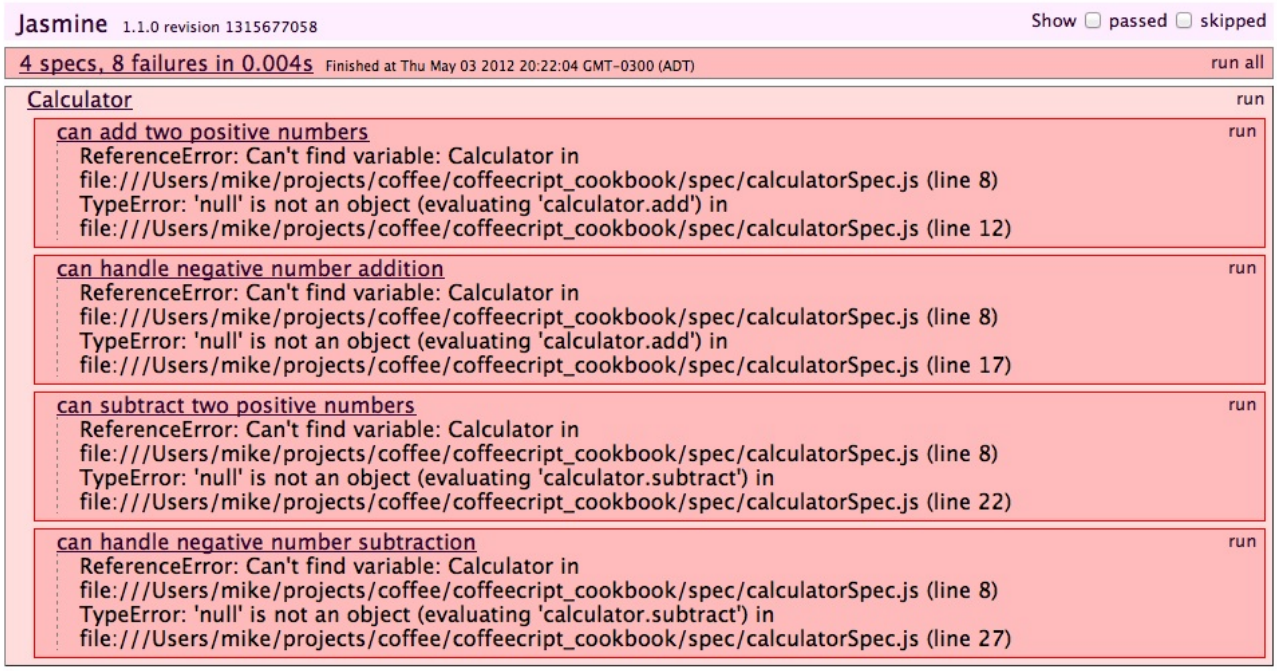
此 spec runner 可以在 GitHub gist 上下载。

使用 SpecRunner.html ，只是简单地参考你编译后的 JavaScript 文件，并且在 jasmine.js 以及其依赖项后编译的测试文件。

在上述示例中，我们在第 14 行包含了尚待开发的 calculator.js 文件，在第17行编译了 calculatorSpec.js 文件。

运行测试

要运行我们的测试，只需要简单地在 web 浏览器中打开 SpecRunner.html 页面。在我们的示例中可以看到 4 个失败的 specs 共 8 个失败情况（如下）。

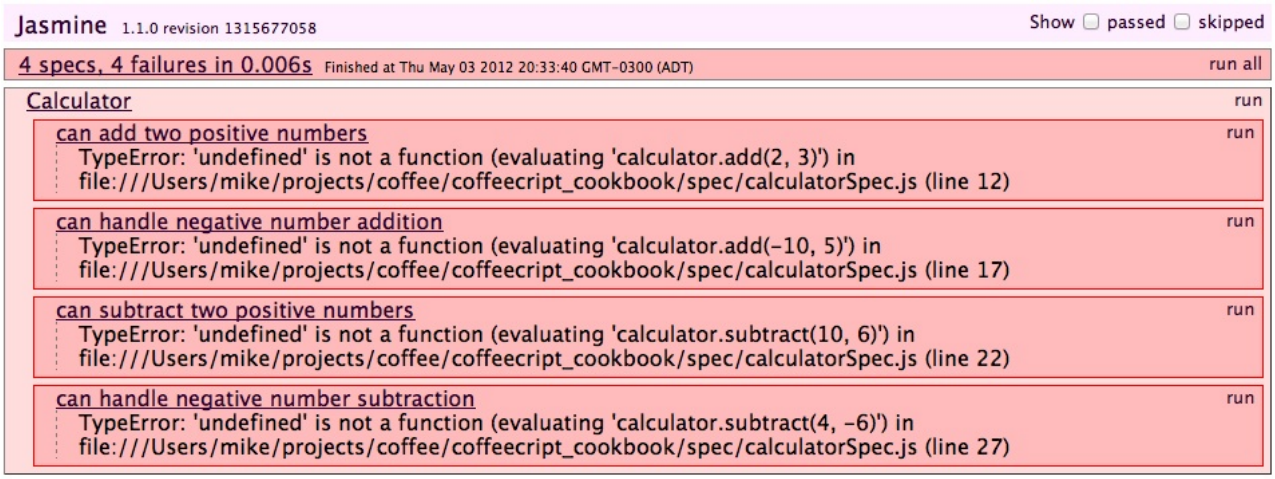


图片 15.1 Alt text

看来我们的测试是失败的，因为 jasmine 无法找到 Calculator 变量。那是因为它还没有被创建。现在让我们来创建一个新文件命名为 js/calculator.coffee 。

```
# calculator.coffee



window.Calculator = class Calculator
```

编译 calculator.coffee 并刷新浏览器来重新运行测试组。



图片 15.2 Alt text

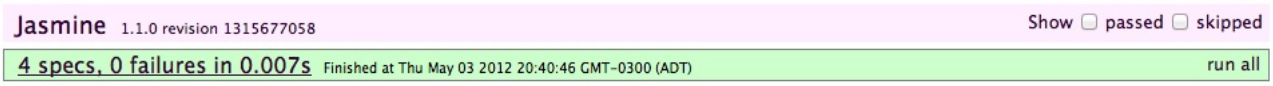
现在我们还有4个失败而不是原来的8个了，只用一行代码便做出了50%的改进。

测试通过

实现我们的方法来看是否可以通过测试。

```
# calculator.coffee


window.Calculator = class Calculator
  add: (a, b) ->
    a + b

  subtract: (a, b) ->
    a - b
```

当我们刷新页面时可以看到全部通过。



图片 15.3 Alt text

重构测试

既然测试全部通过了，我们应看一看我们的代码或测试是否可以被重构。

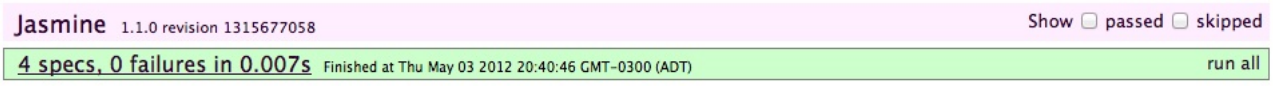
在我们的 spec 文件中，每个测试都创建了自己的 calculator 实例。这会使我们的测试相当的重复，特别是对于大型的测试套件。理想情况下，我们应该考虑将初始化代码移动到每次测试之前运行。

幸运的是 Jasmine 拥有一个 beforeEach 函数,就是为了这一目的设置的。

```
describe 'Calculator', ->
  calculator = null

  beforeEach ->
    calculator = new Calculator()

  it 'can add two positive numbers', ->
    result = calculator.add 2, 3
    expect(result).toBe 5

  it 'can handle negative number addition', ->
    result = calculator.add -10, 5
    expect(result).toBe -5
```

```
it 'can subtract two positive numbers', ->
    result = calculator.subtract 10, 6
    expect(result).toBe 4

it 'can handle negative number subtraction', ->
    result = calculator.subtract 4, -6
    expect(result).toBe 10
```

当我们重新编译我们的 spec 然后刷新浏览器，可以看到测试仍然全部通过。



图片 15.4 Alt text

使用 Nodeunit 测试

问题

假如你正在使用 CoffeeScript 并且想要验证功能是否与预期一致，便可以决定使用 Nodeunit 测试框架。

讨论

Nodeunit 是一种 JavaScript 对于单元测试库（ Unit Testing libraries ）中 xUnit 族的实现，Java, Python, Ruby, Smalltalk 中均可以使用。

当使用 xUnit 族测试框架时，你需要将所需测试的描述预期功能的代码写在一个文件中。

例如，我们希望我们的计算器可以进行加法和减法，并且对于正负数均可以正确计算，我们的测试如下。

```
 # test/calculator.test.coffee

Calculator = require '../calculator'
exports.CalculatorTest =
   'test can add two positive numbers': (test) ->
      calculator = new Calculator
      result = calculator.add 2, 3
      test.equal(result, 5)
      test.done()

   'test can handle negative number addition': (test) ->
      calculator = new Calculator
      result = calculator.add -10, 5
      test.equal(result,  -5)
      test.done()

   'test can subtract two positive numbers': (test) ->
      calculator = new Calculator
      result = calculator.subtract 10, 6
      test.equal(result, 4)
      test.done()

   'test can handle negative number subtraction': (test) ->
      calculator = new Calculator
      result = calculator.subtract 4, -6
```

```
    test.equal(result, 10)
    test.done()
```

安装 Nodeunit

在可以运行你的测试之前，你必须先安装 Nodeunit：

首先创建一个 package.json 文件

```
{
  "name": "calculator",
  "version": "0.0.1",
  "scripts": {
    "test": "./node_modules/.bin/nodeunit test"
  },
  "dependencies": {
    "coffee-script": "~1.4.0",
    "nodeunit": "~0.7.4"
  }
}
```

接下来从一个终端运行。

```
$ npm install
```

运行测试

使用代码行可以简便地运行测试文件：

```
$ npm test
```

测试失败，因为我们并没有 calculator.coffee

```
suki@Yuzuki:nodeunit_testing (master)$ npm test
npm WARN package.json calculator@0.0.1 No README.md file found!

> calculator@0.0.1 test /Users/suki/tmp/nodeunit_testing
> ./node_modules/.bin/nodeunit test



/Users/suki/tmp/nodeunit_testing/node_modules/nodeunit/lib/nodeunit.js:72
    if (err) throw err;
             ^
```

```
Error: ENOENT, stat '/Users/suki/tmp/nodeunit_testing/test'
npm ERR! Test failed.  See above for more details.
npm ERR! not ok code 0
```

我们创建一个简单文件

```
 # calculator.coffee


class Calculator

module.exports = Calculator
```

并且重新运行测试套件。

```
suki@Yuzuki:nodeunit_testing (master)$ npm test
npm WARN package.json calculator@0.0.1 No README.md file found!

> calculator@0.0.1 test /Users/suki/tmp/nodeunit_testing
> ./node_modules/.bin/nodeunit test


calculator.test
? CalculatorTest - test can add two positive numbers

TypeError: Object #<Calculator> has no method 'add'
  ...

? CalculatorTest - test can handle negative number addition

TypeError: Object #<Calculator> has no method 'add'
  ...

? CalculatorTest - test can subtract two positive numbers

TypeError: Object #<Calculator> has no method 'subtract'
  ...

? CalculatorTest - test can handle negative number subtraction

TypeError: Object #<Calculator> has no method 'subtract'
  ...


FAILURES: 4/4 assertions failed (31ms)
```

```
npm ERR! Test failed.  See above for more details.
npm ERR! not ok code 0
```

通过测试

让我们对方法进行实现来观察测试是否可以通过。

```
 # calculator.coffee


class Calculator

  add: (a, b) ->
    a + b

  subtract: (a, b) ->
    a - b

module.exports = Calculator
```

当我们重新运行测试时可以看到全部通过：

```
suki@Yuzuki:nodeunit_testing (master)$ npm test
npm WARN package.json calculator@0.0.1 No README.md file found!

> calculator@0.0.1 test /Users/suki/tmp/nodeunit_testing
> ./node_modules/.bin/nodeunit test


calculator.test
? CalculatorTest - test can add two positive numbers
? CalculatorTest - test can handle negative number addition
? CalculatorTest - test can subtract two positive numbers
? CalculatorTest - test can handle negative number subtraction

OK: 4 assertions (27ms)
```

重构测试

既然测试全部通过，我们应看一看我们的代码或测试是否可以被重构。

在我们的测试文件中，每个测试都创建了自己的 calculator 实例。这会使我们的测试相当的重复，特别是对于大型的测试套件。理想情况下，我们应该考虑将初始化代码移动到每次测试之前运行。

通常在其他的 xUnit 库中，Nodeunit 会提供一个 setUp（以及 tearDown ）功能会在测试前调用。

```
Calculator = require '../calculator'

exports.CalculatorTest =

  setUp: (callback) ->
    @calculator = new Calculator
    callback()

  'test can add two positive numbers': (test) ->
    result = @calculator.add 2, 3
    test.equal(result, 5)
    test.done()

  'test can handle negative number addition': (test) ->
    result = @calculator.add -10, 5
    test.equal(result,  -5)
    test.done()

  'test can subtract two positive numbers': (test) ->
    result = @calculator.subtract 10, 6
    test.equal(result, 4)
    test.done()

  'test can handle negative number subtraction': (test) ->
    result = @calculator.subtract 4, -6
    test.equal(result, 10)
    test.done()
```

我们可以重新运行测试，仍然可以全部通过。