Lisp的本质

jenny

February 25, 2014

Contents

Lisp的本质(The Nature of Lisp)
作者 Slava Akhmechet 译者 Alec Jang
出处: `http://www.defmacro.org/ramblings/lisp.html`

1 简介

最初在web的某些角落偶然看到有人赞美Lisp时, 我那时已经是一个颇有经验的程序员。在我的履历上, 掌握的语言范围相当广泛, 象C++, Java, C#主流语言等等都不在话下,我觉得我差不多知道所有的有关编程语言的事情。对待编程语言的问题上, 我觉得自己不太会遇到什么大问题。其实我大错特错了。

我试着学了一下Lisp, 结果马上就撞了墙。我被那些范例代码吓坏了。我想很多初次接触Lisp语言的人, 一定也有过类似的感受。Lisp的语法太次了。一个语言的发明人, 居然不肯用心弄出一套漂亮的语法, 那谁还会愿意学它。反正, 我是确确实实被那些难看的无数的括号搞蒙了。

回过神来之后, 我和Lisp社区的那伙人交谈, 诉说我的沮丧心情。结果, 立马就有一大套理论砸过来, 这套理论在Lisp社区处处可见, 几成惯例。比如说: Lisp的括号只是表面现象; Lisp的代码和数据的表达方式没有差别, 而且比XML语法高明许多, 所以有无穷的好处; Lisp有强大无比的元语言能力, 程序员可以写出自我维护的代码; Lisp可以创造出针对特定应用的语言子集; Lisp的运行时和编译时没有明确的分界; 等等, 等等, 等等。这么长的赞美词虽然看起来相当动人, 不过对我毫无意义。没人能给我演示这些东西是如何应用的, 因为这些东西一般来说只有在大型系统才会用到。我争辩说, 这些东西传统语言一样办得到。在和别人争论了数个小时之后, 我最终还是放弃了学Lisp的念头。为什么要花费几个月的时间学习语法这么难看的语言呢? 这种语言的概念这么晦涩, 又没什么好懂的例子。也许这语言不是该我这样的人学的。

几个月来, 我承受着这些Lisp辩护士对我心灵的重压。我一度陷入了困惑。我认识一些绝顶聪明的人, 我对他们相当尊敬, 我看到他们对Lisp的赞美达到了宗教般的高度。这就是说, Lisp中一定有某种神秘的东西存在, 我不能忍受自己对此的无知, 好奇心和求知欲最终不可遏制。我于是咬紧牙关埋头学习Lisp, 经过几个月的时间费劲心力的练习, 终于,我看到了那无穷无尽的泉水的源头。在经过脱胎换骨的磨练之后, 在经过七重地狱的煎熬之后, 终于, 我明白了。

顿悟在突然之间来临。曾经许多次, 我听到别人引用雷蒙德(译者注: 论文的作者, 著名的黑客社区理论家)的话: "Lisp语言值得学习。当你学会Lisp之后, 你会拥有深刻的体验。就算你平常并不用Lisp编程, 它也会使你成为更加优秀的程序员"。过去, 我根本不懂这些话的含义, 我也不相信这是真的。可是现在我懂得了。这些话蕴含的真理远远超过我过去的想像。我内心体会到一种神圣的情感, 一瞬间的顿悟, 几乎使我对电脑科学的观念发生了根本的改变。

顿悟的那一刻, 我成了Lisp的崇拜者。我体验到了宗教大师的感受: 一定要把我的知识传布开来, 至少要让10个迷失的灵魂得到拯救。按照通常的办法, 我把这些道理(就是刚开始别人砸过来的那一套, 不过现在我明白了真实的含义)告诉旁人。结果太令人失望了, 只有少数几个人在我坚持之下, 发生了一点兴趣, 但是仅仅看了几眼Lisp代码, 他们就退却了。照这样的办法, 也许费数年功夫能造就了几个Lisp迷, 但我觉得这样的结果太差强人意了, 我得想一套有更好的办法。

我深入地思考了这个问题。是不是Lisp有什么很艰深的东西, 令得那么多老练的程序员都不能领会? 不是, 没有任何绝对艰深的东西。因为我能弄懂, 我相信其他人也一定能。那么问题出在那里? 后来我终于找到了答案。我的结论就是, 凡是教人学高级概念, 一定要从他已经懂得的东西开始。如果学习过程很有趣, 学习的内容表达得很恰当, 新概念就会变得相当直观。这就是我的答案。所谓元编程, 所谓数据和代码形式合一, 所谓自修改代码, 所谓特定应用的子语言, 所有这些概念根本就是同族概念, 彼此互为解释, 肯定越讲越不明白。还是从实际的例子出发最有用。

我把我的想法说给Lisp程序员听, 遭到了他们的反对。"这些东西本身当然不可能用熟悉的知识来解释, 这些概念完全与众不同, 你不可能在别人已有的经验里找到类似的东西",可是我认为这些都是遁词。他们又反问我, "你自己为啥不试一下?" 好吧, 我来试一下。这篇文章就是我尝试的结果。我要用熟悉的直观的方法来解释Lisp, 我希望有勇气的人读完它, 拿杯饮料, 深呼吸一下, 准备被搞得晕头转向。来吧, 愿你获得大能。

2 重新审视XML

千里之行始于足下。让我们的第一步从XML开始。可是XML已经说得更多的了, 还能有什么新意思可说呢? 有的。XML自身虽然谈谈不上有趣, 但是XML和Lisp的关系却相当有趣。XML和Lisp的概念有着惊人的相似之处。XML是我们通向理解Lisp的桥梁。好吧, 我们且把XML当作活马医。让我们拿好手杖, 对XML的无人涉及的荒原地带作一番探险。我们要从一个全新的视角来考察这个题目。

表面上看, XML是一种标准化语法, 它以适合人阅读的格式来表达任意的层次化数据(hirearchical data)。象任务表(to-do list), 网页, 病历, 汽车保险单, 配置文件等等, 都是XML用武的地方。比如我们拿任务表做例子:

```
<todo name="housework">
    <item priority="high">Clean the house.</item>
    <item priority="medium">Wash the dishes.</item>
    <item priority="medium">Buy more soap.</item>
</todo>
```

解析这段数据时会发生什么情况? 解析之后的数据在内存中怎样表示? 显然, 用树来表示这种层次化数据是很恰当的。说到底, XML这种比较容易阅读的数据格式, 就是树型结构数据经过序列化之后的结果。任何可以用树来表示的数据, 同样可以用XML来表示, 反之亦然。希望你能懂得这一点, 这对下面的内容极其重要。

再进一步。还有什么类型的数据也常用树来表示? 无疑列表(list)也是一种。上过编译课吧? 还模模糊糊记得一点吧? 源代码在解析之后也是用树结构来存放的, 任何编译程序都会把源代码解析成一棵抽象语法树, 这样的表示法很恰当, 因为源代码就是层次结构的:函数包含参数和代码块, 代码快包含表达式和语句, 语句包含变量和运算符等等。

我们已经知道, 任何树结构都可以轻而易举的写成XML, 而任何代码都会解析成树, 因此,任何代码都可以转换成XML, 对不对? 我举个例子, 请看下面的函数:

```
int add(int arg1, int arg2)
{
    return arg1+arg2;
}
```

能把这个函数变成对等的XML格式吗? 当然可以。我们可以用很多种方式做到, 下面是其中的一种, 十分简单:

```
<define-function return-type="int" name="add">
    <arguments>
        <argument type="int">arg1</argument>
        <argument type="int">arg2</argument>
    </arguments>
    <body>
        <return>
            <add value1="arg1" value2="arg2" />
        </return>
    </body>
</define>
```

这个例子非常简单, 用哪种语言来做都不会有太大问题。我们可以把任何程序码转成XML,也可以把XML转回到原来的程序码。我们可以写一个转换器, 把Java代码转成XML, 另一个转换器把XML转回到Java。一样的道理, 这种手段也可以用来对付C++(这样做跟发疯差不多么。可是的确有人在做, 看看GCC-XML(http://www.gccxml.org)就知道了)。进一步说,凡是有相同语言特性而语法不同的语言, 都可以把XML当作中介来互相转换代码。实际上几乎所有的主流语言都在一定程度上满足这个条件。我们可以把XML作为一种中间表示法,在两种语言之间互相译码。比方说, 我们可以用Java2XML把Java代码转换成XML, 然后用XML2CPP再把XML转换成C++代码, 运气好的话, 就是说, 如果我们小心避免使用那些C++不具备的Java特性的话, 我们可以得到完好的C++程序。这办法怎么样, 漂亮吧?

这一切充分说明, 我们可以把XML作为源代码的通用存储方式, 其实我们能够产生一整套使用统一语法的程序语言, 也能写出转换器, 把已有代码转换成XML格式。如果真的采纳这种办法, 各种语言的编译器就用不着自己写语法解析了, 它们可以直接用XML的语法解析来直接生成抽象语法树。

说到这里你该问了, 我们研究了这半天XML, 这和Lisp有什么关系呢? 毕竟XML出来之时,Lisp早已经问世三十年了。这里我可以保证, 你马上就会明白。不过在继续解释之前, 我们先做一个小小的思维练习。看一下上面这个XML版本的add函数例子, 你怎样给它分类,是代码还是数据? 不用太多考虑都能明白, 把它分到哪一类都讲得通。它是XML, 它是标准格式的数据。我们也知道, 它可以通过内存中的树结构来生成(GCC-XML做的就是这个事情)。它保存在不可执行的文件中。我们可以把它解析成树节点, 然后做任意的转换。显而易见, 它是数据。不过且慢, 虽然它语法有点陌生, 可它又确确实实是一个add函数, 对吧? 一旦经过解析, 它就可以拿给编译器编译执行。我们可以轻而易举写出这个XML 代码解释器, 并且直接运行它。或者我们也可以把它译成Java或C++代码, 然后再编译运行。所以说, 它也是代码。

我们说到那里了? 不错, 我们已经发现了一个有趣的关键之点。过去被认为很难解的概念已经非常直观非常简单的显现出来。代码也是数据, 并且从来都是如此。这听起来疯疯癫癫的, 实际上却是必然之事。我许诺过会以一种全新的方式来解释Lisp, 我要重申我的许诺。但是我们此刻还没有到预定的地方, 所以还是先继续上边的讨论。

刚才我说过, 我们可以非常简单地实现XML版的add函数解释器, 这听起来好像不过是说说而已。谁真的会动手做一下呢? 未必有多少人会认真对待这件事。随便说说, 并不打算真的去做, 这样的事情你在生活中恐怕也遇到吧。你明白我这样说的意思吧, 我说的有没有打动你? 有哇, 那好, 我们继续。

重新审视Ant

我们现在已经来到了月亮背光的那一面, 先别忙着离开。再探索一下, 看看我们还能发现什么东西。闭上眼睛, 想一想2000年冬天的那个雨夜, 一个名叫James Duncan Davidson的杰出的程序员正在研究Tomcat的servlet容器。那时, 他正小心地保存好刚修改过的文件, 然后执

行make。结果冒出了一大堆错误, 显然有什么东西搞错了。经过仔细检查, 他想, 难道是因为tab前面加了个空格而导致命令不能执行吗? 确实如此。老是这样, 他真的受够了。乌云背后的月亮给了他启示, 他创建了一个新的Java项目, 然后写了一个简单但是十分有用的工具, 这个工具巧妙地利用了Java属性文件中的信息来构造工程, 现在James可以写makefile的替代品, 它能起到相同的作用, 而形式更加优美, 也不用担心有makefile那样可恨的空格问题。这个工具能够自动解释属性文件, 然后采取正确的动作来编译工程。真是简单而优美。

(作者注: 我不认识James, James也不认识我, 这个故事是根据网上关于Ant历史的帖子虚构的)

使用Ant构造Tomcat之后几个月, 他越来越感到Java的属性文件不足以表达复杂的构造指令。文件需要检出, 拷贝, 编译, 发到另外一台机器, 进行单元测试。要是出错, 就发邮件给相关人员, 要是成功, 就继续在尽可能高层的卷(volumn)上执行构造。追踪到最后,卷要回复到最初的水平上。确实, Java的属性文件不够用了, James需要更有弹性的解决方案。他不想自己写解析器(因为他更希望有一个具有工业标准的方案)。XML看起来是个不错的选择。他花了几天工夫把Ant移植到XML，于是，一件伟大的工具诞生了。

Ant是怎样工作的? 原理非常简单。Ant把包含有构造命令的XML文件(算代码还是算数据,你自己想吧)，交给一个Java程序来解析每一个元素，实际情况比我说的还要简单得多。一个简单的XML指令会导致具有相同名字的Java类装入，并执行其代码。

<copy todir="../new/dir"> <fileset dir="src_{dir}" /> </copy>

这段文字的含义是把源目录复制到目标目录，Ant会找到一个"copy"任务(实际上就是一个Java类), 通过调用Java的方法来设置适当参数(todir和fileset)，然后执行这个任务。Ant带有一组核心类, 可以由用户任意扩展, 只要遵守若干约定就可以。Ant找到这些类,每当遇到XML元素有同样的名字, 就执行相应的代码。过程非常简单。Ant做到了我们前面所说的东西: 它是一个语言解释器, 以XML作为语法, 把XML元素转译为适当的Java指令。我们可以写一个"add"任务, 然后, 当发现XML中有add描述的时候, 就执行这个add任务。由于Ant是非常流行的项目, 前面展示的策略就显得更为明智。毕竟, 这个工具每天差不多有几千家公司在使用。

到目前为之, 我还没有说Ant在解析XML时所遇到困难。你也不用麻烦去它的网站上去找答案了, 不会找到有价值的东西。至少对我们这个论题来说是如此。我们还是继续下一步讨论吧。我们答案就在那里。

3 为什么是XML

有时候正确的决策并非完全出于深思熟虑。我不知道James选择XML是否出于深思熟虑。也许仅仅是个下意识的决定。至少从James在Ant网站上发表的文章看起来, 他所说的理由完全是似是而非。他的主要理由是移植性和扩展性, 在Ant案例上, 我看不出这两条有什么帮助。使用XML而不是Java代码, 到底有什么好处? 为什么不写一组Java类, 提供api来满足基本任务(拷贝目录, 编译等等), 然后在Java里直接调用这些代码? 这样做仍然可以保证移植性, 扩展性也是毫无疑问的。而且语法也更为熟悉, 看着顺眼。那为什么要用 XML呢? 有什么更好的理由吗?

有的。虽然我不确定James是否确实意识到了。在语义的可构造性方面, XML的弹性是Java望尘莫及的。我不想用高深莫测的名词来吓唬你, 其中的道理相当简单, 解释起来并不费很多功夫。好, 做好预备动作, 我们马上就要朝向顿悟的时刻做奋力一跃。

上面的那个copy的例子, 用Java代码怎样实现呢? 我们可以这样做:

```
CopyTask copy = new CopyTask();
Fileset fileset = new Fileset();

fileset.setDir("src_dir");
copy.setToDir("../new/dir");
copy.setFileset(fileset);

copy.excute();
```

这个代码看起来和XML的那个很相似, 只是稍微长一点。差别在那里? 差别在于XML构造了一个特殊的copy动词, 如果我们硬要用Java来写的话, 应该是这个样子:

```
    copy("../new/dir");
    {
        fileset("src_dir");
    }
```

看到差别了吗? 以上代码(如果可以在Java中用的化), 是一个特殊的copy算符, 有点像for循环或者Java5中的foreach循环。如果我们有一个转换器, 可以把XML转换到Java, 大概就会得到上面这段事实上不可以执行的代码。因为Java的技术规范是定死的, 我们没有办法在程序里改变它。我们可以增加包, 增加类, 增加方法, 但是我们没办法增加算符, 而对于XML, 我们显然可以任由自己增加这样的东西。对于XML的语法树来说, 只要原意,我们可以任意增加任何元素, 因此等于我们可以任意增加算符。如果你还不太明白的话, 看下面这个例子, 加入我们要给Java引入一个unless算符:

```
    unless(someObject.canFly())
    {
        someObject.transportByGround():
    }
```

在上面的两个例子中, 我们打算给Java语法扩展两个算符, 成组拷贝文件算符和条件算符unless, 我们要想做到这一点, 就必须修改Java编译器能够接受的抽象语法树, 显然我们无法用Java标准的功能来实现它。但是在XML中我们可以轻而易举地做到。我们的解析器根据 XML元素, 生成抽象语法树, 由此生成算符, 所以, 我们可以任意引入任何算符。

对于复杂的算符来说, 这样做的好处显而易见。比如, 用特定的算符来做检出源码, 编译文件, 单元测试, 发送邮件等任务, 想想看有多么美妙。对于特定的题目, 比如说构造软件项目, 这些算符的使用可以大幅减低少代码的数量。增加代码的清晰程度和可重用性。解释性的XML可以很容易的达到这个目标。XML是存储层次化数据的简单数据文件, 而在Java中, 由于层次结构是定死的(你很快就会看到, Lisp的情况与此截然不同), 我们就没法达到上述目标。也许这正是Ant的成功之处呢。

你可以注意一下最近Java和C#的变化(尤其是C#3.0的技术规范), C#把常用的功能抽象出来, 作为算符增加到C#中。C#新增加的query算符就是一个例子。它用的还是传统的作法:C#的设计者修改抽象语法树, 然后增加对应的实现。如果程序员自己也能修改抽象语法树该有多好! 那样我们就可以构造用于特定问题的子语言(比如说就像Ant这种用于构造项目的语言), 你能想到别的例子吗? 再思考一下这个概念。不过也不必思考太甚, 我们待会还会回到这个题目。那时候就会更加清晰。

4 离Lisp越来越近

我们先把算符的事情放一放, 考虑一下Ant设计局限之外的东西。我早先说过, Ant可以通过写Java类来扩展。Ant解析器会根据名字来匹配XML元素和Java类, 一旦找到匹配, 就执行相应任务。为什么不用Ant自己来扩展Ant呢? 毕竟核心任务要包含很多传统语言的结构(例如"if"), 如果Ant自身就能提供构造任务的能力(而不是依赖java类), 我们就可以得到更高的移植性。我们将会依赖一组核心任务(如果你原意, 也不妨把它称作标准库), 而不用管有没有Java 环境了。这组核心任务可以用任何方式来实现, 而其他任务建筑在这组核心任务之上, 那样的话, Ant就会成为通用的, 可扩展的, 基于XML的编程语言。考虑下面这种代码的可能性:

```
<task name="Test">
      <echo message="Hello␣World" />
  </task>
  <Test />
```

如果XML支持"task"的创建, 上面这段代码就会输出"Hello World!". 实际上, 我们可以用Java写个"task"任务, 然后用Ant-XML来扩展它。Ant可以在简单原语的基础上写出更复杂的原语, 就像其他编程语言常用的作法一样。这也就是我们一开始提到的基于XML的编程语言。这样做用处不大(你知道为甚么吗?), 但是真的很酷。

再看一回我们刚才说的Task任务。祝贺你呀, 你在看Lisp代码!!! 我说什么? 一点都不像Lisp吗? 没关系, 我们再给它收拾一下。

5 比XML更好

前面一节说过, Ant自我扩展没什么大用, 原因在于XML很烦琐。对于数据来说, 这个问题还不太大, 但如果代码很烦琐的话, 光是打字上的麻烦就足以抵消它的好处。你写过Ant 的脚本吗? 我写过, 当脚本达到一定复杂度的时候, XML非常让人厌烦。想想看吧, 为了写结束标签, 每个词都得打两遍, 不发疯算好的!

为了解决这个问题, 我们应当简化写法。须知, XML仅仅是一种表达层次化数据的方式。我们并不是一定要使用尖括号才能得到树的序列化结果。我们完全可以采用其他的格式。其中的一种(刚好就是Lisp所采用的)格式, 叫做s表达式。s表达式要做的和XML一样, 但它的好处是写法更简单, 简单的写法更适合代码输入。后面我会详细讲s表达式。这之前我要清理一下XML的东西。考虑一下关于拷贝文件的例子:

```
<copy toDir="../new/dir">
    <fileset dir="src_dir">
</copy>
```

想想看在内存里面, 这段代码的解析树在内存会是什么样子? 会有一个"copy"节点, 其下有一个 "fileset"节点, 但是属性在哪里呢? 它怎样表达呢? 如果你以前用过XML, 并且弄不清楚该用元素还是该用属性, 你不用感到孤单, 别人一样糊涂着呢。没人真的搞得清楚。这个选择与其说是基于技术的理由, 还不如说是闭着眼瞎摸。从概念上来讲, 属性也是一种元素, 任何属性能做的, 元素一样做得到。XML引入属性的理由, 其实就是为了让XML写法不那么冗长。比如我们看个例子:

```
<copy>
    <toDir>../new/dir</toDir>
    <fileset>
        <dir>src_dir</dir>
    </fileset>
</copy>
```

两下比较, 内容的信息量完全一样, 用属性可以减少打字数量。如果XML没有属性的话, 光是打字就够把人搞疯掉。

说完了属性的问题, 我们再来看一看s表达式。之所以绕这么个弯, 是因为s表达式没有属性的概念。因为s表达式非常简练, 根本没有必要引入属性。我们在把XML转换成s表达式的时候, 心里应该记住这一点。看个例子, 上面的代码译成s表达式是这样的:

```
(copy
    (todir "../new/dir")
    (fileset (dir "src_dir")))
```

仔细看看这个例子, 差别在哪里? 尖括号改成了圆括号, 每个元素原来是有一对括号标记包围的, 现在取消了后一个(就是带斜杠的那个)括号标记。表示元素的结束只需要一个")"就可以了。不错, 差别就是这些。这两种表达方式的转换, 非常自然, 也非常简单。s表达式打起字来, 也省事得多。第一次看s表达式(Lisp)时, 括号很烦人是吧? 现在我们明白了背后的道理, 一下子就变得容易多了。至少, 比XML要好的多。用s表达式写代码, 不单是实用, 而且也很让人愉快。s表达式具有XML的一切好处, 这些好处是我们刚刚探讨过的。现在我们看看更加Lisp风格的task例子:

```
(task (name "Test")
    (echo (message "Hellow␣World!")))
(Test)
```

用Lisp的行话来讲, s表达式称为表(list)。对于上面的例子, 如果我们写的时候不加换行, 用逗号来代替空格, 那么这个表达式看起来就非常像一个元素列表, 其中又嵌套着其他标记。

```
(task, (name, "test"), (echo, (message, "Hello␣World!")))
```

XML自然也可以用这样的风格来写。当然上面这句并不是一般意义上的元素表。它实际上是一个树。这和XML的作用是一样的。称它为列表, 希望你不会感到迷惑, 因为嵌套表和树实际上是一码事。Lisp的字面意思就是表处理(list processing), 其实也可以称为树处理, 这和处理XML节点没有什么不同。

经受这一番折磨以后, 现在我们终于相当接近Lisp了, Lisp的括号的神秘本质(就像许多Lisp狂热分子认为的)逐渐显现出来。现在我们继续研究其他内容。

6 重新审视C语言的宏

到了这里, 对XML的讨论你大概都听累了, 我都讲累了。我们先停一停, 把树, s表达式,Ant这些东西先放一放, 我们来说说C的预处理器。一定有人问了, 我们的话题和C有什么关系? 我们已经知道了很多关于元编程的事情, 也探讨过专门写代码的代码。理解这问题有一定难度, 因为相关讨论文章所使用的编程语言, 都是你们不熟悉的。但是如果只论概念的话, 就相对要简单一些。我相信, 如果以C语言做例子来讨论元编程, 理解起来一定会容易得多。好, 我们接着看。

一个问题是, 为什么要用代码来写代码呢? 在实际的编程中, 怎样做到这一点呢? 到底元编程是什么意思? 你大概已经听说过这些问题的答案, 但是并不懂得其中缘由。为了揭示背后的真理, 我们来看一下一个简单的数据库查询问题。这种题目我们都做过。比方说, 直接在程序码里到处写SQL语句来修改表(table)里的数据, 写多了就非常烦人。即便用C#3.0的LINQ, 仍然不减其痛苦。写一个完整的SQL查询(尽管语法很优美)来修改某人的地址, 或者查找某人的名字, 绝对是件令程序员倍感乏味的事情, 那么我们该怎样来解决这个问题? 答案就是: 使用数据访问层。

概念挺简单, 其要点是把数据访问的内容(至少是那些比较琐碎的部分)抽象出来, 用类来映射数据库的表, 然后用访问对象属性访问器(accessor)的办法来间接实现查询。这样就极大地简化了开发工作量。我们用访问对象的方法(或者属性赋值, 这要视你选用的语言而定)来代替写SQL查询语句。凡是用过这种方法的人, 都知道这很节省时间。当然, 如果你要亲自写这样一个抽象层, 那可是要花非常多的时间的–你要写一组类来映射表, 把属性访问转换为SQL查询, 这个活相当耗费精力。用手工来做显然是很不明智的。但是一旦你有了方案和模板, 实际上就没有多少东西需要思考的。你只需要按照同样的模板一次又一次重复编写相似代码就可以了。事实上很多人已经发现了更好的方法, 有一些工具可以帮助你连接数据库, 抓取数据库结构定义(schema), 按照预定义的或者用户定制的模板来自动编写代码。

如果你用过这种工具, 你肯定会对它的神奇效果深为折服。往往只需要鼠标点击数次, 就可以连接到数据库, 产生数据访问源码, 然后把文件加入到你的工程里面, 十几分钟的工作, 按照往常手工方式来作的话, 也许需要数百个小时人工(man-hours)才能完成。可是,如果你的数据库结构定义后来改变了怎么办? 那样的话, 你只需把这个过程重复一遍就可以了。甚至有一些工具能自动完成这项变动工作。你只要把它作为工程构造的一部分, 每次编译工程的时候, 数据库部分也会自动地重新构造。这真的太棒了。你要做的事情基本上减到了0。如果数据库结构定义发生了改变, 并在编译时自动更新了数据访问层的代码,那么程序中任何使用过时的旧代码的地方, 都会引发编译错误。

数据访问层是个很好的例子, 这样的例子还有好多。从GUI样板代码, WEB代码, COM和CORBA存根, 以及MFC和ATL等等。在这些地方, 都是有好多相似代码多次重复。既然这些代码有可能自动编写, 而程序员时间又远远比CPU时间昂贵, 当然就产生了好多工具来自动生成样板代码。这些工具的本质是什么呢? 它们实际上就是制造程序的程序。它们有一个神秘的名字, 叫做元编程。所谓元编程的本义, 就是如此。

元编程本来可以用到无数多的地方, 但实际上使用的次数却没有那么多。归根结底, 我们心里还是在盘算, 假设重复代码用拷贝粘贴的话, 大概要重复6,7次, 对于这样的工作量,值得专门建立一套生成工具吗? 当然不值得。数据访问层和COM存根往往需要重用数百次,甚至上千次, 所以用工具生成是最好的办法。而那些仅仅是重复几次十几次的代码, 是没有必要专门做工具的。不必要的时候也去开发代码生成工具, 那就显然过度估计了代码生成的好处。当然, 如果创建这类工具足够简单的话, 还是应当尽量多用, 因为这样做必然会节省时间。现在来看一下有没有合理的办法来达到这个目的。

现在, C预处理器要派上用场了。我们都用过C/C++的预处理器, 我们用它执行简单的编译指令, 来产生简单的代码变换(比方说, 设置调试代码开关), 看一个例子:

```
#define triple(X) X+X+X
```

这一行的作用是什么? 这是一个简单的预编译指令, 它把程序中的triple(X)替换称为X+X+X。例如, 把所有的triple(5)都换成5+5+5, 然后再交给编译器编译。这就是一个简单的代码生成的例子。要是C的预处理器再强大一点, 要是能够允许连接数据库, 要是能多一些其他简单的机制, 我们就可以在我们程序的内部开发自己的数据访问层。下面这个例子, 是一个假想的对C宏的扩展:

```
#get -db-schema("127.0.0.1")
#iterate -through-tables
#for -each-table
    class #table -name
        {
        };
#end -for-each
```

我们连接数据库结构定义, 遍历数据表, 然后对每个表创建一个类, 只消几行代码就完成了这个工作。这样每次编译工程的时候, 这些类都会根据数据库的定义同步更新。显而易见, 我们不费吹灰之力就在程序内部建立了一个完整的数据访问层, 根本用不着任何外部工具。当然这种作法有一个缺点, 那就是我们得学习一套新的"编译时语言", 另一个缺点就是根本不存在这么一个高级版的C预处理器。需要做复杂代码生成的时候, 这个语言(译者注: 这里指预处理指令, 即作者所说的"编译时语言")本身也一定会变得相当复杂。它必须支持足够多的库和语言结构。比如说我们想要生成的代码要依赖某些ftp服务器上的文件, 预处理器就得支持ftp访问, 仅仅因为这个任务而不得不创造和学习一门新的语言,真是有点让人恶心(事实上已经存在着有此能力的语言, 这样做就更显荒谬)。我们不妨再灵活一点, 为什么不直接用 C/C++自己作为自己的预处理语言呢? 这样子的话, 我们可以发挥语言的强大能力, 要学的新东西也只不过是几个简单的指示字 , 这些指示字用来区别编译时代码和运行时代码。

```
<%
    cout<<"Enter␣a␣number:␣";
    cin>>n;
%>
for(int i=0;i< <% n %>;i++)
{
    cout<<"hello"<<endl;
}
```

你明白了吗? 在<%和%>标记之间的代码是在编译时运行的, 标记之外的其他代码都是普通代码。编译程序时, 系统会提示你输入一个数, 这个数在后面的循环中会用到。而for循环的代码会被编译。假定你在编译时输入5, for循环的代码将会是:

```
for(int i=0;i<5; i++)
{
```

```
    cout<<"hello"<<endl;
}
```

又简单又有效率, 也不需要另外的预处理语言。我们可以在编译时就充分发挥宿主语言(此处是C/C++)的强大能力, 我们可以很容易地在编译时连接数据库, 建立数据访问层, 就像JSP或者ASP创建网页那样。我们也用不着专门的窗口工具来另外建立工程。我们可以在代码中立即加入必要的工具。我们也用不着顾虑建立这种工具是不是值得, 因为这太容易了, 太简单了。这样子不知可以节省多少时间啊。

7 你好, Lisp

到此刻为止, 我们所知的关于Lisp的指示可以总结为一句话: Lisp是一个可执行的语法更优美的XML, 但我们还没有说Lisp是怎样做到这一点的, 现在开始补上这个话题。

Lisp有丰富的内置数据类型, 其中的整数和字符串和其他语言没什么分别。像71或者"hello"这样的值, 含义也和C++或者Java这样的语言大体相同。真正有意思的三种类型是符号(symbol), 表和函数。这一章的剩余部分, 我都会用来介绍这几种类型, 还要介绍Lisp环境是怎样编译和运行源码的。这个过程用Lisp的术语来说通常叫做求值。通读这一节内容, 对于透彻理解元编程的真正潜力, 以及代码和数据的同一性, 和面向领域语言的观念, 都极其重要。万勿等闲视之。我会尽量讲得生动有趣一些, 也希望你能获得一些启发。那好, 我们先讲符号。

大体上, 符号相当于C++或Java语言中的标志符, 它的名字可以用来访问变量值(例如currentTime, arrayCount, n, 等等), 差别在于, Lisp中的符号更加基本。在C++或Java里面, 变量名只能用字母和下划线的组合, 而Lisp的符号则非常有包容性, 比如, 加号(+)就是一个合法的符号, 其他的像-, =, hello-world, *等等都可以是符号名。符号名的命名规则可以在网上查到。你可以给这些符号任意赋值, 我们这里先用伪码来说明这一点。假定函数set是给变量赋值(就像等号=在C++和Java里的作用), 下面是我们的例子:

```
set(test, 5)              // 符号的值为test5
set(=, 5)                 // 符号的值为=5
set(test, "hello")        // 符号的值为字符串test"hello"
set(test, =)              // 此时符号的值为=5, 所以的也为test5
set(* , "hello")           // 符号 * 的值为"hello"
```

好像有什么不对的地方? 假定我们对*赋给整数或者字符串值, 那做乘法时怎么办? 不管怎么说, * 总是乘法呀? 答案简单极了。Lisp中函数的角色十分特殊, 函数也是一种数据类型, 就像整数和字符串一样, 因此可以把它赋值给符号。乘法函数Lisp的内置函数, 默认赋给 * , 你可以把其他函数赋值给*, 那样*就不代表乘法了。你也可以把这函数的值存到另外的变量里。我们再用伪码来说明一下:

```
* (3,4)           // 乘34, 结果是12
set(temp, * )     // 把 * 的值, 也就是乘法函数, 赋值给temp
set( * , 3)        // 把赋予3 *
* (3,4)           // 错误的表达式, * 不再是乘法, 而是数值3
temp(3,4)        // 是乘法函数temp, 所以此表达式的值为乘等于3412
set(* , temp)     // 再次把乘法函数赋予 *
* (3,4)           // 乘等于3412
```

再古怪一点, 把减号的值赋给加号:

```
set(+, - )        // 减号( - 是内置的减法函数)
+(5, 4)          // 加号现在是代表减法函数(+), 结果是减等于541
```

这只是举例子, 我还没有详细讲函数。Lisp中的函数是一种数据类型, 和整数, 字符串,符号等等一样。一个函数并不必然有一个名字, 这和C++或者Java语言的情形很不相同。在这里函数自己代表自己。事实上它是一个指向代码块的指针, 附带有一些其他信息(例如一组参数变量)。只有在把函数赋予其他符号时, 它才具有了名字, 就像把一个数值或字符串赋予变量一样的道理。你可以用一个内置的专门用于创建函数的函数来创建函数,然后把它赋值给符号fn, 用伪码来表示就是:

```
fn [a]
{
    return * (a, 2);
}
```

这段代码返回一个具有一个参数的函数, 函数的功能是计算参数乘2的结果。这个函数还没有名字, 你可以把此函数赋值给别的符号:

```
set(times-two, fn [a] {return * (a, 2)})
```

我们现在可以这样调用这个函数:

```
time-two(5)          // 返回10
```

我们先跳过符号和函数, 讲一讲表。什么是表? 你也许已经听过好多相关的说法。表, 一言以蔽之, 就是把类似XML那样的数据块, 用s表达式来表示。表用一对括号括住, 表中元素以空格分隔, 表可以嵌套。例如(这回我们用真正的Lisp语法, 注意用分号表示注释):

```
()                        ; 空表
(1)                       ; 含一个元素的表
(1 "test")                ; 两元素表, 一个元素是整数1, 另一个是字符串
(test "hello")            ; 两元素表, 一个元素是符号, 另一个是字符串
(test (1 2) "hello")      ; 三元素表, 一个符号test, 一个含有两个元素和的12
                          ; 表, 最后一个元素是字符串
```

当Lisp系统遇到这样的表时, 它所做的, 和Ant处理XML数据所做的, 非常相似, 那就是试图执行它们。其实, Lisp源码就是特定的一种表, 好比Ant源码是一种特定的XML一样。Lisp执行表的顺序是这样的, 表的第一个元素当作函数, 其他元素当作函数的参数。如果其中某个参数也是表, 那就按照同样的原则对这个表求值, 结果再传递给最初的函数作为参数。这就是基本原则。我们看一下真正的代码:

```
( * 3 4)                    ; 相当于前面列举过的伪码 * (3,4), 即计算乘34
(times-two 5)              ; 返回10, 按照前面的定义是求参数的倍times-two2
(3 4)                      ; 错误, 不是函数3
(time-two)                 ; 错误, 要求一个参数times-two
(times-two 3 4)            ; 错误, 只要求一个参数times-two
(set + -)                  ; 把减法函数赋予符号+
(+ 5 4)                    ; 依据上一句的结果, 此时表示减法+, 所以返回1
( * 3 (+ 2 2))             ; 的结果是2+24, 再乘3, 结果是12
```

上述的例子中, 所有的表都是当作代码来处理的。怎样把表当作数据来处理呢? 同样的,设想一下, Ant是把XML数据当作自己的参数。在Lisp中, 我们给表加一个前缀'来表示数据。

```
(set test '(1 2))          ; 的值为两元素表test
(set test (1 2))           ; 错误, 不是函数1
(set test '( * 3 4))       ; 的值是三元素表test, 三个元素分别是 *, 3, 4
```

我们可以用一个内置的函数head来返回表的第一个元素, tail函数来返回剩余元素组成的表。

```
(head '( * 3 4))           ; 返回符号 *
(tail '( * 3 4))           ; 返回表(3 4)
(head (tal '( * 3 4)))     ; 返回3
(head test)                ; 返回 *
```

你可以把Lisp的内置函数想像成Ant的任务。差别在于, 我们不用在另外的语言中扩展Lisp(虽然完全可以做得到), 我们可以用Lisp自己来扩展自己, 就像上面举的times-two函数的例子。Lisp的内置函数集十分精简, 只包含了十分必要的部分。剩下的函数都是作为标准库来实现的。

8 Lisp宏

我们已经看到, 元编程在一个类似jsp的模板引擎方面的应用。我们通过简单的字符串处理来生成代码。但是我们可以做的更好。我们先提一个问题, 怎样写一个工具, 通过查找目录结构中的源文件来自动生成Ant脚本。

用字符串处理的方式生成Ant脚本是一种简单的方式。当然, 还有一种更加抽象, 表达能力更强, 扩展性更好的方式, 就是利用XML库在内存中直接生成XML节点, 这样的话内存中的节点就可以自动序列化成为字符串。不仅如此, 我们的工具还可以分析这些节点, 对已有的XML文件做变换。通过直接处理XML节点。我们可以超越字符串处理, 使用更高层次的概念, 因此我们的工作就会做的更快更好。

我们当然可以直接用Ant自身来处理XML变换和制作代码生成工具。或者我们也可以用Lisp来做这项工作。正像我们以前所知的, 表是Lisp内置的数据结构, Lisp含有大量的工具来快速有效的操作表(head和tail是最简单的两个)。而且, Lisp没有语义约束, 你可以构造任何数据结构, 只要你原意。

Lisp通过宏(macro)来做元编程。我们写一组宏来把任务列表(to-do list)转换为专用领域语言。

回想一下上面to-do list的例子, 其XML的数据格式是这样的:

```
<todo name = "housework">
    <item priority = "high">Clean the hose</item>
    <item priority = "medium">Wash the dishes</item>
    <item priority = "medium">Buy more soap</item>
</todo>
```

相应的s表达式是这样的:

```
(todo "housework"
    (item (priority high) "Clean␣the␣house")
    (item (priority medium) "Wash␣the␣dishes")
    (item (priority medium) "Buy␣more␣soap"))
```

假设我们要写一个任务表的管理程序, 把任务表数据存到一组文件里, 当程序启动时, 从文件读取这些数据并显示给用户。在别的语言里(比如说Java), 这个任务该怎么做? 我们会解析XML文件, 从中得出任务表数据, 然后写代码遍历XML树, 再转换为Java的数据结构(老实讲, 在Java里解析XML真不是件轻松的事情), 最后再把数据展示给用户。现在如果用Lisp, 该怎么做?

假定要用同样思路的化, 我们大概会用Lisp库来解析XML。XML对我们来说就是一个Lisp的表(s表达式), 我们可以遍历这个表, 然后把相关数据提交给用户。可是, 既然我们用Lisp, 就根本没有必要再用XML格式保存数据, 直接用s表达式就好了, 这样就没有必要做转换了。我们也用不着专门的解析库, Lisp可以直接在内存里处理s表达式。注意, Lisp编译器和.net编译器一样, 对Lisp程序来说, 在运行时总是随时可用的。

但是还有更好的办法。我们甚至不用写表达式来存储数据, 我们可以写宏, 把数据当作代码来处理。那该怎么做呢? 真的简单。回想一下, Lisp的函数调用格式:

```
(function-name arg1 arg2 arg3)
```

其中每个参数都是s表达式, 求值以后, 传递给函数。如果我们用(+ 4 5)来代替arg1, 那么, 程序会先求出结果, 就是9, 然后把9传递给函数。宏的工作方式和函数类似。主要的差别是, 宏的参数在代入时不求值。

```
(macro-name (+ 4 5))
```

这里, (+ 4 5)作为一个表传递给宏, 然后宏就可以任意处理这个表, 当然也可以对它求值。宏的返回值是一个表, 然后有程序作为代码来执行。宏所占的位置, 就被替换为这个结果代码。我们可以定义一个宏把数据替换为任意代码, 比方说, 替换为显示数据给用户的代码。

这和元编程, 以及我们要做的任务表程序有什么关系呢? 实际上, 编译器会替我们工作, 调用相应的宏。我们所要做的, 仅仅是创建一个把数据转换为适当代码的宏。

例如, 上面曾经将过的C的求三次方的宏, 用Lisp来写是这样子:

```
(defmacro triple (x)
      (+  x  x  x))
```

(译注: 在Common Lisp中, 此处的单引号应当是反单引号, 意思是对表不求值, 但可以对表中某元素求值, 记号~表示对元素x求值, 这个求值记号在Common Lisp中应当是逗号。反单引号和单引号的区别是, 单引号标识的表, 其中的元素都不求值。这里作者所用的记号是自己发明的一种Lisp方言Blaise, 和common lisp略有不同, 事实上, 发明方言是lisp高手独有的乐趣, 很多狂热分子都热衷这样做。比如Paul Graham就发明了ARC, 许多记号比传统的Lisp简洁得多, 显得比较现代)

单引号的用处是禁止对表求值。每次程序中出现triple的时候,

```
(triple 4)
```

都会被替换成:

```
(+ 4 4 4)
```

我们可以为任务表程序写一个宏, 把任务数据转换为可执行码, 然后执行。假定我们的输出是在控制台:

```
(defmacro item (priority note)
    (block(print stdout tab "Prority:  "  (head (tail priority)) endl)(print stdout tab "Note:  "  note endl endl)))
```

我们创造了一个非常小的有限的语言来管理嵌在Lisp中的任务表。这个语言只用来解决特定领域的问题, 通常称之为DSLs(特定领域语言, 或专用领域语言)。

9 特定领域语言

本文谈到了两个特定领域语言, 一个是Ant, 处理软件构造。一个是没起名字的, 用于处理任务表。两者的差别在于, Ant是用XML, XML解析器, 以及Java语言合在一起构造出来的。而我们的迷你语言则完全内嵌在Lisp中, 只消几分钟就做出来了。

我们已经说过了DSL的好处, 这也就是Ant用XML而不直接用Java的原因。如果使用Lisp, 我们可以任意创建DSL, 只要我们需要。我们可以创建用于网站程序的DSL, 可以写多用户游戏, 做固定收益贸易(fixed income trade), 解决蛋白质折叠问题, 处理事务问题, 等等。我们可以把这些叠放在一起, 造出一个语言, 专门解决基于网络的贸易程序, 既有网络语言的优势, 又有贸易语言的好处。每天我们都会收获这种方法带给我们的益处, 远远超过Ant所能给予我们的。

用DSL解决问题, 做出的程序精简, 易于维护, 富有弹性。在Java里面, 我们可以用类来处理问题。这两种方法的差别在于, Lisp使我们达到了一个更高层次的抽象, 我们不再受语言解析器本身的限制, 比较一下用Java库直接写的构造脚本和用Ant写的构造脚本其间的差别。同样的, 比较一下你以前所做的工作, 你就会明白Lisp带来的好处。

10 接下来

学习Lisp就像战争中争夺山头。尽管在电脑科学领域, Lisp已经算是一门古老的语言, 直到现在仍然很少有人真的明白该怎样给初学者讲授Lisp。尽管Lisp老手们尽了很大努力,今天新手学习Lisp仍然是困难重重。好在现在事情正在发生变化, Lisp的资源正在迅速增加, 随着时间推移, Lisp将会越来越受关注。

Lisp使人超越平庸, 走到前沿。学会Lisp意味着你能找到更好的工作, 因为聪明的雇主会被你与众不同的洞察力所打动。学会Lisp也可能意味着明天你可能会被解雇, 因为你总是强调, 如果公司所有软件都用Lisp写, 公司将会如何卓越, 而这些话你的同事会听烦的。Lisp值得努力学习吗? 那些已经学会Lisp的人都说值得, 当然, 这取决于你的判断。

你的看法呢?

这篇文章写写停停, 用了几个月才最终完成。如果你觉得有趣, 或者有什么问题, 意见或建议, 请给我发邮件coffeemug@gmail.com,我会很高兴收到你的反馈。