**宏和元编程(Macros and Metaprogramming)**

**什么是元编程?(What Is Metaprogramming? )**

元编程是使用代码，修改或创建其他代码这主要是一个开发工具，就像“力量倍增器”，允许通过宿主语言（或者元语言）的少量语句生成大量预期的代码。自动化重复的样板代码，这是非常有用的。

大多数编程语言都支持某种形式的元编程。C有一个预处理和C + +的模板。 Java提供了注解和面向方面的编程扩展。脚本语言“EVAL”的语句。大多数语言有一些API，可以用来反省或修改的核心语言特性（如类和方法）。作为最后的手段，任何一种语言，可用于建立源代码，使用字符串操作，然后给它一个编译器。

**代码与数据(Code vs. Data)**

不论实现，元编程系统有一个共同的特点：他们操纵代码作为数据。从概念上讲，程序执行代码销毁或是生产数据作为输入和输出。根据定义，元编程颠倒了这种关系。程式会销毁或产生的代码（作为数据），因此生成的程序运行时，它是执行数据的（如它的代码）。

对于大多数语言中，处理代码作为数据或数据作为代码或多或少的一个繁琐的过程，取决于代码的数据类型。

一个常见的​​策略是把它当​​作一个文本字符串的代码。代码可以创建通过连接关键字，变量名和文字符号，witht4反馈的语言分析器或评估产生的文字。不用说，这可能是非常杂乱和容易混淆，但这是最简单的元编程任务。

另一种策略是提供一套API公开在语言中的对象的编程语言的概念，允许程序员如createClass（）或addMethod（）的调用，编程来构建代码结构。这是更有效地比编写和解析字符串，并广泛地使用在许多面向对象语言。在这种情况下，数据是对象，其中有一个与语言运行时的特殊关系。

**同像(Homoiconicity)**

Clojure的（和其他的Lisp）提供了一个处理代码/数据的区分的第三条道路：没有什么区别。在Clojure，所有代码是数据和所有数据也是代码。

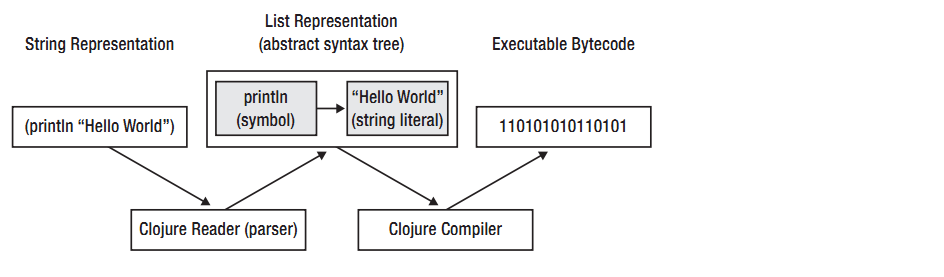
这个属性被称为同像，这意味着该语言的代码是代表语言的数据结构方面。举例来说，这是一个Clojure代码：

**(println "Hello, world")**

这是一个序列（数据）：

**'(println "Hello, world")**

只有一个细微的差别的单引号。它应该只读的list，它在第一代码片段中，代替读取，并立即赋值。这样的形式（称为引用形式）后面禁止读取，而是去进行赋值。



**Figure 12-1**. *How Clojure code is loaded*

关键的一点是，Clojure的源代码没有根本的字符串组成：Clojure的源代码是数据结构vectors，map，和sequences of symbols，文字和其他序列组成。在Clojure，数据结构是非常容易的工作，由于序列抽象。元编程没有更多的困难比创建一个列表了。

**宏(Macros)**

宏是在Clojure的元编程中的主要手段。一个Clojure的宏是一个可以用来改造或更换之前，它是编译的代码构造。在语法上，它们看起来像很多功能，但有几个关键的区别：

* 宏不应该直接返回一个值,而是返回一个形式。
* 宏的参数传递是没有赋值的，他们可以被改变，被忽略，或者添加到宏输出中。
* 宏只在编译时进行赋值操作。

当你在代码中使用宏时，当你在代码中使用宏时，实际上Clojure中是采用宏的返回表达式，而非宏本身。这是一个抽象的手段，并实施控制结构或消除样板或“包装”的代码是非常有用的。

例如，可以定义一个叫做**triple-do**的宏，这个宏接受一个表达式作为参数，可以用这个宏代替do来执行参数表达式3次 。程序员只会键入下列表达式：

**(triple-do (println "Hello"))**

然而，这个表达式被编译成：

**(do (println "Hello") (println "Hello") (println "Hello"))**

除了调试，这里没有让程序员需要关注或关心这其中的形式。他们可以使用它直接在其方案中，不用担心底下的复杂性：

**user=> (triple-do (println "Hello"))**

**Hello**

**Hello**

**Hello**

**Nil**

**使用宏(working with Macros)**

要创建一个宏，使用的**defmacro**宏。这定义了一个函数和寄存器作为宏与Clojure的编译器。此后编译器遇到宏，它会调用的函数和使用返回值，而不是原来的表达式。

**defmacro**需要为**defn**基本相同的参数：一个名字，一个可选的文档字符串，vector作为参数，和body。正如前面提到的，身体应该赋值是一种有效Clojure的形式。在使用中，如果该宏返回的形式语法上是无效的，它会导致一个错误出现。

例如，下面的代码定义的非常简单的**triple-do**宏已经提到：

**(defmacro triple-do [form]**

**(list 'do form form form))**

这只是简单地使用内置的list函数，创造了四个项目的list：特殊形式和重复所提供的形式。请注意，作为引用，所以它被添加到结果列表作为符号，而不是在评估宏体内。如果所提供的形式是**(println "test")**，该list将 **(do (println "test") (println "test") (println "test"))**。这个list是有效Clojure的语法，所以使用宏：

**user=> (triple-do (println "test"))**

**test**

**test**

**test**

**nil**

宏的可能性的另一个例子，它是可以编写一个宏重写infixed数学表达式为一个标准的Clojure的前缀表达，所以它可以进行赋值。例如，它可能会变换（1 +1）的标准（在Clojure）（+1 1）。前缀表示法是Lisp的标准，是所有编程任务中最好的。不要在你的clojure代码中像这样使用。然而，这种功能对于那些不懂lisp的人写dsl是很有用的。

它的第一个有所帮助对于你想要的输入和输出表达式要清晰想法。这个宏，你希望转换表达式像这样：

**(infix (2 + 3))**

到:

**(+ 2 3)**

宏定义是：

**(defmacro infix [form]**

**(cons (second form) (cons (first form) (nnext form))))**

它检查所提供的形式，和使用cons，以建立一个新的表达式，开始与第二项（操作员），再第一项（第一个数字），然后任何其他项。您可以使用以下的代码进行验证：

**user=> (infix (2 + 3))**

**5**

其次，在一般情况下，这是不好的形式作为去解决重新定义标准的方法形式进行赋值。通常情况下，用户应得到他们的表达式是否在一个宏体内或一致的性能。尽管如此，这个例子演示了宏的力量，偶尔也有很好的理由做这种极端变革表达式。

**宏调试(Debugging Macros)**

使用宏可能有点难以理解，因为你要记住，不仅是你写的代码，还有你生成的代码。Clojure的提供了两个函数，帮助调试宏：**macroexpand**和**macroexpand-1**. 他们都作为一个参数一个单引号的形式。他们返回扩展的宏结果评价它，使它能够检查和宏是做什么。**macroexpand**扩展反复给定的形式，直到它不再是一个宏表达式。**macroexpand- 1**的表达扩展，只有一次。他们都扩展只有宏目前在原始表达形式;他们不递归展开目前的宏输出。

下面的例子显示**macroexpand**适用于在上一节所定义的宏：

**user=> (macroexpand '(triple-do (println "test")))**

**(do (println "test") (println "test") (println "test")))**

**user=> (macroexpand '(infix (2 + 3)))**

**(+ 2 3)**

您可以使用不同的表达式和**macroexpand**，看到您的宏的任何参数的输出看起来像什么，即使它可能会很快变得复杂：

**user=> (macroexpand '(triple-do (do (println "a") (println "b"))))**

**(do (do (println "a") (println "b")) (do (println "a") (println "b")) (do (println "a")**

**(println "b")))**

有时候，你可以看到错误，作为防患于未然。例如，如果你传递一个表达式已经前缀**infix**宏，它实际上是将扭转的过程**infix**的结果，这是：

**user=> (macroexpand '(infix (+ 1 2)))**

**(1 + 2)**

使用macroexpand提供了一个有机会看到潜在的问题，然后再尝试评价他们。您还可以对macroexpand输出运行单元测试，以验证您的宏是按预期进行。

**代码模板(Code Templating)**

手动创建的形式返回从宏的函数有时可能会乏味。更糟糕的是，复杂的宏，实际上它可能很难确定输出。

为了缓解这一问题，Clojure的提供了一个代码模板系统。实际上，它可以让宏开发进入宏的返回形式为文字，在必要的值拼接。

模板系统是根据各地的语法引号字符，一个反引号：`。语法引用的作品几乎一模一样经常用单引号引用相同，但有一个重要的例外：在任何时候插入内部的语法被引用的表达式的值，你可以使用引文结束的符号（波浪号，〜）。此外，语法引用内部直接引用的符号假定为顶层，命名空间限定的符号，将扩展等。

**triple-do**宏的模板版本看起来像下面这样：

**(defmacro template-triple-do [form]**

**`(do ~form ~form ~form))**

**do**表达式是代表list文字，和宏函数的返回值。它使用的语法引号字符，以确保它是一个文字处理，而不是马上赋值。里面该语法引号是三个~,他们实际上在里面的文字值点插入的**form**参数的值。

模板**triple-do**扩展到原来的版本是相同的：

**user=> (macroexpand '(template-triple-do (println "test")))**

**(do (println "test") (println "test") (println "test"))**

**拼接引文结束(Splicing Unquotes)**

在语法引号未被引号引起的序列序列并不总是完全按预期进行。有时，插入序列的模板列表，而不是list本身的内容是可取的。要了解原因，尝试实施infix宏如前所述，使用模板：

**(defmacro template-infix [form]**

**`(~(second form) ~(first form) ~(nnext form)))**

它看起来应该正常工作。但尝试扩展它：

**user=> (macroexpand '(template-infix (1 + 3)))**

**(+ 1 (3))**

有一个额外的括号在3左右，这将导致问题。原因是〜（**nnext**形式）的表达式解析为一个list，而不是一个单独的符号。在这种情况下，你要插入序列（**nnext**形式），而不是序列本身返回的内容。

要插入一个list的内容，使用拼接引文结束，〜@。〜@插入值的一个序列，连续到父序列。使用它，而不是在**template-infix**宏的正常引文结束，得到正确的结果：

**(defmacro template-infix [form]**

**`(~(second form) ~(first form) ~@(nnext form)))**

**user=> (macroexpand '(template-infix (1 + 3)))**

**(+ 1 3)**

**生成符号(Generating Symbols)**

Clojure的宏一个非常重要的规则，尽管它有可能在宏生成的代码创建和绑定的本地符号，如本地的名称可能不与任何现有的符号冲突。但是，这是有问题的：写一个宏时，它是不可能知道所有以后可能运行宏的潜在环境。所以Clojure的强制执行的规则：不绑定在宏命名的符号。

尽管如此，有时有必要在一个宏定义的本地符号。为了绕开这个限制，Clojure的提供了一个功能称为自动***gensym***引用内语法形式。在任何语法引用的形式（形式使用的back - tick，`），您可以附加任何地方的符号名的＃字符，宏展开时，它会替换为一个随机生成的符号，保证不会与任何冲突的符号，并会匹配自动***gensym***在相同的语法引号模板创建的任何其他符号。只要你对他们使用的自动***gensym***功能，您可以定义为许多地方的符号为你喜欢宏定义。

看这样一个例子，研究**debug-println**执行相同的函数作为**println**的宏，但返回**nil**，而不是返回表达式的值。这允许它里面的表达式中使用和调试它们。你想能这样使用它：

**(+ 5 (\* 4 (debug-println (/ 4 3)))**

首先，确定你想要生成的代码看起来像。在这种情况下，它像下面这样：

**(let [result (/ 4 3)]**

**(println (str "Value is: " result))**

**result)**

然后构建宏定义。注意**result**符号是如何使用自动***gensym***功能：

**(defmacro debug-println [expr]**

**`(let [result# ~expr]**

**(println (str "Value is: " result#))**

**result#))**

调用**macroexpand-1**展示了所生成的符号名：

**user=> (macroexpand '(debug-println (/ 4 3)))**

**(clojure.core/let [result\_2349\_auto (/ 4 3)]**

**(clojure.core/println (clojure.core/str "Value is: " result\_2349\_auto)**

**result\_2349\_auto)**

除了对result符号名的更改，和全限定的函数名之外，这看起来完全就是我们所需要的。而且它能够生效！

**user=> (+ 5 (\* 4 (debug-println (/ 4 3)))**

**Value is: 4/3**

**31/3**

**什么时候使用宏(When to Use Macros)**

宏有极其强大的力量，可以让你用其它方法做不到的方式控制和抽象代码。他们工作在一个更高的抽象层，所以，他们都明显更难以正常的代码的原因。如果出现问题，它可能要很麻烦调试，因为有多余的一级之间的问题实际上是一个间接的错误消息的来源。