第14章 文件和文件I/O

Common Lisp提供了一个功能丰富的用于处理文件的函数库。在本章里，我将把重点放在少数基本的文件相关的任务上：读写文件以及列出文件系统中的文件。对这些基本任务，Common Lisp的I/O机制与其他语言相似。Common Lisp为读写数据提供了一个流的抽象和一个称为路径名（pathname）的抽象，它们以一种与操作系统无关的方式来管理文件名。另外，Common Lisp还提供了其他一些只有Lisp才有的功能，比如读写S-表达式。

14.1 读取文件数据

最基本的文件I/O任务是读取文件的内容。可以通过**OPEN**函数获得一个流并从中读取文件的内容。默认情况下，**OPEN**返回一个基于字符的输入流，你可以将它传给许多函数以便读取文本中的一个或多个字符：**READ-CHAR**读取单个字符；**READ-LINE**读取一行文本，去掉行结束字符后作为一个字符串返回；而**READ**读取单一的S-表达式并返回一个Lisp对象。当完成了对流的操作后，你可以使用**CLOSE**函数来关闭它。

OPEN的唯一必要参数是需要读取的文件名。如同你将会在14.6节里看到的那样，Common Lisp提供了许多表示文件名的方式，但最简单的方式是使用一个含有以本地文件命名语法表示的文件名的字符串。因此，假设`/some/file/name.txt`是一个文件，那么可以像下面这样打开它：

```
(open "/some/file/name.txt")
```

你可以把返回对象作为任何读取函数的第一个参数。例如，为了打印文件的第一行，你可以组合使用**OPEN**、**READ-LINE**和**CLOSE**，如下所示：

```
(let ((in (open "/some/file/name.txt")))
  (format t "~a~%" (read-line in))
  (close in))
```

当然，在试图打开和读取一个文件时可能会出现一些错误。该文件可能不存在或者可能在读取时无意中遇到了文件结尾。默认情况下，**OPEN**和`READ-*`系列函数将在出现这些情况时报错。在第19章里，我将讨论如何从这类错误中恢复。不过眼下有一个更轻量级的解决方案：每个这样的函数都可接受参数来修改这些异常情况下的行为。

如果你想打开一个可能不存在的文件而又不想让**OPEN**报错，那么可以使用关键字参数`:if-does-not-exist`来指定不同的行为。三个可能的值是：`:error`，报错（默认

值)；:create，继续进行并创建该文件，然后就像它已经存在那样进行处理；NIL，让它返回NIL来代替一个流。这样，你就可以改变前面的示例来处理文件可能不存在的情况。

```
(let ((in (open "/some/file/name.txt" :if-does-not-exist nil)))
  (when in
    (format t "~a~%" (read-line in))
    (close in)))
```

读取函数，即READ-CHAR、READ-LINE和READ，都接受一个可选的参数，其默认值为真并指定当函数在文件结尾处被调用时是否应该报错。如果该参数为NIL，它们在遇到文件结尾时将返回它们第三个参数的值，默认为NIL。因此，可以像下面这样打印一个文件的所有行：

```
(let ((in (open "/some/file/name.txt" :if-does-not-exist nil)))
  (when in
    (loop for line = (read-line in nil)
         while line do (format t "~a~%" line))
    (close in)))
```

在这三个文本读取函数中，READ是Lisp独有的。这跟提供了REPL中R部分的函数是同一个，它用于读取Lisp源代码。当每次被调用时，它会读取单一的S-表达式，跳过空格和注释，然后返回由S-表达式代表的Lisp对象。例如，假设/some/file/name.txt带有下列内容：

```
(1 2 3)
456
"a string" ; this is a comment
((a b)
 (c d))
```

换句话说，它含有四个S-表达式：一个数字列表、一个数字、一个字符串和一个列表的列表。你可以像下面这样读取这些表达式：

```
CL-USER> (defparameter *s* (open "/some/file/name.txt"))
*S*
CL-USER> (read *s*)
(1 2 3)
CL-USER> (read *s*)
456
CL-USER> (read *s*)
"a string"
CL-USER> (read *s*)
((A B) (C D))
CL-USER> (close *s*)
T
```

如同第3章所述，你可以使用PRINT以“可读的”形式打印Lisp对象。这样，每当你需要在文件中保存一点数据时，PRINT和READ就提供了一个做这件事的简单途径，而无须设计一套数据格式或编写一个解析器。它们甚至可以让你自由地添加注释，如同前面的示例所演示的那样。并且由于S-表达式被设计成是供人编辑的，所以它也是用于诸如配置文件等事务的良好格式。[①]

① 尽管如此，注意Lisp读取器知道如何跳过注释，它会完全跳过它们。这样，如果你使用READ来读取一个含有注释的配置文件，然后使用PRINT来保存对数据的修改，那么你将失去那些注释。

14.2 读取二进制数据

默认情况下，OPEN返回字符流，它根据特定的字符编码方案将底层字节转化成字符。[1]为了读取原始字节，你需要向OPEN传递一个值为'(unsigned-byte 8)的:element-type参数。[2]你可以将得到的流传给READ-BYTE，它将在每次被调用时返回0～255的整数。与字符读取函数一样，READ-BYTE也支持可选的参数以便指定当其被调用在文件结尾时是否应该报错，以及在遇到结尾时返回何值。第24章将构建一个库，它允许使用READ-BYTE来便利地读取结构化的二进制数据。[3]

14.3 批量读取

最后一个读取函数READ-SEQUENCE可同时工作在字符和二进制流上。你传递给它一个序列（通常是一个向量）和一个流，然后它会尝试用来自流的数据填充该序列。它返回序列中第一个没有被填充的元素的索引，或是在完全填充的情况下返回该序列的长度。你也可以传递:start和:end关键字参数来指定一个应当被代替填充的子序列。该序列参数的元素类型必须足以保存带有该流元素类型的元素。由于多数操作系统支持某种形式的块I/O，READ-SEQUENCE通常比重复调用READ-BYTE或READ-CHAR来填充一个序列更加高效。

14.4 文件输出

为了向一个文件中写数据，你需要一个输出流，你可以通过在调用OPEN时使用一个值为:output的:direction关键字参数来获取它。当你打开一个用于输出的文件时，OPEN会假设该文件不该存在并会在文件存在时报错。但你可以使用:if-exists关键字参数来改变该行为。传递值:supersede可以告诉OPEN来替换已有文件。传递:append将导致OPEN打开已有的文件并保证新数据被写到文件结尾处，而:overwrite返回一个从文件开始处开始的流从而覆盖已有数据。而传递NIL将导致OPEN在文件已存在时返回NIL而不是流。一个典型的使用OPEN来输出的例子如下所示：

```
(open "/some/file/name.txt" :direction :output :if-exists :supersede)
```

Common Lisp也提供了几个用于写数据的函数：WRITE-CHAR会向流中写入一个单一字符；WRITE-LINE写一个字符串并紧跟一个换行，其将被输出成用于当前平台的适当行结束字符或字

① 默认情况下，OPEN使用当前操作系统的默认字符编码，但它也接受一个关键字参数:external-format，它可以传递由实现定义的值来指定一个不同的编码。字符流也会转换平台相关的行结束序列到单一字符#\Newline上。

② 类型(unsigned-byte 8)代表8位字节。Common Lisp的“字节”类型并不是固定大小的，由于Lisp曾经运行在不同时期的体系结构上，其字节长度为6～9位。更不用说还有PDP-10计算机，其带有可独立寻址的长度为1～36比特的变长字节域。

③ 一般情况下，一个流要么是字符流要么是二进制流，因此你不能混和调用READ-BYTE、READ-CHAR或者其他基于字符的函数。不过某些实现，例如Allegro，支持所谓的二义流（bivalent stream），其同时支持字符和二进制I/O。

符序列。另一个函数WRITE-STRING写一个字符串而不会添加任何行结束符。有两个不同的函数可以只打印一个换行：TERPRI是“终止打印”（terminate print）的简称，即无条件地打印一个换行字符；FRESH-LINE打印一个换行字符，除非该流已经在一行的开始处。在想要避免由按顺序调用的不同函数所生成的文本输出中的额外换行时，FRESH-LINE很有用。例如，假设一个函数在其生成输出时总是带有一个换行，而另一个函数应当每次从一个新行开始输出。但假设这两个函数被依次调用，而你又不希望在两个输出操作之间产生一个空行，如果你在第二个函数开始处使用FRESH-LINE，那么它的输出将总是从一个新行开始。但如果它刚好在前一个函数之后调用，则它将不会产生一个额外换行。

一些函数会将Lisp数据输出成S-表达式：PRINT打印一个S-表达式，前缀一个换行及一个空格；PRIN1只打印S-表达式；而函数PPRINT会像PRINT和PRIN1那样打印S-表达式，但使用的是美化打印器（pretty printer），它会试图将输出打印得赏心悦目。

但并非所有对象都能以一种READ可理解的形式打印出来。当试图使用PRINT、PRIN1或PPRINT来打印这样一种对象时，变量*PRINT-READABLY*将会予以控制。当它是NIL时，这些函数将以一种导致READ在试图读取时肯定会报错的特殊语法来打印该对象；否则它们将直接报错而不打印该对象。

另一个函数PRINC也会打印Lisp对象，但其工作方式很适合人们使用。例如，PRINC在打印字符串时不带有引号。你可以使用极其灵活但有时略显神秘的FORMAT函数来生成更加复杂的文本输出。我将在第18章里讨论一些关于FORMAT的更重要的细节，它从本质上定义了一种用于产生格式化输出的微型语言。

为了向一个文件中写入二进制数据，你需要在使用OPEN打开文件时带有与读取该文件时相同的:element-type实参，其值为'(unsigned-byte 8)，然后就可以使用WRITE-BYTE向流中写入单独的字节。

批量输出函数WRITE-SEQUENCE可同时接受二进制和字符流，只要序列中的所有元素都是用于该流的适当类型即可，无论其是字符还是字节。和READ-SEQUENCE一样，该函数会比每次输出一个序列元素更为高效。

14.5 关闭文件

任何编写过处理大量文件代码的人都知道，当处理完文件之后，关闭它们是多么重要。因为文件句柄往往是稀缺资源，如果打开一些文件却不关闭它们，你很快会发现不能再打开更多文件了。[9]确保每一个OPEN都有一个匹配的CLOSE可能是非常显而易见的。例如，完全可以像下面这样来组织文件使用代码：

① 有些人可能认为这在诸如Lisp的垃圾收集型语言里不是个问题。在多数Lisp实现中，当一个流变成垃圾之后都将会自动关闭。不过这种行为不能被依赖——问题在于垃圾收集器通常只在内存变少时才运行，它们不认识诸如文件句柄这样的稀缺资源。如果有大量的可用内存，就可以轻易地在垃圾收集器运行之前用光所有的文件句柄。

```
(let ((stream (open "/some/file/name.txt")))
  ;; do stuff with stream
  (close stream))
```

但这一方法还是有两方面的问题。一是容易出现错误——如果忘记使用**CLOSE**，那么代码将在每次运行时泄漏一个文件句柄。而更重要的一点是，该代码并不保证你能够到达**CLOSE**那里。例如，如果**CLOSE**之前的代码含有一个**RETURN**或**RETURN-FROM**，那就会在没有关闭流的情况下离开**LET**语句块。或者如同第19章里将要介绍的，如果任何**CLOSE**之前的代码产生了一个错误，那么控制流可能就会跳出LET语句块而转到一个错误处理器中，然后不再回来关闭那个流。

Common Lisp对于如何确保一直运行特定代码这一问题提供了一个通用的解决方案：特殊操作符**UNWIND-PROTECT**，第20章将予以讨论。不过因为这种打开文件，对产生的流做一些事情，然后再关闭流的模式是如此普遍，Common Lisp就提供了一个构建在**UNWIND-PROTECT**之上的宏**WITH-OPEN-FILE**来封装这一模式。下面是它的基本形式：

```
(with-open-file (stream-var open-argument*)
  body-form*)
```

其中*body-form*中的形式将在*stream-var*被绑定到一个文件流的情况下进行求值，该流由一个以*open-argument*为实参的OPEN调用而打开。**WITH-OPEN-FILE**会确保*stream-var*中的流在**WITH-OPEN-FILE**返回之前被关闭。因此，从一个文件中读取一行的代码应如下所示。

```
(with-open-file (stream "/some/file/name.txt")
  (format t "~a~%" (read-line stream)))
```

为了创建一个新文件，你可以这样写。

```
(with-open-file (stream "/some/file/name.txt" :direction :output)
  (format stream "Some text."))
```

在你所使用的90%～99%的文件I/O中都可能会用到**WITH-OPEN-FILE**——需要使用原始**OPEN**和**CLOSE**调用的唯一情况是，当需要在一个函数中打开一个文件并在函数返回之后仍然保持所产生的流时。在那种情况下必须注意一点，最终要由你自己来关闭这个流，否则你将会泄漏文件描述符并可能最终无法打开更多文件。

14.6 文件名

到目前为止，文件名都是用字符串表示的。但使用字符串作为文件名会将代码捆绑在特定操作系统和文件系统上。而且如果按照一个特定的文件命名方案的规则（比如说，使用“/”来分隔目录）用程序来构造文件名，那么你就会将代码捆绑到特定的文件系统上。

为了避免这种不可移植性，Common Lisp提供了另一种文件名的表示方式：路径名(pathname)对象。路径名以一种结构化的方式来表示文件名，这种方式使得它们易于管理而无须捆绑在特定的文件名语法上。而在以本地语法写成的字符串，即名字字符串（namestring），和路径名之间进行来回转换的责任则被放在了Lisp实现身上。

不幸的是，如同许多被设计用于隐藏本质上不同的底层系统细节的抽象那样，路径名抽象也

引入了它们自己的复杂性。当路径名最初被设计时，通常使用的文件系统集合比今天所使用的更加丰富多彩。这带来的结果是，在你只关心如何表示Unix或Windows文件名时，路径名抽象的某些细微之处就没有什么意义了。不过，一旦你理解了路径名抽象中的哪些部分可以作为路径名发展史中的遗留产物而忽略时，你就会发现它们确实提供了一种管理文件名的便捷方式。[1]

在多数使用文件名的调用场合里，你都可以同时使用名字字符串或是路径名。具体使用哪个在很大程度上取决于该名字的来源。由用户提供的文件名（例如作为参数或是配置文件中的值）通常是名字字符串，因为用户只知道它们所运行在的文件系统，而不关心Lisp如何表示文件名的细节。但通过编程方法产生的文件名是路径名，因为你能可移植地创建它们。一个由OPEN返回的流也代表文件名，也就是那个当初用来打开该流的文件名。这三种类型的文件名被总称为*路径名描述符*（pathname designator）。所有内置的以文件名作为参数的函数都能接受所有这三种路径名描述符。例如，前面章节里所有的用字符串来表示文件名的位置都同样可以传入路径名对象或流。

进化历程

存在于20世纪七八十年代的文件系统的历史多样性很容易被遗忘。Kent Pitman，Common Lisp标准的主要技术编辑之一，有一次在comp.lang.lisp（Message-ID: sfwzo74np6w.fsf@world.std.com）新闻组上描述了如下情形。

> 在Common Lisp的设计完成时期，处于支配地位的文件系统是：TOPS-10、TENEX、TOPS-20、VAX VMS、AT&T Unix、MIT Multics、MIT ITS，更不用说还有许多大型机操作系统了。它们中的一些只支持大写字母，一些是大小写混合的，另一些则是大小写敏感但却能自动作大小写转换（就像Common Lisp）。它们中的一些将目录视为文件，而另一些则不会。一些对于特殊的文件字符带有引用字符，另一些不会。一些带有通配符，而另一些没有。一些在相对路径名中使用:up，另一些不这样做。一些带有可命名的根目录，而另一些没有。还存在没有目录的文件系统，使用非层次目录结构的文件系统，不支持文件类型的文件系统，没有版本的文件系统以及没有设备的文件系统，等等。

如果从任何单一文件系统的观点上观察路径名抽象，那么它看起来显得过于复杂。不过，如果考察两种像Windows和Unix这样相似的文件系统，你可能已经开始注意路径名系统可能帮你抽象掉的一些区别了。例如，Windows文件名含有一个驱动器字母，而Unix文件名却没有。使用这种用来处理过去存在的广泛的文件系统的路径名抽象带来的另一种好处是，它有可能可以处理将来可能存在的文件系统。比如说，如果版本文件系统重新流行起来的话，Common Lisp就已经准备好了。

---

① 路径名系统从某种意义上被认为结构复杂的另一个原因是其对逻辑路径名（logical pathname）的支持。不过，你可以完美地使用路径名系统的其余部分而无需了解任何关于逻辑路径名更多的东西，从而安全地忽略它们。简洁地说，逻辑路径名允许Common Lisp程序含有对路径名的引用而无须命名特定的文件。逻辑路径名随后可以在一个实际的文件系统中被映射到特定的位置上，前提是当程序被安装时，通过定义“逻辑路径名转换”（logical pathname translation）来将匹配特定通配符的文件路径名转化成代表文件系统中文件的路径名，也就是所谓的物理路径名。它们在特定场合下有其自己的用途，但是你可以足够远离且无须担心它们。

14.7 路径名如何表示文件名

路径名是一种使用6个组件来表示文件名的结构化对象：主机（host）、设备（device）、目录（directory）、名称（name）、类型（type）以及版本（version）。这些组件的多数都接受原子值，通常是字符串。只有目录组件有其进一步的结构，含有一个目录名（作为字符串）的列表，其中带有关键字:absolute或:relative作为前缀。但并非所有路径名组件在所有平台上都是必需的——这也是路径名让许多初级Lisp程序员感到无端复杂的原因之一。另一方面，你真的不必担心哪个组件在特定文件系统上是否可被用来表示文件名，除非你需要手工地从头创建一个新路径名对象。相反，通常将通过让具体实现来把一个文件系统相关的名字字符串解析到一个路径名对象，或是通过从一个已有路径名中取得其多数组件来创建新路径名，从而得到路径名对象。

例如，为了将名字字符串转化成路径名，你可以使用**PATHNAME**函数。它接受路径名描述符并返回等价的路径名对象。当该描述符已经是一个路径名时，它就会被简单地返回。当它是一个流时，最初的文件名就会被抽取出然后返回。不过当描述符是一个名字字符串时，它将根据本地文件名语法来解析。作为一个平台中立的文档，语言标准没有指定任何从名字字符串到路径名的特定映射，但是多数实现遵守了与其所在操作系统相同的约定。

在Unix文件系统上，只有目录、名称和类型组件通常会被用到。在Windows上，还有一个组件（通常是设备或主机）保存了驱动器字母。在这些平台上，一个名字字符串在解析时首先会被路径分隔符——在Unix上是一个斜杠而在Windows上是一个斜杠或反斜杠——分拆成基本元素。在Windows上，驱动器字母要么被放置在设备中，要么就是在主机组件中。其他名字元素除最后一个之外都被放置在一个以:absolute或:relative开始的列表中，具体取决于该名字是否（如果有的话，忽略驱动器字母）以一个路径分隔符开始。这个列表将成为路径名的目录组件。如果有的话，最后一个元素将在最右边的点处被分拆开，然后得到的两部分将被放进路径名的名称和类型组件中。[①]

你可以使用函数**PATHNAME-DIRECTORY**、**PATHNAME-NAME**和**PATHNAME-TYPE**来检查一个路径名中的单独组件。

```
(pathname-directory (pathname "/foo/bar/baz.txt")) → (:ABSOLUTE "foo" "bar")
(pathname-name (pathname "/foo/bar/baz.txt"))      → "baz"
(pathname-type (pathname "/foo/bar/baz.txt"))      → "txt"
```

其他三个函数**PATHNAME-HOST**、**PATHNAME-DEVICE**和**PATHNAME-VERSION**允许你访问其他三个路径名组件，尽管它们在Unix上不太可能带有感兴趣的值。在Windows上，**PATHNAME-HOST**和**PATHNAME-DEVICE**两者之一将返回驱动器字母。

① 许多基于Unix的实现特别地对待那些最后一个元素以点开始并且不含有任何其他点的文件名，将整个元素包括点在内放置在名称组件中，并且保留类型组件为**NIL**。

```
(pathname-name (pathname "/foo/.emacs")) → ".emacs"
(pathname-type (pathname "/foo/.emacs")) → NIL
```

尽管如此，并非所有实现都遵守这一约定。一些实现创建一个以空字符串作为名称同时将"emacs"作为类型的路径名。

和许多其他内置对象一样，路径名也有其自身的读取语法：#p后接一个双引号字符串。这允许你打印并且读回含有路径名对象的S-表达式，但由于其语法取决于名字字符串解析算法，这些数据在操作系统之间不一定可移植。

```
(pathname "/foo/bar/baz.txt") → #p"/foo/bar/baz.txt"
```

为了将一个路径名转化回一个名字字符串，例如，为了呈现给用户，你可以使用函数NAMESTRING，其接受一个路径名描述符并返回一个名字字符串。其他两个函数**DIRECTORY-NAMESTRING**和**FILE-NAMESTRING**返回一个部分名字字符串。**DIRECTORY-NAMESTRING**将目录组件的元素组合成一个本地目录名，而**FILE-NAMESTRING**则组合名字和类型组件。[①]

```
(namestring #p"/foo/bar/baz.txt")           → "/foo/bar/baz.txt"
(directory-namestring #p"/foo/bar/baz.txt") → "/foo/bar/"
(file-namestring #p"/foo/bar/baz.txt")      → "baz.txt"
```

14.8　构造新路径名

你可以使用**MAKE-PATHNAME**函数来构造任意路径名。它对每个路径名组件都接受一个关键字参数并返回一个路径名，任何提供了的组件都被填入其中而其余的为**NIL**。[②]

```
(make-pathname
  :directory '(:absolute "foo" "bar")
  :name "baz"
  :type "txt") → #p"/foo/bar/baz.txt"
```

不过，如果你希望程序是可移植的，你不会想要完全用手工生成路径名：就算路径名抽象可以保护你免于使用不可移植的文件名语法，文件名也可能以其他方式不可移植。例如，文件名/home/peter/foo.txt对于Mac OS X来说就不是一个好的文件名，因为在那里/home/被称为/Users/。

不推荐完全用手工生成路径名的另一个原因是，不同的实现使用路径名组件的方式略有差异。例如，前面已经提到过，某些基于Windows的Lisp实现会将驱动器字母保存在设备组件中，而其他一些实现则将它保存在主机组件中。如果你将代码写成这样

```
(make-pathname :device "c" :directory '(:absolute "foo" "bar") :name "baz")
```

那么它在一些实现里将是正确的而在其他实现里则不是。

与其用手工生成路径名，不如基于一个已有的路径名使用**MAKE-PATHNAME**的关键字参数:defaults来构造一个新路径名。你可以为该参数提供一个路径名描述符，它将提供没有被其他参数指定的任何组件的值。例如，下面的表达式创建了一个带有.html扩展名的路径名，同时所有其他组件都与变量input-file中的路径名相同：

```
(make-pathname :type "html" :defaults input-file)
```

① 由**FILE-NAMESTRING**返回的名字在支持版本的文件系统上也含有版本组件。

② 主机组件可能不是默认为**NIL**，但如果不是的话，它将是一个不透明的由实现定义的值。

假设input-file中的值是一个用户提供的名字，这一代码对于操作系统和实现的区别来说是健壮的，无论文件名是否带有驱动器字母或是它被保存在路径名的哪个组件上。[①]

你可以使用相同的技术来创建一个带有不同目录组件的路径名：

```
(make-pathname :directory '(:relative "backups") :defaults input-file)
```

不过，这会创建出一种整个目录组件是相对目录backups/的路径名，而不管input-file是否可能会有任何目录组件。例如：

```
(make-pathname :directory '(:relative "backups")
               :defaults #p"/foo/bar/baz.txt") → #p"backups/baz.txt"
```

但有时你会想要通过合并两个路径名的目录组件来组合两个路径名，其中至少一个带有相对的目录组件。例如，假设有一个诸如#p"foo/bar.html"的相对路径名，你想将它与一个诸如#p"/www/html/"这样的绝对路径名组合起来得到#p"/www/html/foo/bar.html"。在这种情况下，**MAKE-PATHNAME**将无法处理。相反，你需要的是**MERGE-PATHNAMES**。

MERGE-PATHNAMES接受两个路径名并合并它们，用来自第二个路径名的对应值填充第一个路径名中的任何**NIL**组件，这和**MAKE-PATHNAME**使用来自:defaults参数的组件来填充任何未指定的组件非常相似。不过，**MERGE-PATHNAMES**会特别对待目录组件：如果第一个组件名的目录是相对的，那么生成的路径名的目录组件将是第一个路径名的目录相对于第二个路径名的目录。这样：

```
(merge-pathnames #p"foo/bar.html" #p"/www/html/") → #p"/www/html/foo/bar.html"
```

第二个路径名也可以是相对的，在这种情况下得到的路径名也将是相对的：

```
(merge-pathnames #p"foo/bar.html" #p"html/") → #p"html/foo/bar.html"
```

为了反转这一过程以便获得一个相对于特定根目录的文件名，你可以使用函数**ENOUGH-NAMESTRING**，这很有用。

```
(enough-namestring #p"/www/html/foo/bar.html" #p"/www/") → "html/foo/bar.html"
```

随后可以组合**ENOUGH-NAMESTRING**和**MERGE-PATHNAMES**来创建一个表达相同名字但却在不同根目录中的路径名。

```
(merge-pathnames
  (enough-namestring #p"/www/html/foo/bar/baz.html" #p"/www/")
  #p"/www-backups/") → #p"/www-backups/html/foo/bar/baz.html"
```

MERGE-PATHNAMES也被用来实际访问文件系统中标准函数内部用于填充不完全的路径名的文件。例如，假设有一个只有名称和类型的路径名：

① 对于完全最大化的可移植性，你真的应该写成这样：

```
(make-pathname :type "html" :version :newest :defaults input-file)
```

没有:version参数的话，在一个带有内置版本支持的文件系统上，输出的路径名将继承来自输入文件的版本号，这可能不是正确的行为——如果输入文件已经被保存了许多次，它将带有一个比生成的HTML文件大的多的版本号。*在没有文件版本的实现上*，:version参数会被忽略。如果你比较在意可移植性，最好加上它。

```
(make-pathname :name "foo" :type "txt") → #p"foo.txt"
```

如果想用这个路径名作为**OPEN**的一个参数，那么缺失的组件（诸如目录）就必须在Lisp可以将路径名转化成一个实际文件名之前被填充进去。Common Lisp将通过合并给定路径名与变量***DEFAULT-PATHNAME-DEFAULTS***中的值来获得缺失组件的值。该变量的初始值由具体实现决定，通常是一个路径名，其目录组件表示Lisp启动时所在的目录，如果需要，主机和设备组件也带有适当的值。如果只有一个参数被调用，**MERGE-PATHNAMES**将把该参数与***DEFAULT-PATHNAME-DEFAULTS***的值进行合并。例如，如果***DEFAULT-PATHNAME-DEFAULTS***是`#p"/home/peter/"`，那么将得到下面的结果：

```
(merge-pathnames #p"foo.txt") → #p"/home/peter/foo.txt"
```

14.9　目录名的两种表示方法

当处理命名目录的路径名时，你需要注意一点。路径名将目录和名称组件区分开，但Unix和Windows却将目录视为另一种类型的文件。这样，在这些系统里，每一个目录都有两种不同的路径名表示方法。

一种表示方法，我将它称为文件形式（file form），将目录当成像其他任何文件一样来对待，将名字字符串中的最后一个元素放在名称和类型组件中。另一种表示方法，目录形式（directory form）将名字中的所有元素都放在目录组件中，而留下名称和类型组件为**NIL**。如果`/foo/bar/`是一个目录，那么下面两个路径名都可以命名它：

```
(make-pathname :directory '(:absolute "foo") :name "bar") ; file form
(make-pathname :directory '(:absolute "foo" "bar"))       ; directory form
```

用**MAKE-PATHNAME**创建路径名时，你可以控制得到的形式，但需要在处理名字字符串时多加小心。当前所有实现都创建文件形式的路径名，除非名字字符串以一个路径分隔符结尾。但你不能依赖于用户提供的名字字符串必须是以一种或另一种形式。例如，假设提示用户输入一个保存文件的目录而他们输入了`"/home/peter"`。如果你将该值作为**MAKE-PATHNAME**的`:defaults`参数，像这样

```
(make-pathname :name "foo" :type "txt" :defaults user-supplied-name)
```

来传递，那么最后你将把文件保存成`/home/foo.txt`而不是你想要的`/home/peter/foo.txt`，因为当`user-supplied-name`被转化成一个路径名时，名字字符串中的`"peter"`将被放在名称组件中。在下一章我将讨论的路径名可移植库中，你将编写一个称为`pathname-as-directory`的函数，它将一个路径名转化成目录形式。使用该函数，你可以放心地在用户给出的目录里保存文件。

```
(make-pathname
  :name "foo" :type "txt" :defaults (pathname-as-directory user-supplied-name))
```

14.10 与文件系统交互

通常，与文件系统的多数交互可能是用OPEN打开文件来读写。你偶尔也需要测试一个文件是否存在，列出一个目录的内容，删除和重命名文件，创建目录以及获取一个文件的信息，诸如谁拥有它、它在何时被最后修改以及它的长度。这就是由路径名抽象所带来的通用性开始造成痛苦的地方：因为语言标准并没有指定那些与文件系统交互的函数是如何映射到任何特定的文件系统上的，从而给实现者们留下了充分的余地。

这就是说，多数与文件系统进行交互的函数仍然是相当直接的。我将在这里讨论标准函数，并且指出其中哪些是在实现之间存在不可移植性的。在下一章里，你将开发一个路径名可移植库来消除那些不可移植因素中的一部分。

为了测试一个对应于某个路径名描述符（路径名、名字字符串或文件流）的文件是否存在于文件系统中，你可以使用函数PROBE-FILE。如果由路径名描述符命名的文件存在，那么PROBE-FILE将返回该文件的*真实名称*（truename），一个进行了诸如解析符号链接这类文件系统层面转换的路径名。否则它返回NIL。不过，并非所有实现都支持使用该函数来测试一个目录是否存在。同样，Common Lisp也不支持用一种可移植的方式来测试一个给定文件是否作为一个正规文件或目录而存在。在下一章里，你将把PROBE-FILE包装在一个新函数file-exists-p中，它不但可以测试目录是否存在，还可以告诉你一个给定的名字究竟是文件名还是目录名。

类似地，用于列出文件系统中文件的标准函数DIRECTORY对于简单的情形工作得很好，但实现之间的区别却使得它难以可移植地使用。在下一章里，你将定义一个list-directory函数来消除这些区别。

DELETE-FILE和RENAME-FILE的功能恰如其名。DELETE-FILE接受一个路径名描述符并删除所命名的文件，当其成功时返回真。否则它产生一个FILE-ERROR报错。[1]

RENAME-FILE接受两个路径名描述符，并将第一个名字命名的文件重命名为第二个名字。

你可以使用函数ENSURE-DIRECTORIES-EXIST来创建目录。它接受一个路径名描述符并确保目录组件中的所有元素存在并且是目录，如果必要的话它会创建它们。它返回被传递的路径名，这使得它易于内联使用。

```
(with-open-file (out (ensure-directories-exist name) :direction :output)
  ...
  )
```

注意如果你传给ENSURE-DIRECTORIES-EXIST一个目录名，它应该是目录形式的，否则目录的最后一级子目录将不会被创建。

函数FILE-WRITE-DATE和FILE-AUTHOR都接受一个路径名描述符。FILE-WRITE-DATE返回文件上次被写入的时间，表示形式是自从格林尼治标准时间（GMT）1900年1月1日午夜起的

① 更多关于错误处理的内容请参见第19章。

秒数，而FILE-AUTHOR在Unix和Windows上返回该文件的拥有者。[1]

为了知道一个文件的长度，你可以使用函数FILE-LENGTH。出于历史原因，FILE-LENGTH接受一个流而不是一个路径名作为参数。在理论上，这允许FILE-LENGTH返回在该流的元素类型意义下的长度。尽管如此，由于在当今大多数操作系统上关于一个文件的长度唯一可以得到的信息（除了实际读取整个文件来测量它以外）只有以字节为单位的长度，这也是多数实现所返回的，甚至当FILE-LENGTH被传递了一个字符流时情况也是如此。不过，标准并没有强制要求这一行为，因此为了得到可预测的结果，获得一个文件长度的最佳方式是使用一个二进制流。[2]

```
(with-open-file (in filename :element-type '(unsigned-byte 8))
  (file-length in))
```

一个同样接受打开的文件流作为参数的相关函数是FILE-POSITION。当只用一个流来调用它时，该函数返回文件中的当前位置，即已经被读取或写入该流的元素的数量。当以两个参数（流和位置描述符）来调用该函数时，它将该流的位置设置到所描述的位置上。这个位置描述符必须是关键字:start、:end或者非负的整数。两个关键字可以将流的位置设置到文件的开始或结尾处，而一个整数将使流的位置移动到文件中指定的位置上。对于二进制流来说，这个位置就是文件中的字节偏移量。尽管如此，因为字符编码因素的存在，对于字符流来说事情变得有一点儿复杂。当你需要在一个文本数据的文件中做跳转时，最可靠的方法就是只为两参数版本的FILE-POSITION的第二个参数传递一个由单参数版本FILE-POSITION同样的流参数下曾经返回的一个值。

14.11　其他I/O类型

除了文件流以外，Common Lisp还支持其他类型的流，它们也可被用于各种读、写和打印I/O函数。例如，你可以使用STRING-STREAM从一个字符串中读取或写入数据，你可以使用函数MAKE-STRING-INPUT-STREAM和MAKE-STRING-OUTPUT-STREAM来创建STRING-STREAM。

MAKE-STRING-INPUT-STREAM接受一个字符串以及可选的开始和结尾指示符来鉴定字符串中数据应被读取的区域，然后返回一个可被传递到任何诸如READ-CHAR、READ-LINE或READ这

① 对于需要访问特定操作系统或文件系统上其他文件属性的应用来说，第三方库提供了对底层C系统调用的绑定。http://common-lisp.net/project/osicat/上的Osicat库提供了一个构建在Universal Foreign Function Interface（UFFI）之上的简单API，该库应当可以运行在POSIX操作系统的多数Common Lisp上。

② 就算在你没有使用多字节字符编码时，一个文件中字节和字符的数量也可能是不同的。因为字符流也会将平台相关的行结束符转化成单一的#\Newline字符，在Windows上（其中使用CRLF作为行结束符），字符的数量通常小于字节的数量。如果你真想知道文件中字符的数量，你不得不亲自数一下并书写类似下面这样的代码：

```
(with-open-file (in filename)
  (loop while (read-char in nil) count t))
```

或者可能是如下更高效的代码：

```
(with-open-file (in filename)
  (let ((scratch (make-string 4096)))
    (loop for read = (read-sequence scratch in)
          while (plusp read) sum read)))
```

些基于字符的输入函数中的字符流。例如，如果你有一个含有Common Lisp语法的字面浮点数的字符串，那么你可以像下面这样将它转化成一个浮点数：

```
(let ((s (make-string-input-stream "1.23")))
  (unwind-protect (read s)
    (close s)))
```

类似地，**MAKE-STRING-OUTPUT-STREAM**创建一个流，其可被用于**FORMAT**、**PRINT**、**WRITE-CHAR**以及**WRITE-LINE**等。它不接受参数。无论你写了什么，字符串输出流都将被累积到字符串中，你随后可以通过函数**GET-OUTPUT-STREAM-STRING**来获取该字符串。每次当你调用**GET-OUTPUT-STREAM-STRING**时，该流的内部字符串会被清空，因此就可以重用一个已有的字符串输出流。

不过你将很少直接使用这些函数，因为宏**WITH-INPUT-FROM-STRING**和**WITH-OUTPUT-TO-STRING**提供了一个更加便利的接口。**WITH-INPUT-FROM-STRING**和**WITH-OPEN-FILE**相似，它从给定字符串中创建字符串输入流，并在该流绑定到你提供的变量的情况下执行它的主体形式。例如，与其使用**LET**形式并带有显式的**UNWIND-PROTECT**，你可以这样来写。

```
(with-input-from-string (s "1.23")
  (read s))
```

宏**WITH-OUTPUT-TO-STRING**与之相似：它把新创建的字符串输出流绑定到你所命名的变量上，然后执行它的主体。在所有主体形式都被执行以后，**WITH-OUTPUT-TO-STRING**返回由**GET-OUTPUT-STREAM-STRING**返回的值。

```
CL-USER> (with-output-to-string (out)
           (format out "hello, world ")
           (format out "~s" (list 1 2 3)))
"hello, world (1 2 3)"
```

语言标准中定义的其他流提供了多种形式的流拼接技术，它允许你以几乎任何配置将流拼接在一起。**BROADCAST-STREAM**是一个输出流，它将向其写入的任何数据发送到一组输出流上，这些流是它的构造函数**MAKE-BROADCAST-STREAM**的参数。[①]与之相反的，**CONCATENATED-STREAM**是一个输入流，它从一组输入流中接收其输入，在每个流的结尾处它从一个流移动到另一个。你可以使用函数**MAKE-CONCATENATED-STREAM**来构造**CONCATENATED- STREAM**，其接受任何数量的输入流作为参数。

两种可以将流以多种方式拼接在一起的双向流是**TWO-WAY-STREAM**和**ECHO-STREAM**。它们的构造函数**MAKE-TWO-WAY-STREAM**和**MAKE-ECHO-STREAM**都接受两个参数，一个输入流和一个输出流，并返回一个适当类型的可同时用于输入和输出函数的流。

在**TWO-WAY-STREAM**中，你所做的每一次读取都会返回从底层输入流中读取的数据，而每次写入将把数据发送到底层的输出流上。除了所有从底层输入流中读取的数据也被回显到输出流中之外，**ECHO-STREAM**本质上以相同的方式工作。这样，**ECHO-STREAM**中的输出流将含有会话双方的一个副本。

① 通过不带参数地调用它，**MAKE-BROADCAST-STREAM**可以生成一个数据黑洞。

使用这五种流，你可以构造出几乎任何你想要的流拼接拓扑结构。

最后，尽管Common Lisp标准并没有涉及有关网络API的内容，但多数实现都支持socket编程并且通常将socket实现成另一种类型的流，因此你可以使用正规I/O函数来操作它们。[①]

现在，你已准备好开始构建一个库来消除不同Common Lisp实现在基本路径名函数行为上的一些区别了。

① Common Lisp的标准I/O机制最缺失的是一种允许用户定义新的流类（stream class）的方式。不过，存在两种用户自定义流的事实标准。在Common Lisp标准化期间，德州仪器的David Gray编写了一份API草案，其中允许用户定义新的流类。然而不幸的是，当时没有时间解决由这份草案产生的所有问题，从而将其包含到语言标准中。尽管如此，许多实现都支持某种形式的所谓Gray Streams，它们的API都是基于Gray的草案。另一种更新的API称为Simple Streams，它由Franz开发并包括在Allegro Common Lisp中。它被设计用来改进用户自定义流相对于Gray Streams的性能，并且已经被某些开源Common Lisp实现所采用。