Clojure的环境

“Hello World” in Clojure

要立即开始在Clojure编程，只需打开一个Clojure的REPL，这代表阅读、赋值、打印、循环。REPL是一个简单而强大的的方式作为创建程序交互方式以及与运行中程序进行互动。

开始REPL最简单的方法是直接从系统的命令行1。要这样做，找到你系统目录中的Clojure位置，一个包含”Clojure-1.0.0.jar”的文件，然后键入之后开始Clojure：

**Java –jar Clojure-1.0.0.jar**

这将启动Java虚拟机加载Clojure的环境,就在REPL开始时,你应该看到以下字样:

**user=>**

这表明，REPL是准备接受输入。为了写你的第一个程序，只需在提示符下键入以下内容：

**user=>(println “Hello World”)**

按回车键，REPL应该显示以下：

**Hello World**

**nil**

**user=>**

究竟在这儿发生了什么？首字母缩写的REPL本身提供了一个线索。

*Read*: Clojure的读取键入的内容，（println “Hello World”），并解析它作为  
Clojure的形式，确保它是有效的Clojure的语法。

1这是最简单的方式使用Clojure的，但它绝不是最好的。由于你的程序扩大规模  
和复杂性，你几乎肯定会需要移动到一个更完整的Clojure的开发环境，将提供帮助文件和类路径管理，语法高亮，调试，和其他的基本特征, 和其他插件存在于 Emacs，VI，NetBeans，Eclipse，IntelliJ IDEA和其他编辑器，它们提供这些和其他的功能。

*Evaluate*:Clojure的编译器所提供的形式和赋值。在这种情况下，它调用了一个println函数，一个文本参数，”Hello World”。Clojure执行函数,并按规范体系打印”Hello World”。

*Print*:Clojure打印的println函数是没有返回值的。在这种情况下，它是nil，(与Java的null是相同的，这意味这没有任何值，或者是”没有”[原文为”nothing”])，因为println不是一个有返回值的函数。

*Loop*: Clojure返回到输入提示，随时为您键入另一个形式。

这不同于大部分其他编程语言工作。在大多数语言的书写时,编译时和运行时的程序是非常不同的步骤。Clojure不允许你分开这些步骤，你应该想的，但大多数Clojure的程序员更愿意使用在REPL集成开发，书写，并在同一时间运行他们的代码。这样可以大大缩短开发时间。它允许开发者看到自己的代码做什么，立即在一个已经运行的程序的情况下，没有任何时间上的开销需要去停止程序，编辑代码，重新编译，并再次启动它，这种基本的，自上而下的编码风格很快就开始感觉极为自然，并且很快会感觉一个静态的开发环境缓慢和繁琐。

相比其他“脚本”语言也提供实时的赋值，然而，Clojure的联机(原文为on-the-fly)能力更加健壮。当在REPL赋值时，它不仅只是赋值，实际上是编译，添加到正在运行的程序上与之前的代码的程序状态相当。也不是REPL只有一个特殊的调试功能：动态代码始终是语言所固有的。这是完全可能的，而且并不少见，连接到一个远端Clojure的生产实例，打开REPL,检查应用程序的状态，诊断问题，并调试代码，修正错误，而且程序运行，0停机代码修复。

从理论上来讲，它是可以打开一个REPL，从头开始写一个完整的，复杂程序，从观点根据讲，因为它没有停止或重新启动。

**Clojure的形式(Clojure Froms)**

一个Clojure的程序的基本单位是不是行,关键字，或类，而是形式。在Clojure中，形式可以是任何单位代码可以被赋值并返回一个值。当您在REPL中键入什么东西时，它必须是一个有效的形式和Clojure的源文件包含了一连串形式。有四种形式的基本的品种。

**文字(Literals)**

文字的形式解析自己。文字的例子是，你直接输入到代码的字符串，数字和字符。您可以验证，文字解析自己企图在REPL：

**user=>”I’m a string!”**

**I’m a string!**

当您键入一个简单的，双引号的字符串进行赋值，返回值是字符串本身。同样的事情，数字也是一样的。

**user=> 3**

**3**

**符号(Symbols)**

符号值的形式解析。它们可以被认为是大致类似变量，虽然这在技术上并不准确，因为他们实际上并没有同样的方式在大多数语言变量的变量。在Clojure，符号是用来识别函数的参数，和全局或局部定义的值。符号和他们的分析是在下面的章节详细讨论。

**复合形式(Composite Forms)**

复合形式使用对称的括号，括号或大括号，其他形式的群体。赋值时，其价值取决于什么类型的形式，括号内赋值一个vector和大括号到map。第4章详细讨论了这些类型。

然而，这里的特别感兴趣是复合形式的使用括号。这些表明一个list，和多个list在Clojure，具有特殊的含义。它毕竟是一个Lisp的方言，这源于它的名字“列表处理。”(原文为”LIST Processing.”)

在Clojure（和所有的Lisp）中，列出了赋值函数调用。当一个list赋值，它调用一个相同的函数，赋值操作以值的形式是从该函数的返回值。list中的第一项是要调用的函数，其余项目都作为传递给函数的参数。例如，Clojure的形式**(A B C)**，当赋值时，意思是调用A并且B和C作为它的参数。在其他编程语言可能写作**A(B C)**。

这似乎可能对一个没有Lisp背景的程序员很陌生。然而，在Clojure的能力范围内，优势是相当可观的。？整个程序都是集合构成的，集合中包含集合,以此类推,代码是数据，数据可以被看作代码。在第12章中，你会看到如何可以利用，很容易地创建和编写代码。

**特殊形式(Special Forms)**

特殊形式，是一个特定类型的复合形式。对于大多数用途，使用它们的函数调用非常相似。不同的是，第一种形式的一种特殊形式的，是不是某处定义一个函数，而是一种特殊形式的Clojure的内置。

特殊形式，是一个Clojure的程序最基本的构建模块，是用来控制程序流程，绑定VAR(变量)的，定义除其它外的函数。重要的是要记住的是，如函数调用，list中的第一种形式确定正在使用的一种特殊形式，在list中的其他形式的特殊形式的参数一样。为了能看到的例子中每个这些类型的形式，让我们做一个比较复杂的Hello World程序，你会使用两种形式，而不只是一个。在REPL中，键入以下命令，并按下ENTER：

**user=> (def message "Hello, World!")**

**在下一个提示符下，键入以下内容：**

**user=> (println message)**

**您应该看到的第一个Hello World程序相同的输出：**

**Hello, World**

**nil**

这个简单的程序，只有两种形式，包含前面讨论的每种类型的形式。

分析第一种形式,**(def message “Hello World!”)** ,您最先看到它是括号里。因此它是一个list, 将作为一个函数的应用程序或一种特殊的形式进行赋值。List中的项目有3个:**def,message,”Hello World!”**。Def是第一项，也就是所谓的函数或者是特殊形式，在这种情况下它是特殊形式，但是，它像一个函数，它需要2个参数var来定义，和值来绑定它，赋值，这种形式并创建一个变量，建立一个值绑定**”Hello world!”**符号**message**。

第二种形式**（println message）**也是一个list，这个时候它的正常函数的应用。它有两个组成部分形式，他们中的每一个符号。符号**println**解析为**println**函数，符号**message**解析为字符串**“Hello World!”**，因为在前面的形式确立了VAR绑定约束。

那么，最终的结果是与第一个Hello World程序相同的**println**函数的参数称为**“Hello World!”**

**编写和运行源文件(Writing and Running Source Files)**

由于REPL非常方便，在真实开发过程中，也会有保存源代码并使其能够重用并不需要重写代码的需要。CLOJURE当然也有这个功能。

按照惯例，Clojure的源代码文件的扩展名为\*. CLJ。在一个正常的Clojure的程序，没有必要显式编译源文件，它们会被自动加载，因为它们是编译的，就像个体形式进入在REPL。如果您需要预编译Clojure的标准Java的\*. class文件，（例如，运行在一个非标准的Java环境，如移动电话），它是完全可能的，并Clojures AOT（提前[Ahead Of Time]）编译处理功能。这些都是在第10章讨论。

要运行这个例子从一个\*. CLJ文件的Hello World程序，创建一个新文件，名为“HELLO - world.clj”在任何纯文本编辑器，包含下面的代码清单2-1。

*清单2-1。 HELLO- world.clj*

**(def message1 "Hello, World!")**

**(def message2 "I'm running Clojure code from a file.")**

**(println message1)**

**(println message2)**

有两种方法运行此文件。最简单，最经常用于发展，打开一个REPL和键入以下（代\*. CLJ文件的实际路径，并在按照与Java公约中使用正斜杠）：

**user=> (load-file "c:/hello-world.clj")**

你应该看到下面的输出：

**Hello, World!**

**I'm running Clojure code from a file.**

**Nil**

load-file函数接受一个参数：一个文件系统路径的字符串表示。然后加载在路径中找到的文件，并执行该文件中的每个list顺序，就好像它已在REPL输入，并返回在文件的最后形式的返回值。你可以看到nil，println的返回值作为输出的最后一行。在文件中定义的所有符号都仍然可用。尝试输入在REPL文件中定义的一个符号，它能够解析绑定到它的的值：

**user=> message1**

**"Hello, World!"**

另一种方式来执行Clojure的文件是直接从系统的命令行。这种方法产生一个新的Clojure的运行时在一个新的Java虚拟机实例，然后立即载入选定的文件。这是正常运行Clojure的开发程序（除非你打包成\*. class文件Clojure的一个jar包）的方法。要运行这样一个Clojure的文件，只需在命令行中输入以下内容：

**java –jar c:/clojure-1.0.0.jar c:/hello-world.clj**

Java将其识别为一个Java标准调用。**c:/clojure-1.0.0.jar**，确保Clojure的运行时库是在当前的CLASSPATH。修改路径以反映您的Clojure的jar文件的实际位置与Clojure的安装。最后一个参数是你要运行的脚本的路径。

此命令启动Clojure的运行时，加载HELLO - world.clj的文件，并按顺序赋值给它们的形式。在这种情况下，您在系统控制台中看到的结果仅是那些打印到标准系统输出：

**Hello, World!**

**I'm running Clojure code from a file.**

**Vars, Namespaces, and the Environment**

正如在第一章提到，Clojure的程序是灵活的，基本的实体，它可以进化，而无需关闭和重新运行。这主要是由于REPL的存在，提供的能力，以赋值在现有程序的情况下的形式。但这个工作到底是如何呢？

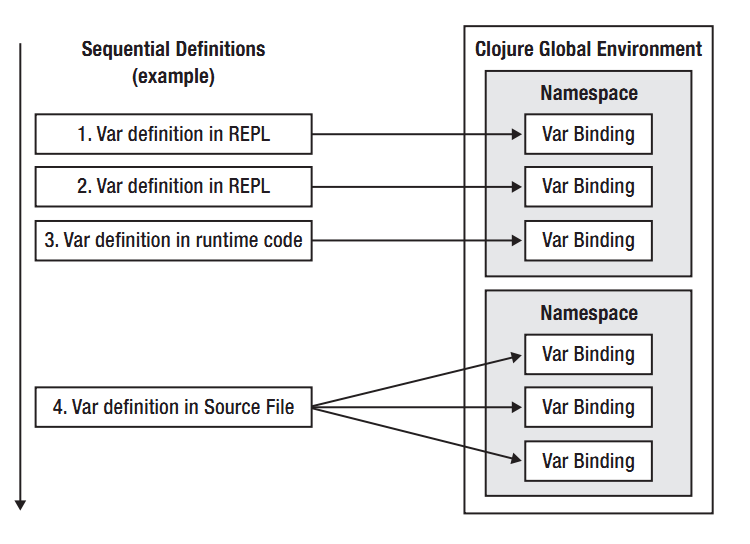
当您启动Clojure的程序，通过开一个新的REPL或直接运行一个源文件，你正在创建一个新的全局环境。这种环境持续，一直到程序终止，并包含所有需要运行的程序，包括全局变量，（绑定的值的名称）的信息。参见**图2-1**，它被添加（或保留）的全局环境。之后被保留，它是从任何地方引用可用的，在相同的环境中。在Hello World示例，您创建了一个变量符号**message**绑定到一个字符串值，你可以看到这一点，并在以后使用。

变量可以使用def的一种特殊形式的符号定义和约束。它的语法如下：

**(def var-name var-value)**

var-name是变量创建的名称，和var-value就是它的值。var-value可以是任何Clojure的形式，将赋值和由此产生的值绑定到了var。然后，每当var-name的符号在全局范围内的Clojure的环境中使用，它能够解析的var-value。

**注意**: 一定要以正确的顺序定义你的依赖。由于Clojure引用var方式，必须定义一个在var的引用符号，可以进行赋值。通常情况下，这不是一个问题，但它可以导致一些“陷阱”，如果你在REPL做了很多工作。由于我们经常会在代码中用不同的顺序来进行REPL的定义，又由于这些代码一旦被输入REPL后，这些代码在整个程序中都是一直有效的。在我们的工作中，你或许只会在停止了整个程序之后才会发现你定义过的依赖性（dependency ）失效了。这是一个很容易解决的问题，一旦我们注意了这一点，也是很容避免类似的错误的。但是这个问题确实给Clojure初学者们造成了很多的困惑。



*图2-1 Clojure的环境*

**Are Vars Variables?**

虽然他们有许多相似之处，vars并不像其他编程语言中的变量，最重要的是，一旦定义，他们不打算改变，起码，不作为一部分普通运行的程序。这是真的，如果你使用一个已经绑定的var def，其value将发生变化，后续赋值解析会以新的value为准。而这不是线程安全的，只能用DEF定义的全局符号了。可变的全局符号，将使您程序工作的一部分破坏，即使你可能可以得到它的运行。如果您需要使用多变的值作为你的程序的一部分，全局或以其他方式，你应该总是​​使用Clojure的线程安全的引用类型和重新定义符号。

这就是说，有一个很好的适当使用，重新定义现有的值：手动更新或改变一个程序运行时。这是Clojure的一种能力，重新绑定一个符号，它允许你建立或更改，而无需重新启动一个程序。当你做探索性编程时，重新绑定值是Clojure的优点。另一个例子可能是您的基于服务器的程序使用一个符号来存储一个特定的常量，就是说，**\*max-users\***，并且您以后决定，该系统可以处理更多的用户，你应该碰到。在这种情况下，这是完全可以的重新定义符号的值，而无需重新启动程序。关键的一点是不依赖于方案的符号重新定义他们使用可变状态。在任何情况下多线程，这是极不安全的，它可能是很糟糕的表现，是在任何情况下破坏Clojure的做法。

**Symbols and Symbol Resolution**

在Clojure中符号是无处不在的(Symbols)，它值得让你花费一些时间来了解他们真正和它们是如何工作的。概括地说，一个符号，是一个标识符解析值。它们可以被定义在地方一级（例如，函数的参数或本地绑定），或全局（使用vars）。您看到任何关于Clojure代码，是没有文字或一个基本的语法字符（引号，括号，大括号，方括号，等）中看到的任何东西都可能是一个符号。这涵盖什么通常是认为在其他语言中的变量，但还一个很好的协议更多：

* Clojure的所有函数名的符号。当一个函数被称为作为一个复合形式的一部分，它首先解析符号的功能，然后将其应用于。
* 大多数运算符（相比较而言，数学等）的符号，从而解决一个特别的、内置的、优化的函数。他们解决的和应用在一起作为额外的性能优化的函数的方式。
* 宏（Macro）名称是一种符号。在这不做详细，宏像函数，只有在编译时而不是运行时应用，请参阅第12章宏在深入讨论。

**符号名称（Symbol Names）**

符号名是区分大小写的，和用户定义的符号有以下限制：

* 可以包含任何字母数字字符，字符\*, +, !, -, \_, and ?.
* 不得以数字开头。
* 可能包含冒号字符：，不是在开始或结束的符号名，并可能不会重复。

根据这些规则，合法符号名的例子包括符号名，symbol\_name，symbol123，\*symbol\*, symbol! , symbol? , 和name+symbol. 非法符号名的例子123symbol, :symbol: , symbol//name, 等。

按照惯例，通常在Clojure符号名小写，划线字符（ - ）分隔的单词。如果一个符号是一个常量或全局程序设置，它往往开始和结束的星符号（\*）。例如，一个程序可能定义（def \* PI\* 3.14159）。

**符号的解析和范围（Symbol Resolution and Scope）**

当您在您的代码中使用一个符号名称，Clojure赋值给符号，并返回绑定到它的值。这样的解析是如何产生的，视符号范围而定，无论是用户定义的，或者是特殊形式和内置形式。

Clojure的使用以下步骤解析符号：

1. Clojure判断，如果该符号指向一种特殊的形式。如果是这样，并相应地使用。
2. 其次，如果Clojure的检查符号是本地的绑定。通常情况下，本地绑定这意味着它是一个函数的参数，或**let**（第3章中讨论）定义。如果它找到一个本地的值，它将使用它。请注意，这意味着，如果有一个本地定义的符号和VAR具有相同的名称，赋值的符号名，将返回本地符号的值。局部符号覆盖相同的Vars。
3. Clojure的搜索在全局环境中对一个var的名称的符号对应，并返回该值。
4. 如果Clojure在前面的步骤中并未发现一个value的符号名称，就返回一个错误：java.lang.Exception: unable to resolve symbol <symbol> in this context (NO\_SOURCE\_FILE:0). (未能解析符号在上下文中，NO\_SOURCE\_FILE部分将被替换为实际的文件名，除非你是在REPL运行。)

**命名空间Namespaces**

当你使用def定义一个var时，你正在对该value定义一个全局绑定的符号名。然而，真正的全局变量和符号早就被称为是一个坏主意，在一个大程序，它是太容易在一个程序的一部分中定义，在不经意间与另一个发生碰撞，导致困难和极难发现错误。

出于这个原因，在Clojure中所有vars都在命名空间范围内的。每个var的一部分作为命名空间（有时是隐式显示的）当使用一个符号来指向一个var时，您可以用正斜杠符号（/）在符号前制定命名空间。

看到这，仔细看下，一个符号定义在REPL。

**user=> (def first-name "Luke")**

**#'user/first-name**

**user=> user/first-name**

**"Luke"**

请注意提示本身：**user=>**。提示中的字符串**user**实际上指的是当前的命名空间。如果你工作在不同的命名空间，它会显示不同的东西。没用关于user的特殊命名空间，它不是一个特殊的，它只是个默认的。你实际上还没有定义，当你定义**user/first-name**然后就可以使用赋值的符号了。既然你已经在**user**空间了，使用**first-name**也可以工作。

**声明命名空间（Declaring Namespaces）**

声明命名空间，使用ns的形式。ns需要大量的参数，其中一些比较高级的。最简单的形式，您可以传入一个参数，作为命名空间的名称。如果命名空间不存在，它会创建，并设置成当前命名空间，如果已经存在，它会切换到命名空间。

**user=> (ns new-namespace)**

**nil**

**new-namespace=>**

现在，当你定义一个变量时，将传入**new-namespace**的命名空间，而不是**user。**

**引用命名空间（Referencing Namespaces）**

为了引用一个不同的命名空间中的var，只需使用其完整的名称。请遵守以下REPL会话:

**user=> (def my-number 5)**

**#'user/my-number**

**user=> (ns other-namespace)**

**nil**

**other-namespace=> my-number**

**java.lang.Exception: Unable to resolve symbol: my-number in this context...**

**other-namespace=> user/my-number**

**5**

在这里，你首先在默认的用户命名空间中定义一个var。然后，创建一个新的命名空间，并切换到它。当您尝试赋值**my-nameber**，它会导致一个错误：无法在当前命名空间中找到**my-namber**。然而，当您使用完全合格的名称，它解析了var，并传回你原来绑定到它的值。您只能使用完全合格的名称来赋值var。要定义一个命名空间内的一个符号，你必须确实你在命名空间中。

有时，如果你是严重依赖于另一个命名空间，它的完全限定每个引用您需要该命名空间中的一个var太麻烦了。对于这种情况，Clojure的提供这种功能。使一个命名空间“包括”（include），使用**:use** ,NS参数。例如，申报进口Clojure的内置XML库，你可以做到这一点的所有符号的命名空间：

**user=> (ns my-namespace**

**(:use clojure.xml))**

**my-namespace=>**

现在，所有的XML相关的符号，已经在我的命名空间中了。（**:use clojure.xml**）的形式指定clojure.xml命名空间将被加载，还导入到我的命名空间中定义的符号。这也是非常有用的依赖管理，而不是要求您手动加载clojure.xml使用它之前，你可以使用**:use**指定它作为一个命名空间声明的依赖。Clojure的命名空间声明的一部分，然后加载它，如果它不是已经加载，确保它始终是您的新的命名空间内。

除了**:use**，Clojure的提供了另一种可以使用NS，**:require**。用法与**:use**相同，区别在于，它不仅确保必要的命名空间加载，并导入没有应用的符号。您还可以使用**:require**指定命名空间列表包括进来。在这里你包括Clojure.xml库及clojure.set：

**user=> (ns my-namespace**

**(:require clojure.xml**

**clojure.set))**

**my-namespace=>**

此外，您可以括在方括号的命名空间和使用**:as**关键字来指定一个较短的别名命名空间：

**user=> (ns my-namespace**

**(:require [clojure.xml :as xml]))**

**my-namespace=> xml/parse**

**my-namespace=> #<xml$parse\_7630 clojure.xml$parse\_7630@1484105>**

不用担心那些凌乱的值，它是作为clojure的一个函数的字符串表示形式，表明Clojure是能够解析**xml/parse** 符号的。

**构建源文件(Structuring Source Files)**

如何使用命名空间来组织你的源代码，并保持组织样式？这不难。照惯例，每个Clojure的源文件有其自己的命名空间的一个ns应该是在任何Clojure的文件第一种形式的声明，这使得它易于管理的命名空间和文件。它也是类似于每一个Java公约类。事实上，它可能会帮助Java程序员想到，类的命名空间。它也可以像类一样将相互有关的代码分组到一起。

为了帮助Clojure的发现命名空间，引用**:use**或**:require**，是遵循一个特定的命名约定。在一个文件中声明的命名空间，必须在类路径的文件名称和位置相匹配。如果你有一个“x/y/z.clj”Clojure的源文件，它应该包含命名空间x.y.z的声明当你参考x.y.z，它会知道在哪个路径和文件搜索该命名空间。再次，这是非常相似的Java包结构。

**总结(Summary)**

这是所有真正需要的知识运行Clojure的方案。当然，你将要学习用工具，以帮助使源文件更容易地管理和运行。特别是，类路径可能是痛苦的管理，Eclipse或NetBeans等工具可以减轻这一负担。大多数Clojure环境都提供了另一个有用的特性，那就是可以打开一个文件并选择性的执行一些单个的形式，而不是加载整个文件。这对敏捷开发、测试、和现有的应用程序调试是非常宝贵的。

要记住，无论你使用哪一种工具，重要的是，Clojure的方案包括完全的形式，其本身无论是文字，特殊形式，符号，或其他形式的复合。牢记这是朝着理解Clojure的程序结构的一个大步。

此外，重要的是理解的符号。符号是源代码中的标识符与实际值的手段，它是有助于清晰的把握，它们如何分配和得到解析。

常用的var与符号的结合。var代表了一个名字，一个在Clojure的环境value的约束与绑定，并通过命名空间范围内。

最后，较高水平，当一个程序一个源文件变得太大便分解成多个文件，并给每个文件一个单独的命名空间。然后，您可以使用的命名空间的依赖功能，以确保符号总是定义需要它们的地方。