别看着是一个英文的标题，其实是中文文章滴。可以去看我的博客上的全部，分三篇：  
<http://blog.csdn.net/aimingoo/archive/2009/09/08/4532394.aspx>  
<http://blog.csdn.net/aimingoo/archive/2009/09/08/4532496.aspx>  
<http://blog.csdn.net/aimingoo/archive/2009/09/08/4532567.aspx>  
  
这里取出里面讲DSL设计部分的两篇。部分内容，需要参见以前关于QoBean元编程的文章：  
<http://blog.csdn.net/aimingoo/archive/2008/07/25/2707504.aspx>  
<http://blog.csdn.net/aimingoo/archive/2008/07/25/2709638.aspx>  
  
  
  
一、QoBean的元编程框架  
  
现在开讲“DSL in JavaScript，with QoBean’s meta programming framework”。先说说QoBean的元编程框架。这是一个仅仅100来行的小小框架，里面有几个关键函数是我们在讲DSL时要用到的：  
  
1、Weave(where, code)：表示对于一个函数f，在指定where的位置，替换成代码code。如果where是正则表达式，则code中可以使用能用匹配；如果where是字符串，则表明将第一个查找到的该字符串替换成code。  
  
2、Block(func, tag)：对函数func（或直接是它的代码文本）进行结构分析。tag标志为['body', 'param', 'name']之一时，返回该函数源代码相应位置的一个字符串；tag标志为’scope’时，返回一个字符串str，用eval(str)可以在当前位置生成一个新的、具有该代码上下文位置上的闭包的函数；tag标志为’anonymous’时，直接返回一个函数，但该函数是全局函数。对于tag是['scope','anonymous']之一的情况，所得到的新函数是传入参数func的一个完全相同的副本，只是新函数所在的闭包位置不同。  
  
3、Unique(obj)：生成obj对象的一个唯一化的实例。新实例obj2具有原obj对象的全部属性，但修改新对象obj2的成员时，不会影响到原obj对象。这个过程，与Block(func, tag)调用时，tag为['scope','anonymous']之一的情况有类似之处：都是生成一个新的复制，不同的是Block针对函数，Unique针对对象。  
  
4、Scope(obj, func)：使函数func（或直接是它的代码文本）执行在obj的对象闭包中。当func执行在obj的闭包中之后，func所访问到的变量名，即是obj的属性；func访问到的函数名，即是obj的方法。  
  
5、Owner(obj, func)：使函数func（或直接是它的代码文本）执行在以obj为this对象的环境中。在JS中，当一个函数执行时，this关键字要么指向全局的window对象（func是普通函数）；要么指向方法所属的对象实例（func是对象方法）。Owner()函数用于改变这种关系，使得函数func在执行时，this关键字总是指向obj。——一般来说，这个功能在其它的JS框架中被实现为Function.prototype.bind()。  
  
QoBean的元编程系统中还有一些其它的函数，但对于我们讲DSL不关键，所以先不讨论。除了上述讨论的函数之外，我们在DSL部分不会用到其它的任何函数，也不会用到某种JavaScript引擎的独特功能——甚至不会用到arguments.caller。所以，我们在这里实现的DSL，可以跑在ECMS Script标准中的任何一个JavaScript引擎之中。  
  
最后综述一下上面的五个函数。他们其实对于Meta Programming有着特殊的含义。函数：  
  
1、Block()，其实是一个简单的Parser。他能够快速分析一个代码文本块，以使重新组织它。相当于我们在书写代码时的重构、改写。  
  
2、Unique()与Scope()是功能近似的函数，只是一个作用于函数，一个作用于对象。作用是得到一个复制，相当于我们书写代码时的ctrl + C操作。  
  
3、Scope()与Owner()也是功能近似的函数，前者用于改变代码的上下文，相当于书写代码时移动一段代码（例如把局部函数变成全局公共的），有点类似于ctrl+V操作；后者用于改变代码（作为对象方法时）的属主，相当于基类中的方法，或重构，或范型等等。  
  
我们注意到，上面几个函数，事实上模拟了我们书写代码时的很多行为。在前面讲到过的文章中，我说到过“那么QoBean如何定义‘元语言’呢？QoBean对此有两项解释”：  
  – 元语言定义程序(program)的基础元素：算法与数据结构  
  – 元语言说明编程(programming)的基本方法：代码的组织形式  
  
这里的五个函数，就是第二项解释中的“编程（书写程序）的基本方法”的抽象、实现。  
  
  
  
二、DSL的基本设计  
  
现在我们来考虑一个“通用DSL”应该是什么样子，也就是如何设计它的问题。首先，它是一种语言——这很废话对吧？哈哈。其实不是，这是一个语言，表明它应该有语法、语义、语用的问题。语法就意味着需要一个解析器（parser）；语义就意味着对于语言中的关键字要有功能实现，即要有执行器（evaluator）；语用，就意味着说相同的话——相同的代码文本，在不同的环境下效果未必一致，所以请用就意味着要有环境（environment）设定，亦即是“上下文相关文法”或“上下文无关文法”的问题。  
  
这三个方面的问题有点令人挠头，但用个类比，就挺简单的。例如说“吃饭了吗”这句话，首先就包括汉语语法的问题，例如省略主语、疑问句和主谓结构等等。所以，我们可以改变一种新语法来陈述它，例如“饭，吃了吗”，或“吃了吗，饭”。这些，只是语法上的变化。说话的、写程序的人，先约定一种规则，然后按这个规则来理解它，就行了。  
  
那么语用呢？或者说所谓的“上下文相关/无关”是什么意思呢？同样的，上面这句话，如果是早晨我跟你碰面在公司楼下，我问这句话的意义，跟说“Hello”，或者“今天天气不错”其实差不多，只是个问候语。但如果是朱镕基同学在汶川问某个老乡，那可就真是问“有饭吃没”。话是一样的话，语法是一样的语法，放在不同的环境中，语义上是有差别的。这个，就是语用学讨论的问题。我们用的计算机很笨，没有人那么复杂的思维能力，所以一般来说，要求我们设计的语言是“上下文无关的”，以便于将来开发出来一个机器人，你问“吃饭没”，他真的能回答你“电能充足”，而不是做泪流满面状。  
  
所以，回到DSL的设计上来。所谓一个语言，也就是“通过某种规则来解析(parser)一段文本，将它执行(evaluator)在某个上下文环境（environment）”中。这个体系中，有一个东西是不变的，也就parser/evaluator/environment的关系。所以，一个新的dsl语言的产生过程，可以描述成这样的一个模式：  
  
dsl = DSL(environment, evaluator, parser)  
  
而这个语言执行——或说是讲述、表达、运行、生效——起来，则可以描述成下面这样一个模式：  
  
information = dsl(….)  
  
至于这个语言的规则部分，是parser负责的；表述效果部分，是evaluator负责的。而要让JavaScript DSL对这个新的dsl起来“维护”作用的，应该是对语境的（语用的）设定——简单的说，我们要帮助新的语言管理上下文环境，其它的则由“创建语言”的人来做。  
  
  
  
三、DSL的基本实现  
  
由于在  
  
dsl = DSL(environment, evaluator, parser)  
  
中，dsl最终是需要有调用能力的（表达成”dls(…)”的形式。所以  
1、DSL函数应该返回一个函数，并且它被执行在environment环境中；  
2、考虑到dsl应该与JavaScript的环境无关，它的this对象应该指向environment而非window。  
  
而上述两件事情，在QoBean meta programming中用两个函数来实现，即：  
===  
function DSL(environment, evaluator, parser) {     
  var dsl = Scope(environment,  …);  
  return Owner(environment, dsl);  
}  
  
===  
  
OK. 这个结构基本就完成了。但是还有个问题，就是上面Scope()调用中的“…”，他表明我们要在environment中执行代码的内容和方法。如何执行呢？简单的说，就是“先分析输入的代码，然后调用执行器执行它”。我们把这两个步骤，我们可以：  
  
1、把它连在一起，  
2、放在environment中去run一下。  
  
就好了。这个“连在一起”的事情，就是Weave()函数能做的。所以完整的DSL()函数的代码是下面这样两：  
  
===  
function DSL(environment, evaluator, parser) {  
  var dsl = Scope(environment, Weave.call(evaluator, /^/, Block(parser, ‘body’)+’\n\n’));  
   return Owner(environment, dsl);  
}  
===  
  
其中的Weave()调用表明：  
  
1、将parser()函数的body部分，放到evaluator()函数的开始部分之前执行；  
2、将上述的结果（代码文本），放在environment的scope闭包中执行。  
  
OK。我们的“通用DSL语言生成器”就做完了。它只有两行代码。  
  
  
  
四、示例  
  
这样的一个示例其实很简单。比如说我们想要有一种语言，它具有如下的性质：  
  
1、可以调用一些函数；  
2、可以访问一些预定义的值；  
3、（为了方便），我们假定它跟JavaScript的基本语法是一样的。  
  
我们简单的用DSL来实现一下它，以便对这个QoBean’s DSL framework有个概要认识。它还相当不完善，下一段落里，我们再来补全这个DSL系统。  
  
实现上面的这个小小的语言的方案是：  
  
==============  
// 1. 声明一个环境对象  
Env = {  
  language: ‘langg’,  
  max: 100,  
  min: -3,  
  calc: function(adj) {  
    return adj \* 2  
  },  
  show: function(msg) {  
    alert(msg)  
  }  
}  
  
// 2. 基本的执行器  
function myeval(source) {  
  return eval(source);  
}  
  
// 3. 基本的解析器  
function myparser(source){  
    source = Block(source);  
}  
  
// 4. 构建一个新的dsl  
myenv = Unique(Env);  
dsl = DSL(myenv, myeval, myparser);  
  
// 5. 执行一下下？   
dsl(function() {  
  show(min+max);  
  show(calc(min+max));  
});  
==============  
  
OK。结果出来了，显示97、194两个值。对于dsl()后面执行的代码来说，环境environment为他们准备了min/max/show/calc这四个标识符；myeval()提供了执行能力；myparser()则用于将dsl()调用传入的函数中的代码块取出来——之所以传入一个函数，是这样一来，就可以省了一个“语法分析器”（能当函数传入，当然是能通过JavaScript的语法分析过程的）。  
  
先到这里，吃饭去也。下午来继续扩充这个过程，然后我们就知道一个完整的……相当完整的DSL()实现……其实也并不复杂了。

五、这个DSL框架有什么问题？  
  
有什么问题吗？有一点，并不严重。比如说，我们在Env中声明了一些属性和方法。对于Env这个对象  
====  
Env = {  
  max: 100,  
  min: -3,  
  calc: function(adj) {  
    …  
  }  
}  
====  
  
我们要在calc()方法中访问max/min属性，应该写成“this.max/this.min”，这一则是不方便，另外，在用户的dsl代码中还不得不考虑“当前this是谁”的问题。这是问题之一。  
  
第二个，我们传入了一个evaluator(),相当于脚本执行器，那么我们能不能在dsl()代码中也使用这个执行器呢？也就是说，我们的dsl不单是“domain-specific language”，也可以是一个“domain-script language”的。  
  
第三个问题，我们是不是需要一个类似在JavaScript中的window对象的东西，以便能引用到执行环境的全局。  
  
这三个问题都应该是在DSL()层面解决的。简单说来，第二、三个问题，实质是在初始化环境environment，使之具有某些在dsl代码中能访问到的性质。所以很容易处理：  
====  
function DSL(environment, evaluator, parser) {  
  var dsl = Scope(environment, Weave.call(evaluator, /^/, Block(parser, ‘body’)+’\n\n’));  
  environment.system = environment;  
  environment[Block(evaluator, 'Name')] = dsl;  
  return Owner(environment, dsl);  
}  
====  
  
同理的，用户可以在上面这里对environment加入更多性质，这些都是可以在用户的dsl(…)中访问到的。以上面为例，当用户传入的执行器evaluator是一个具名函数的时候，则该函数名会成为dsl(…)环境中的可用的执行函数（类似于exec, execScript或eval等)。例如：  
====  
function myeval() { … }  
dsl = DSL(aEnv, myeval, aParser);  
dsl(function(){  
  myeval(…);  
});  
====  
注意在dsl()访问到的myeval()方法，其实不是用户原始的myeval()，而是上述dsl变量的一个引用。这个，从DSL()函数的实现中可以看到。  
  
接下来，就是上面三个问题中的第一个，亦即是在calc()方法从必须使用this.max/this.min的问题。事实上，这是因为声明calc方法的时候，该函数位于Env变量所在的全局闭包里面。这样，它就默认只能访问到全局的变量、标识符。所以，解决这个问题的方法，仍然和前面一样：改变它的闭包位置——使用Scope()函数。如下：  
====  
function DSL(environment, evaluator, parser) {  
  var dsl = Scope(environment, Weave.call(evaluator, /^/, Block(parser, ‘body’)+’\n\n’));  
  for (var n in environment) {  
    if (environment[n] instanceof Function) environment[n] = Scope(environment, environment[n]);  
  }  
  …  
}  
====  
  
现在有了一个新的、完善的DSL()。使用方法与前面是一致的。比如：  
====  
Env = {  
  max: 100,  
  calc: function(adj) { return max + adj },   //可以直接访问max了  
  show: function(msg) { alert(msg) }  
};  
  
dsl = DSL(Env, myeval, myparser);  
dsl(function() {  
  show(calc(30));   //显示130  
});  
====  
  
最后，留意一下当调用DSL()的时候，我们标出了”Env”这个全局变量。注意的是，我们直接使用了这个对象。那么它与使用Unique(Env)有什么不同呢？答案是，直接使用Env时，在dsl(…)中的代码可以直接修改到Env中的成员，而如果使用Unique(Env)，则dsl(…)中的代码只会修改到Env的一个副本。这样一来，我们就有机会为不同的dsl语言提供各各独立的环境了——这有点象沙箱。  
  
  
  
六、变量泄漏？  
  
在JavaScript语言中有一个“根深蒂固”的问题，就是“当在函数内访问一个不存在的变量时，引擎会试图在全局变量环境中打找该变量”。这通常是很多很多烂系统的根源。对于我们上面的dsl语言来说，系统其实只给出了五个标识符：max/calc/show/system/myeval。其中的后面两个，是DSL()函数在“语言引擎层面”提供的，其它的则是Env环境变量提供的。“变量泄漏”带来的直接问题是，对于上面的这个例子，dsl(…)中除了能访问这五个标识符之外，还能访问全局的window/String/Number/Math/RegExp/NaN等等预定义对象和属性。而这，可能根本就不是我们的dsl语言需要的。  
  
这怎么办呢？  
  
由于Unique()得到了Env环境对象的一个副本，而且在dsl(…)中无法通过这个副本来修改原始的Env的成员，也不能delete它。所以如果我们在Env的属性中加入这些“受保护的标识符”，那么dsl(…)就只能访问到Env的这些属性，而不会访问到全局里面的了。下面的代码简单地实现这一效果：  
  
=======  
Env = { … };  
protoected = ['window', 'setTimeout', 'setInterval',  //window和Global的成员...  
'Array', 'Object', 'Function',  // 全局的对象构造器...  
'null', 'undefined',   //引擎定义的，类似系统关键的...  
'Env', 'tinyParser', 'dsl', 'myeval' //用户代码环境中的...  
];  
protoected.forEach(function(item) { this[item] = undefined}, Env);  
  
dsl = DSL(Unique(Env), myeval, myparser);  
dsl(function() {  
  show(Array); // 显示undefined  
  Array = ‘local defined’;  
  show(Array); // 显示local defined  
});  
=======  
  
  
七、evaluator/parser是不是太简单了？  
  
当然。我们在evaluator, parser中基本什么也没有做，当然是相当简单的。如果你要做一个完整的DSL，那么你得花一些工夫来做语法解析，并实现在语法树的基础上的代码执行、运行环境的维护等等。我QoBean的DSL()中，主要是提供了一个运行你的代码的基础语言环境，有点象是——嗯——沙箱。  
  
当然，除了沙箱的基本功能之外。DSL()通过environment来维护给用户代码的一组基本标识符（或称为保留字），并保证用户在不同的environment之间不会相互影响。  
  
除了上述的基本描述之外，我们最后再关注一下evaluator和parser的实现。对于下面的代码：  
========  
function myeval(source) {  
  return eval(source);  
}  
  
function myparser(source){  
  source = Block(source);  
}  
  
dsl(function() {  
  show(min+max);  
  show(calc(min+max));  
});  
========  
  
实际上的效果是dsl()将红色显示部分的函数作为一个一个参数source，传入myparser()和myeval()。parser通过Block()取出这个函数代码的body部分，然后交给myeval()中的eval()函数执行。也就是说，我们在DSL()中调用Weave()的效果就是，将myparser()和myeval()并在一起，变成了：  
========  
function(source) {  
  source = Block(source);  
  return eval(source);  
}  
========  
而dsl()最终执行的就是上面这个匿名函数。更进一步，在environment上也会有一个名为’myeval’的方法，指向这个匿名函数。  
  
但是，首先这里就有一个不小的问题：’source’在这里也是一个标识符。在eval(…)中执行时，代码是可以感知到这个标识符的——而对于dsl(…)中的用户代码，source可能是另外需要的一个标识符，所以这里我们要想办法屏蔽掉对这个变量名的依赖。这其实处理起来很简单：  
========  
function myeval(source) {  
  return eval(arguments[0]);  
}  
  
function myparser(source){  
  arguments[0] = Block(arguments[0]);  
}  
=========  
  
你应该注意到，我们用arguments[0]就可以简单地绕过一个入口参数名的使用了。这个，很简单，也很实用。  
  
接下来，我们总不能要求用户每次执行dsl(…)时都要传入一个函数吧？我们最终声明的用户的DSL可能是相当怪异的、完全不符合JS的语法的，根本就不能写到一个函数中去，又该怎么办呢？这个问题，显然的——首先的——他该是parser的问题。因此我们也就简单地讲一下扩充myparser()的方法。比如说，我们想实现下面的效果：  
  
1、当dsl(…)传入一个字符串时，让myeval()直接执行该字符串；  
2、当dsl(…)传入一个函数，但函数体内是完整的整块注释时，让myeval()执行这个注释块。  
  
例如如下的调用：  
=========  
// 示例1  
dsl(”\  
  apple.more->hi(form) % \  
  tree.clear+>do(function() .. ). \  
“);  
  
//示例2  
dsl(function(){/\*  
  apple.more->hi(form) %  
  tree.clear+>do(function() .. ).  
\*/});  
=========  
上面两个示例中都有一个相同的dsl代码片断——这是一种假想的、完全不符合javascript的规范的新语言。示例1是通过一个字符串传给dsl()的，示例2仍然是通过一个函数，但函数体内是从/\*..\*/的一个注释块。  
  
现在我们需要进一步完善我们的myparser()，提供一个基本的模式来支持这种设计。简单的方法如下：  
=========  
function tinyParser(){  
  switch (typeof arguments[0]) {  
  case ‘function’:  
    arguments[0] = Block(arguments[0]);  
    arguments[0] = arguments[0].replace(/^\s\*\/\\*([\d\D]\*)\\*\/\s\*$/, ‘$1′);  
    break;  
  }  
/\* 现在你需要  
    1、对字符串arguments[0]进行语法分析，形成语法树或符号某种规则的代码块,  
    2、将结果传回arguments[0]。  
\*/  
}  
=========  
  
当然，由于代码的语法规则改变了，所以myeval()的设计也应该发生相应的变化了。而这些，就应该是DSL语言设计者的工作，而不是QoBean在DSL()框架上要考虑的事情了。  
  
  
  
八、终结：DSL，关键不在用什么语言实现，而在于为什么Domain设计什么样的语言  
  
我们用Javascript，只写了不到了10行代码，就实现了一个DSL()的通用框架，但是，我们却没有做出对任何一个真实的Domain有意义的DSL。对于Ruby、Python、Erlang还是Scala，或者更原始的LISP或更新的F#这些基础语言，对他们的选择更多的只是喜好或者出于某些局部的优异与方便的考虑，与我们“设计一个DSL”是没有多大的关系的。一个DSL的设计，在于对领域的、领域相关业务的分析与抽象。在这些分析、抽象的基础上，进行语法设计、语义定义，最终才表现为“怎样的一个语言”。当我们看到这个“表现”的时候，整个DSL的设计都已经结束了——我们接下来只需要构建基本运行库（runtime library），以及其上的应用逻辑就好了。所以，大多数看到某个DSL的人，只是它的实现者和使用者，而不是它的设计者。多数人只是埋头于使用，或者激情于评说，而忘了看看“一个具体DSL的背景”。  
  
例如，难道DOS批处理不是一个DSL吗？10行的JavaScript难道不就是一个完整的DSL framework吗？如果是，那么我们还有必要讨论“什么是DSL”，以及“怎样的DSL开发环境更好”的问题吗？我们是不是看看“我们在什么Domain”，以及“这个Domain如何描述、如何结构化和如何逻辑驱动之”，这些问题是不是才是更关键的？