**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。

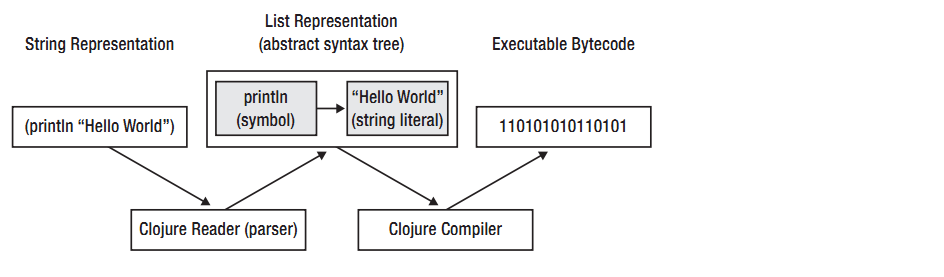
这个属性被称为同像，这意味着该语言的代码是代表语言的数据结构方面。举例来说，这是一个Clojure代码：

**(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。

Therefore, the best way to use macros is to use them as little as possible. A few macros go a long way. Most things you need macros for (including some of the examples in this chapter) could also be accomplished with first-class functions. When you can, do that instead, and don't use macros.

这就是说，有某些情况下使用宏是最好的，最简单的，只有这样，才能完成一个给定的任务。通常情况下，他们分为以下类别之一：

* *实现控制结构*：宏和函数之间的主要区别之一是，宏的参数都不会计算。如果你需要写一个控制结构，可能无法赋值给它的一些参数，因为它是一个宏。
* *Wrap****def****或****defn***: 通常情况下，你只希望在编译时调用def或defn。编写一个程序,当程序运行时调用它们通常是一个灾难, Usually, you only want to call def or defn at compile time. Calling them programmatically while a program is running is usually a recipe for disaster. So, if you need to wrap their behavior in additional logic, the best place to do it is usually a macro.
* 性能：因为他们是扩展在编译时，使用宏可以比调用一个函数更快。通常，这并没有多大的差别，但在极其严密的循环，你有时可以通过消除一个或两个函数调用和使用宏，而不是勉强维持性能。
* Codify reoccurring patterns:宏可以规整任何模式这普遍发生在你的代码中, 从本质上讲，宏语言本身，为满足您的需求而修改你的手段。宏不是只有这样，才能做到这一点，但他们有时可以做的方式，至少侵入到你的代码的其他部分。

**使用宏（Using Macros）**

了解宏，知道什么时候可以使用，这是一件艰难的事情。所以，在有益的范围内获得哪些宏可以用于你的东西。不幸的是，没有可以选择的例子，可以完全涵盖可以做的事用宏类型：宏代表没有更改语言本身的能力，并可能会希望这样做的潜在的方法之一也多种多样。不过有一些常见模式，通常宏熟悉他们可以给你一个了解的头开始时，他们可以用实施。

**实现控制结构(Implementing a Control Structure)**

如前所述，宏和函数之间的重要区别之一是因为宏是在编译之前扩展，而不是在运行时，它们的参数可能全部无法赋值。这是一个控制结构的重要组成部分，它是必要的，只有一些提供的表达式实际上评估，并不是所有的。

考虑这两个表达式，并执行其中只有一个随机的一个控制形式。这可能是在一场比赛中，或在人工智能的实现。您希望它看起来像下面这样：

**(rand-expr (println "A") (println "B") )**

这不能用函数来实现，被称为**rand-expr**是因为两个**println**语句作为参数进行赋值。但你想，只有一个表达式评估随机。这只能用一个宏完成。

要做的第一件事是您要宏拓展的形式来规划。在这种情况下，它包括提供的随机挑选一个表达式的逻辑。扩展应该是这个样子：

**(let [n (rand-int 2)]**

**(if (zero? n) (println "A") (println "B")))**

首先，宏需要挑选一个随机数0和1之间。然后，如果数字为0，它执行的第一个表达式，否则第二。

这个宏也比较简单，语法所描述：

**(defmacro rand-expr [form1 form2]**

**`(let [n# (rand-int 2)]**

**(if (zero? n#) ~form1 ~form2)))**

而且，它的工作正如所料，相同的表达式有时赋值（println“A”），有时（println “B”），从来都。

**user=> (rand-expr (println "A") (println "B"))**

**B**

**nil**

**user=> (rand-expr (println "A") (println "B"))**

**B**

**nil**

**user=> (rand-expr (println "A") (println "B"))**

**A**

**nil**

**实现可变参数的宏(Implementing a Macro with Variadic Arguments)**

宏可以取可变数量的参数。这方面的一个例子是前面的宏，但同要求，随机赋值任何数量的表达式，而不是仅仅两个。

**(rand-expr-multi (println "A") (println "B") (println "C"))**

创建一个宏，它作为“参数”可变数目的形式是很容易做到，以同样的方式，因为它是一个函数：

**(defmacro rand-expr-multi [& forms] …)**

宏体是什么？如何处理可变数量的参数吗？很明显，因为你不知道有多少，你不能只是引用名称和插槽他们在表达在**rand-expr**的第一个草稿。你也许会使用类似第n个功能选择从列表中随机表达，但考虑：宏扩大之际，当你建立的结构，你不访问随机值。内扩展在运行时产生的。如果您在编译时生成的，它会有效地成为一个常数。没有获得随机值在扩张时，你需要Clojure的更原始的控制结构作为选择的所有可能的表达式列表。宏扩展是一个纯粹的代码的转化过程中牢记这一事实将有助于避免关于什么是扩张时间，而不是运行时提供了很多的困惑。

一个可行的解决方案将尝试并生成一个扩展按照这些方针的东西：

**(let [ct (count <number of expressions>))]**

**(case (rand-int ct)**

**0 (println "A")**

**1 (println "B")**

**2 (println "C")))**

最简洁的方法是使用拼接引文结束，拼接形式构成身体的情况下。注意到，这些形式交替索引和表达式可让您使用交错功能，拼接，大大缩短代码生成列表：

**(defmacro rand-expr-multi [& exprs]**

**`(let [ct# ~(count exprs)]**

**(case (rand-int ct#)**

**~@(interleave (range (count exprs)) exprs))))**

它产生预期的扩展：

**user=> (macroexpand-1 ‘(rand-expr-multi (println "A") (println "B") (println "C")))**

**(clojure.core/let [ct\_\_2188\_\_auto\_\_ 3]**

**(clojure.core/case (clojure.core/rand-int ct\_\_2188\_\_auto\_\_)**

**0 (println "A")**

**1 (println "B")**

**2 (println "C")))**

经测试，它按预期工作：

**user=> (rand-expr (println "A") (println "B"))**

**B**

**nil**

**user=> (rand-expr (println "A") (println "B"))**

**A**

**nil**

**user=> (rand-expr (println "A") (println "B"))**

**C**

**nil**

**user=> (rand-expr (println "A") (println "B"))**

**B**

**nil**

**user=> (rand-expr (println "A") (println "B"))**

**B**

**Nil**

**使用递归实现宏(Implementing a Macro Using Recursion)**

宏也可以递归应用。作为一个例子，考虑一个自定义宏，++可以用来代替+，并自动替换嵌套的二进制表达式在Clojure执行略好（见第14章）为更全面地讨论了这个问题多参数此外表达式。换句话说，它需要易于阅读，如表达式（++1 2 3 4 5）和转化他们稍微更好的性能，但更复杂的表达式如（+1（+2（+3（+ 4 5））））。

递归宏和递归函数一样，必须有在他们不再复发，否则他们将永远继续递归导致堆栈溢出错误，但在编译时，而不是运行时的一个基本情况。 + +宏，基本情况是当它通过只有一个或两个参数。在这种情况下，它只是发出一个标准+表达式。当三个或更多的参数，它适用于本身的递归参数列表，发出出额外的每一级递归的嵌套表达式。

这是最简单的,看看代码:

**(defmacro ++ [& exprs]**

**(if (>= 2 (count exprs)**

**`(+ ~@exprs)**

**`(+ ~@(first exprs) (++ ~@(rest exprs)))))**

它是非常简单的。如果条件之一，基础和递归之间的区别。在基础情况下，简单地拼接成一个“+”函数的直接应用所提供的表达式。在递归的情况下，它也创建了一个+功能的应用，在第一个表达式作为第一个参数和接头。对于第二个参数，递归插入+，作为它的参数表达式的其余部分的拼接。

当宏展开，第一层是解开，并显示它是正确的，至少到目前为止。

**user=> (macroexpand '(++ 1 2 3 4))**

**(clojure.core/+ 1 (user/++ 2 3 4))**

要看到整个递归扩展，您可以使用斯图尔特塞拉利昂的clojure.walk库，这是与Clojure的包装。它包括一个macroexpand-all macroexpand或macroexpand- 1，不同的是，不递归展开所有的宏，它可以发现直至有没有离开。导入和运行macroexpand-all所有给出最终的扩展：

**user=> (clojure.walk/macroexpand-all '(++ 1 2 3 4))**

**(clojure.core/+ 1 (clojure.core/+ 2 (clojure.core/+ 3 4)))**

实际使用宏显示为+具有相同的语义。应该不断使速度稍快，以及，虽然不同的是没有通过严谨的基准检测。

**user=> (++ 1 2 3 4)**

**10**

**使用宏创建的DSL(Using Macros to Create DSLs)**

宏的一个常见的用途是生成自定义的DSL。使用宏，一些简单，直观的表达式可以生成更笨重，复杂的代码，不暴露给用户。

在Clojure DSL的潜力是无限的。Enclojure（目前盛行的Clojure的网站框架）允许用户定义Web应用程序的路径和RESTful API的使用一个简单的，立即可以理解的DSL语法。另一个Clojure的项目，Incanter，提供R编程语言，是令人难以置信的简洁和有用的，做统计和建设图表为基础的DSL。

Clojure的DSL是特别有效，因为有一个API和一个DSL之间没有明显的区别。每精心设计Clojure的API自动结束寻找了很多像DSL，和Clojure的计划变得更加复杂，他们倾向于发展高层次的功能和宏是非常容易阅读。

下面的宏演示了一个非常简陋Clojure的DSL，一个使用Clojure的表达式来构建一些非常类似于XML（减去复杂的属性和命名空间）。

这里显示的是XML宏宏前面的例子稍有不同，它的扩展是一个字符串，而不是形式的集合。宏是用来代替一个函数，因为DSL重写的正常处理所提供的形式，使它们为一个字符串，而不是赋值工作。在Clojure这不是最好的方式来处理XML，由一个长期拍摄，看看clojure.xml，clojure.zip，和斯图尔特塞拉利昂的clojure.contrib.prxml库。这只是一个小的，易于管理的例子，将显示一些宏提供的多功能性。

宏的输入仅仅是一系列嵌套形式。形式没有解析：它们将被改造成一个字符串宏无需被赋值。宏转换这样输入：

**(xml**

**(book**

**(authors**

**(author "Luke")**

**(author "Stuart"))))**

这样的输出：

**<book><authors><author>Luke</author><author>Stuart</author></authors></book>**

代码本身是如下：

**(defn xml-helper [form]**

**(if (not (seq? form))**

**(str form)**

**(let [name (first form)**

**children (rest form)]**

**(str "<" name ">"**

**(apply str (map xml-helper children))**

**"</" name ">"))))**

**(defmacro xml [form]**

**(xml-helper form))**

宏是非常轻量级的。它是通过一个单一的形式，它立即传递一个辅助函数。宏辅助的函数是一种常见的的惯用法。通常，在这种情况下，宏本身并不做任何工作，但只是为了获得一个序列的原始形态。从那里，的函数可以做所有的实际工作转化。如果这是可能的，它通常是可取的，因为的函数往往比宏更容易理由左右。记住，该函数将在编译时计算，宏展开，因此它不会有你的程序运行时的状态。

**Helper**函数是一个简单的递归函数。基本情况时，提供的形式是一种原始的（不是一个序列）。它只是作为一个字符串返回。当表单是一个序列，它创建并返回一个XML字符串，元素的名称和项目作为它处理递归的子节点，其余的第一项。

宏运行时展示工作:

**user=> (xml (book (authors (author "Luke") (author "Stuart"))))**

**"<book><authors><author>luke</author><author>Stuart</author></authors></book>"**

从XML处理的角度看，它是极其原始，不应该使用任何实际工作。作为宏的功率示范，它是美丽的。嵌套的表达式从XML字符串转换发生在编译时。因为XML是一个宏，它返回一个字符串，程序，使用它实际上是将“看到”的XML作为字符串文字表达！这里显示的迷你XML DSL是现在Clojure的编译器本身的延伸。

显然，这种权力被滥用，它可以使用宏来构建令人难以置信的错综复杂的表达式。在正确使用时，它们提供几乎无限的权力，改变语言，以满足任何需要。

**总结(Summary)**

通过宏，Clojure的提供了强大的，优雅的元编程设施。在Clojure，代码和数据互换，和宏发出数据，成为代码的编译时的函数。

宏可以生成代码，直接或使用语法引用其输出到模板。他们的卫生，由宏约束的符号必须使用自动gensym功能，以避免潜在的碰撞与现有的符号。

虽然他们可以添加到程序的复杂性，使用时审慎宏提供的手段来消除几乎所有重复的样板代码。它们允许开发人员创建语言级控制结构和抽象，扩展正是需要适应的问题域的语言。品味和克制使用的宏，以及与其他Clojure的动态特性，如一流的的函数，允许开发人员创建自定义的DSL，有机地适应他们的系统，以适应一个问题域，而不是被迫重申他们的问题，只是不灵活的系统，以满足需求。