第15章

15 实践：可移植路径名库

如同我在前面章节里讨论的那样，Common Lisp提供了一种称为路径名的抽象，这样一来，你就不用顾忌不同操作系统和文件系统在文件命名方式上的差异。路径名提供了一个有用的API来管理作为路径的名字，但是当它涉及实际与文件系统交互的函数时，事情就会变得有些复杂。

如同我提到的，问题的根源在于，路径名抽象被设计用来表示比当今常用的文件系统更加广泛的系统上的文件名。不幸的是，为了让路径名足够抽象从而可以应用于广泛的文件系统，Common Lisp的设计者们留给了实现者们大量的选择空间，来决定究竟如何将路径名抽象映射到任何特定文件系统上。这样带来的结果是，不同的实现者虽然在相同的文件系统上实现路径名抽象，但在一些关键点上却做出了不同的选择，从而导致遵循标准的实现在一些主要的路径名相关函数上不可避免地提供了不同的行为。

然而，所有的实现都以这样或那样的方式提供了相同的基本功能，因此你可以写一个库，对多数跨越不同实现的常见操作提供一致的接口。这就是你在本章中的任务。编写这个库，你不但可以获得后续几章将会用到的几个有用的函数，还可以有机会学习如何编写处理不同Lisp实现间区别的代码。

15.1 API

该库支持的基本操作是获取目录中的文件列表，并检测给定名字的文件或目录是否存在。你也将编写函数用于递归遍历目录层次，并在目录树的每个路径名上调用给定的函数。

从理论上来讲，这些列目录和测试文件存在性的操作已经由标准函数`DIRECTORY`和`PROBE-FILE`提供了。不过正如你将看到的那样，会有许多不同的方式来实现这些函数——所有这些均属于语言标准的有效解释，因此你希望编写新的函数，以在不同实现间提供一致的行为。

15.2 `*FEATURES*`和读取期条件化

在能够实现这个可在多个Common Lisp实现上正确运行的库的API之前，我需要首先介绍编写实现相关代码的手法。

“在任何符合标准的Common Lisp实现上正确运行的代码都将产生相同的行为”，从这个意义

www.TopSage.com

上说，尽管你所编写的多数代码都是“可移植的”，但你可能偶尔需要依赖于实现相关的功能，或者为不同实现编写稍有差别的代码。为了使你在不完全破坏代码可移植性的情况下做到这点，Common Lisp提供了一种称为读取期条件化的机制，从而允许你有条件地包含基于当前运行的实现等各种特性的代码。

该机制由一个变量***FEATURES***和两个被Lisp读取器理解的附加语法构成。***FEATURES***是一个符号的列表，每个符号代表存在于当前实现或底层平台的一个“特性”。这些符号随后用在特性表达式中，根据表达式中的符号是否存在于***FEATURES***求值为真或假。最简单的特性表达式是单个符号，当符号在***FEATURES***中时该表达式为真，否则为假。其他的特性表达式是构造在**NOT**、**AND**和**OR**操作符上的布尔表达式。例如，如果要条件化某些代码使其只有当特性foo和bar存在时才被包含，那么可以将特性表达式写成(and foo bar)。

读取器将特性表达式与两个语法标记#+和#-配合使用。当读取器看到任何一个这样的语法时，它首先读取特性表达式并按照我刚刚描述的方式求值。当跟在#+之后的特性表达式为真时，读取器会正常读取下一个表达式；否则它会跳过下一个表达式，将它作为空白对待。#-以相同的方式工作，只是它在特性表达式为假时才读取后面的形式，而在特性表达式为真时跳过它。

FEATURES的初始值是实现相关的，并且任何给定符号的存在所代表的功能也同样是由实现定义的。尽管如此，所有的实现都包含至少一个符号来指示当前是什么实现。例如，Allegro Common Lisp含有符号:allegro，CLISP含有:clisp，SBCL含有:sbcl，而CMUCL含有:cmu。为了避免依赖于在不同实践中可能不存在的包，***FEATURES***中的符号通常是关键字，并且读取器在读取特性表达式时将***PACKAGE***绑定到**KEYWORD**包上。这样，不带包限定符的名字将被读取成关键字符号。因此，你可以像下面这样编写一个在前面提到的每个实现中行为稍有不同的函数：

```
(defun foo ()
  #+allegro (do-one-thing)
  #+sbcl (do-another-thing)
  #+clisp (something-else)
  #+cmu (yet-another-version)
  #-(or allegro sbcl clisp cmu) (error "Not implemented"))
```

在Allegro中读取上述代码，就好像代码原本就写成下面这样：

```
(defun foo ()
  (do-one-thing))
```

在SBCL中读取器将读到下面的内容：

```
(defun foo ()
  (do-another-thing))
```

而在一个不属于上述特定条件化实现的平台上，它将被读取成下面这样：

```
(defun foo ()
  (error "Not implemented"))
```

因为条件化过程发生在读取器中，编译器根本无法看到被跳过的表达式，[1]这意味着你不会

① 这种读取器条件化工作方式所带来的一个稍为麻烦的后果是，无法简单地编写fall-through case。例如，如果通过在foo中增加另一个#+前缀的表达式来为其添加对另一种实现的支持，那么你需要记得也要在#-之后的or特性表达式中添加同样的特性，否则**ERROR**形式将会在新代码运行以后被求值。

www.TopSage.com

为不同实现的不同版本付出任何运行时代价。另外，当读取器跳过条件化的表达式时，它不会保留其中的符号，因此被跳过的表达式可以安全地包含在其他实现中可能不存在的包中的符号。

对库打包

从包的角度讲，如果你下载了该库的完整代码，会看到它被定义在一个新的包com.gigamonkeys.pathnames中。我将在第21章讨论定义以及使用包的细节。目前你应当注意，某些实现提供了它们自己的包，其中含有一些函数与你将在本章中定义的一些函数有相同的名字，并且这些名字可在CL-USER包中访问。这样，如果你试图在CL-USER包中定义该库中的某些函数，可能会得到关于破坏了已有定义的错误或警告。为了避免发生这种情况，你可以创建一个称为packages.lisp的文件，其中带有下面的内容：

```
(in-package :cl-user)

(defpackage :com.gigamonkeys.pathnames
  (:use :common-lisp)
  (:export
   :list-directory
   :file-exists-p
   :directory-pathname-p
   :file-pathname-p
   :pathname-as-directory
   :pathname-as-file
   :walk-directory
   :directory-p
   :file-p))
```

接着加载它，然后在REPL中或者在你输入定义的文件顶端，输入下列表达式：

```
(in-package :com.gigamonkeys.pathnames)
```

将库以这种方式打包，除了可以避免与那些已存在于CL-USER包中的符号产生冲突以外，还可以使其更容易被其他代码使用，你在后续几章中将看到这一点。

15.3 列目录

你可以将用于列举单独目录的函数list-directory，实现成标准函数**DIRECTORY**外围的一个包装层。**DIRECTORY**接受一种特殊类型的路径名，称为通配路径名，其带有一个或多个含有特殊值:wild的组件，然后返回一个路径名的列表，用来表示文件系统中匹配该通配路径名的文件。[1]匹配算法和多数在Lisp与一个特定文件系统之间的交互一样，它没有被语言标准定义，但Unix与Windows上的多数实现遵循了相同的基本模式。

① 另一个特殊值:wild-inferiors可以作为一个通配路径名的目录组件的一部分出现，但在本章里你不需要它们。

www.TopSage.com

DIRECTORY函数有两个需要在list-directory中解决的问题。其中主要的一个是，即便在相同的操作系统上，其行为的特定方面在不同的Common Lisp的实现间也具有相当大的区别。另一个问题在于，尽管DIRECTORY提供了一个强大的用于列举文件的接口，但要想正确地使用它需要对路径名抽象有相当细致的理解。在这些不同细微之处和不同实现的特征影响下，实际编写可移植代码来使用DIRECTORY完成一些像列出单个目录中所有文件和子目录这样简单的事情，都可能令人沮丧。你可以通过编写list-directory来一次性地处理所有这些细节和特征，并从此忘记它们。

我在第14章里讨论过的一个细节是，将一个目录的名字表示成路径名有两种方式，即目录形式和文件形式。

为了让DIRECTORY返回/home/peter/中的文件列表，需要传给它一个通配路径名，其目录组件是你想要列出的目录，其名称和类型组件是:wild。因此，为了列出/home/peter/中的文件，需要这样来写：

```
(directory (make-pathname :name :wild :type :wild :defaults home-dir))
```

其中的home-dir是代表/home/peter/的路径名。如果home-dir是以目录形式表示的，那么上述写法是有效的。但如果它以文件形式表示，例如，它通过解析名字字符串"/home /peter"来创建，那么同样的表达式将列出/home中的所有文件，因为名字组件"peter"将被替换成:wild。

为了避免两种形式间的显式转换，可以定义list-directory来接受任何形式的非通配路径名，随后它将被转化成适当的通配路径名。

为此，应当定义一些助手函数。其中一个是component-present-p，它将测试一个路径名的给定组件是否“存在”，也就是说该组件既不是NIL也不是特殊值:unspecific。[①]另一个函数是directory-pathname-p，用来测试一个路径名是否已经是目录形式，而第三个函数pathname-as-directory可以将任何路径名转换成目录形式的路径名。

```
(defun component-present-p (value)
  (and value (not (eql value :unspecific))))

(defun directory-pathname-p  (p)
  (and
   (not (component-present-p (pathname-name p)))
   (not (component-present-p (pathname-type p)))
   p))

(defun pathname-as-directory (name)
  (let ((pathname (pathname name)))
    (when (wild-pathname-p pathname)
      (error "Can't reliably convert wild pathnames."))
    (if (not (directory-pathname-p name))
      (make-pathname
```

① 具体实现返回:unspecific来代替NIL，作为某些特定情况下路径名组件的值，例如当该组件没有被该实现使用时。

www.TopSage.com

```
    :directory (append (or (pathname-directory pathname) (list :relative))
                       (list (file-namestring pathname)))
    :name      nil
    :type      nil
    :defaults pathname)
   pathname)))
```

现在看起来似乎可以通过在由pathname-as-directory返回的目录形式名字上调用**MAKE-PATHNAME**，来生成一个通配路径名并传给**DIRECTORY**。不幸的是，事情没有那么简单，多亏了CLISP的**DIRECTORY**实现中的一个怪癖。在CLISP中，除非通配符中的类型组件是**NIL**而非:wild，**DIRECTORY**将不会返回那些没有扩展名的文件。因此，你可以定义一个函数directory-wildcard，其接受一个目录形式或文件形式的路径名，并返回一个给定实现下的适当的通配符，它通过使用读取期条件化在除CLISP之外的所有实现里生成一个带有:wild类型组件的路径名，而在CLISP中该类型组件为**NIL**。

```
(defun directory-wildcard (dirname)
  (make-pathname
   :name :wild
   :type #-clisp :wild #+clisp nil
   :defaults (pathname-as-directory dirname)))
```

注意每一个读取期条件是怎样在单个表达式层面上操作的。在#-clisp之后，表达式:wild要么被读取要么被跳过；同样，在#+clisp之后，NIL要么被读取要么被跳过。

现在你可以首次看到list-directory函数了：

```
(defun list-directory (dirname)
  (when (wild-pathname-p dirname)
    (error "Can only list concrete directory names."))
  (directory (directory-wildcard dirname)))
```

使用上述定义，该函数在SBCL、CMUCL和LispWorks中正常工作。不幸的是，你还需要谨慎处理另外一些实现间的区别。其中一点是，并非所有实现都返回给定目录中的子目录。Allegro、SBCL、CMUCL和LispWorks可以返回子目录。OpenMCL默认不会这样做，但如果为**DIRECTORY**传递一个实现相关的、值为真的关键字参数:directories，那么它将返回子目录。对于CLISP的**DIRECTORY**，只有当传递给它一个以:wild作为目录组件的最后一个元素且名字和类型组件为NIL的通配路径名时，才可以返回子目录。而且，在这种情况下，它只返回子目录，因此需要使用不同的通配符，调用**DIRECTORY**两次并组合结果。

一旦所有实现都返回目录了，你会发现它们返回的目录名有些是目录形式的，有些是文件形式的。你想要list-directory总是返回目录形式的目录名，以便可以只通过名字来区分子目录和正规文件。除Allegro之外，所有实现都支持做到这点。Allegro要求为**DIRECTORY**传递一个实现相关的、值为NIL的关键字参数:directories-are-files，从而使其以目录形式返回目录。

一旦你知道如何使每一个实现做到你想要的事，那么实际编写list-directory就只是简单地将不同版本用读取期条件组合起来了。

```
(defun list-directory (dirname)
  (when (wild-pathname-p dirname)
```

www.TopSage.com

```
    (error "Can only list concrete directory names."))
  (let ((wildcard (directory-wildcard dirname)))

    #+(or sbcl cmu lispworks)
    (directory wildcard)

    #+openmcl
    (directory wildcard :directories t)

    #+allegro
    (directory wildcard :directories-are-files nil)

    #+clisp
    (nconc
     (directory wildcard)
     (directory (clisp-subdirectories-wildcard wildcard)))

    #-(or sbcl cmu lispworks openmcl allegro clisp)
    (error "list-directory not implemented")))
```

函数clisp-subdirectories-wildcard事实上并非是CLISP相关的，由于任何其他实现不需要它，因而可以将其定义放在一个读取期条件之后。在这种情况下，由于跟在#+后面的表达式是整个**DEFUN**，因此整个函数定义是否被包含将取决于clisp是否存在于***FEATURES***中。

```
#+clisp
(defun clisp-subdirectories-wildcard (wildcard)
  (make-pathname
   :directory (append (pathname-directory wildcard) (list :wild))
   :name nil
   :type nil
   :defaults wildcard))
```

15.4　测试文件的存在

为了替换**PROBE-FILE**，你可以定义一个称为file-exists-p的函数。它应当接受一个路径名，并在其代表的文件存在时返回一个等价的路径名，否则返回**NIL**。它应当可以接受目录形式或文件形式的目录名，但如果该文件存在并且是一个目录，那么它应当总是返回目录形式的路径名。这将允许你使用file-exists-p和directory-pathname-p来测试任意一个名字是文件名还是目录名。

从理论上讲，file-exists-p和标准函数**PROBE-FILE**非常相似。确实，在一些实现，即SBCL、LispWorks和OpenMCL里，**PROBE-FILE**已经提供了file-exists-p的行为，但并非所有实现的**PROBE-FILE**都具有相同的行为。

Allegro和CMUCL的**PROBE-FILE**函数接近于你想要的行为——接受任何形式的目录名但不会返回目录形式的路径名，而只是简单地返回传给它的参数。幸运的是，如果以目录形式传递给它一个非目录的名字，它会返回**NIL**。因此对于这些实现，为了得到想要的行为，你可以首先以目录形式将名字传给**PROBE-FILE**。如果文件存在并且是一个目录，它将返回目录形式的名字。

www.TopSage.com

大家网 TopSage.com

如果该调用返回NIL，那么你可以用文件形式的名字再试一次。

另一方面，CLISP再一次有其自己的做事方式。它的PROBE-FILE将在传递目录形式的名字时立即报错，无论该名字所代表的文件或目录是否存在。它也会在以文件形式传递一个名字且该名字实际上是一个目录的名字时报错。为了测试一个目录是否存在，CLISP提供了它自己的函数probe-directory（在ext包中）。这几乎就是PROBE-FILE的镜像：它将在传递文件形式的名字，或者目录形式的名字而刚好该名字是一个文件时报错。唯一的区别在于，当命名的目录存在时，它返回T而不是路径名。

就算在CLISP中，你也可以通过将对PROBE-FILE和probe-directory的调用包装在IGNORE-ERRORS中来实现想要的语义。[1]

```
(defun file-exists-p (pathname)
  #+(or sbcl lispworks openmcl)
  (probe-file pathname)

 #+(or allegro cmu)
  (or (probe-file (pathname-as-directory pathname))
      (probe-file pathname))

  #+clisp
  (or (ignore-errors
        (probe-file (pathname-as-file pathname)))
      (ignore-errors
        (let ((directory-form (pathname-as-directory pathname)))
          (when (ext:probe-directory directory-form)
            directory-form))))

  #-(or sbcl cmu lispworks openmcl allegro clisp)
  (error "list-directory not implemented"))
```

CLISP版本的file-exists-p用到的函数pathname-as-file，是前面定义的pathname-as-directory的逆函数，它返回等价于其参数的文件形式的路径名。尽管该函数只有CLISP用到，但它通常很有用，因此我们为所有实现定义它并使其成为该库的一部分。

```
(defun pathname-as-file (name)
  (let ((pathname (pathname name)))
    (when (wild-pathname-p pathname)
      (error "Can't reliably convert wild pathnames."))
    (if (directory-pathname-p name)
      (let* ((directory (pathname-directory pathname))
             (name-and-type (pathname (first (last directory)))))
        (make-pathname
         :directory (butlast directory)
         :name (pathname-name name-and-type)
         :type (pathname-type name-and-type)
```

① 这个方法稍微有一点问题，例如，PROBE-FILE可能因为其他原因报错，这时代码将错误地解释它。不幸的是，CLISP文档并未指定PROBE-FILE和probe-directory可能报错的类型，并且从经验来看，在多数出错情况下它们将会报出simple-file-error。

www.TopSage.com

```
      :defaults pathname))
    pathname)))
```

15.5　遍历目录树

最后，为了完成这个库，你可以实现一个称为walk-directory的函数。与前面定义的那些函数不同，这个函数不需要做任何事情来消除实现间的区别，它只需要用到你已经定义的那些函数。尽管如此，该函数很有用，你将在后续几章里多次用到它。它接受一个目录的名字和一个函数，并在该目录下所有文件的路径名上递归地调用该函数。它还接受两个关键字参数：:directories和:test。当:directories为真时，它将在所有目录的路径名和正规文件上调用该函数。如果有:test参数，它指定另一个函数，在调用主函数之前在每一个路径名上调用该函数，主函数只有当测试参数返回真时才会被调用。:test参数的默认值是一个总是返回真的函数，它是通过调用标准函数CONSTANTLY而生成的。

```
(defun walk-directory (dirname fn &key directories (test (constantly t)))
  (labels
      ((walk (name)
         (cond
           ((directory-pathname-p name)
            (when (and directories (funcall test name))
              (funcall fn name))
            (dolist (x (list-directory name)) (walk x)))
           ((funcall test name) (funcall fn name)))))
    (walk (pathname-as-directory dirname))))
```

现在你有了一个用于处理路径名的有用的函数库。正如我提到的那样，这些函数在后面的章节里将会很有用，尤其是第23章和第27章，在那里你将使用walk-directory在含有垃圾信息和MP3文件的目录树中将会艰难前行，但在我们到达那里之前，我还需要在接下来的两章中谈论一下面向对象。

www.TopSage.com