第七章：I/O

就算不是全部，绝大多数的程序员显然还是致力于从外界收集数据，处理这些数据，然后把结果传回外界。也就是说，关键就是输入输出。

Haskell的I/O系统是很强大和富有表现力的。它易于使用，也很有必要去理解。Haskell严格地把纯代码从那些会让外部世界发生事情的代码中分隔开。就是说，它给纯代码提供了完全的副作用隔离。除了帮助程序员推断他们自己代码的正确性，它还使编译器可以自动采取优化和并行化成为可能。

我们将用简单标准的I/O来开始这一章。然后我们要讨论下一些更强大的选项，以及提供更多I/O是怎么适应纯的，惰性的，函数式的Haskell世界的细节。

Haskell经典I/O

让我们开始使用Haskell的I/O吧。先来看一个程序，它看起来很像在C或者Perl等其他语言的I/O。

```
-- file: ch07/basicio.hs
main = do
    putStrLn "Greetings!  What is your name?"
    inpStr <- getLine
    putStrLn $ "Welcome to Haskell, " ++ inpStr ++ "!"
```

你可以编译这个程序，变成一个单独的可执行文件，然后用 **runghc** 运行它，或者从 **ghci** 调用 `main` 。这里有一个使用**runghc**的例子：

```
$ runghc basicio.hs
Greetings!  What is your name?
John
Welcome to Haskell, John!
```

这相单简单，结果很明显。你可以看到 `putStrLn` 输出一个 `string` ，后面跟了一个换行符。 `getLine` 从标准输入读取一行。 `<-` 语法对于你可能比较新。简单来看，它绑定一个I/O动作的结果到一个名字。我们用简单的列表串联运算符 `++` 来联合输入字符串和我们自己的文本。

让我们来看一下 `putStrLn` 和 `getLine` 的类型。你可以在库参考手册里看到这些信息，或者直接问 `ghci` ：

```
ghci> :type putStrLn
putStrLn :: String -> IO ()
ghci> :type getLine
getLine :: IO String
```

注意，这些类型在他们的返回值里面都有IO。现在关键的是，你要从这里知道他们可能有副作用，或者他们用相同的参数调用可能返回不同的值，或者两者都有。 `putStrLn` 的类型看起来像一个函数，它接受一个 `String` 类型的参数，并返回 `IO ()` 类型的值。可是 `IO ()` 是什么呢?

`IO something` 类型的所有东西都是一个IO动作，你可以保存它但是什么都不会发生。我可以说 `writefoo = putStrLn "foo"` 并且现在什么都不发生。但是如果我过一会在另一个I/O动作中间使用 `writefoo` ， `writefoo` 动作将会在它的父动作被执行的时候执行 – I/O动作可以粘合在一起来形成更大的I/O动作。 `()` 是一个空的元组（读作“unit”），表明从 `putStrLn` 没有返回值。这和Java或C里面的 `void` 类似。

> **Tip**
>
> I/O动作可以被创建，赋值和传递到任何地方，但是它们只能在另一个I/O动作里面被执行。

我们在 **ghci** 下看下这句代码：

```
ghci> let writefoo = putStrLn "foo"
ghci> writefoo
foo
```

v: latest

在这个例子中，输出 `foo` 不是 `putStrLn` 的返回值，而是它的副作用，把 `foo` 写到终端上。

还有另一件事要注意， 实际上是 **ghci** 执行的 `writefoo` 。意思是，如果给 **ghci** 一个I/O动作，它将会在那个地方帮你执行它。

> **Note**
>
> 什么是I/O动作？ * 类型是 `IO t` * 是Haskell的头等值，并且和Haskell的类型系统无缝结合。 * 在运行（perform）的时候产生作用，而不是在估值（evaluate）的时候。 * 任何表达式都会产生一个动作作为它的值，但是这个动作直到在另一个I/O动作里面被执行的时候才会运行。 * 运行（执行）一个 `IO t` 类型的动作可能运行I/O，并且最终交付一个类型 `t` 的结果。

`getLine` 的类型可能看起来比较陌生。它看起来像一个值，而不像一个函数。但实际上，有一种看它的方法： `getLine` 保存了一个I/O动作。当这个动作运行了你会得到一个 `String` 。 `<-` 运算符是用来从运行I/O动作中抽出结果，并且保存到一个变量中。

`main` 自己就是一个I/O动作，类型是 `IO ()` 。你可以在其他I/O动作中只是运行I/O动作。Haskell程序中的所有I/O动作都是由从 `main` 的顶部开始驱动的， `main` 是每一个Haskell程序开始执行的地方。然后，要说的是给Haskell中副作用提供隔离的机制是：你在I/O动作中运行I/O，并且在那儿调用纯的（非I/O）函数。大部分Haskell代码是纯的，I/O动作运行I/O并且调用存代码。

`do` 是用来定义一串动作的方便方法。你马上就会看到，还有其他方法可以用来定义。当你用这种方式来使用 `do` 的时候，缩进很重要，确保你的动作正确地对齐了。

只有当你有多余一个动作需要运行的时候才要用到 `do` 。 `do` 代码块的值是最后一个动作执行的结果。想要看 `do` 语法的完整介绍，可以看 **`do代码块提取`_** .

我们来考虑一个在I/O动作中调用存代码的一个例子：

```
-- file: ch07/callingpure.hs
name2reply :: String -> String
name2reply name =
    "Pleased to meet you, " ++ name ++ ".\n" ++
    "Your name contains " ++ charcount ++ " characters."
    where charcount = show (length name)

main :: IO ()
main = do
       putStrLn "Greetings once again.  What is your name?"
       inpStr <- getLine
       let outStr = name2reply inpStr
       putStrLn outStr
```

注意例子中的 `name2replay` 函数。这是一个Haskell的一个常规函数，它遵守所有我们告诉过你的规则：给它相同的输入，它总是返回相同的结果，没有副作用，并且以惰性方式运行。它用了其他Haskell函数： `(++)` ， `show` 和 `length` 。

往下看到 `main` ，我们绑定 `name2replay inpStr` 的结果到 `outStr` 。当你在用 `do` 代码块的时候，你用 `<-` 去得到I/O动作的结果，用 `let` 得到存代码的结果。 当你在 `do` 代码块中使用 `let` 声明的时候，不要在后面放上 `in` 。

你可以看到这里是怎么从键盘读取这人的名字的。然后，数据被传到一个纯函数，接着它的结果被打印出来。实际上， `main` 的最后两行可以被替换成 `putStrLn (name2reply inpStr)` 。所以， `main` 有副作用（比如它在终端上显示东西） ， `name2replay` 没有副作用，也不能有副作用。因为 `name2replay` 是一个纯函数，不是一个动作。

我们在 **ghci** 上检查一下：

```
ghci> :load callingpure.hs
[1 of 1] Compiling Main             ( callingpure.hs, interpreted )
Ok, modules loaded: Main.
ghci> name2reply "John"
"Pleased to meet you, John.\nYour name contains 4 characters."
ghci> putStrLn (name2reply "John")
Pleased to meet you, John.
Your name contains 4 characters.
```

字符串里面的 `\n` 是换行符， 它让终端在输出中开始新的一行。在 **ghci** 直接调用 `name2replay "John"` 会字面上显示 `\n` ， v: latest 来显示返回值。但是使用 `putStrLn` 来发送到终端的话，终端会把 `\n` 解释成开始新的一行。

如果你就在 **ghci** 提示符那打上 `main` ，你觉得会发生什么？来试一下吧。

看完这几个例子程序之后，你可能会好奇Haskell是不是真正的命令式语言呢，而不是纯的，惰性的，函数式的。这些例子里的一些看起来是按照顺序的一连串的操作。这里面还有很多东西，我们会在这一章的 **`Haskell是不是真正的命令式的呢？`_** 和 **惰性I/O** 章节来讨论这个问题。

Pure vs. I/O

这里有一个比较的表格，用来帮助理解存代码和I/O之间的区别。 当我们说起存代码的时候，我们是在说Haskell函数在输入相同的时候总是返回相同结果，并且没有副作用。在Haskell里面只有I/O动作的执行违反这些规则。

表格7.1. Pure vs. Impure

Pure	Impure
输入相同时总是产生相同结果	相同的参数可能产生不同的结果
从不会有副作用	可能有副作用
从不修改状态	可能修改程序、系统或者世界的全局状态

为什么纯不纯很重要?

在这一节中，我们已经讨论了Haskell是怎么在存代码和I/O动作之间做了很明确的区分。很多语言没有这种区分。在C或者Java这样的语言中，编译器不能保证一个函数对于同样的参数总是返回同样的结果，或者保证函数没有副作用。要知道一个函数有没有副作用只有一个办法，就是去读它的文档，并且希望文档说的准确。

程序中的很多错误都是由意料之外的副作用造成的。函数在某些情况下对于相同参数可能返回不同的结果，还有更多错误是由于误解了这些情况而造成的。 多线程和其他形式的并行化变得越来越普遍， 管理全局副作用变得越来越困难。

Haskell隔离副作用到I/O动作中的方法提供了一个明确的界限。你总是可以知道系统中的那一部分可能修改状态哪一部分不会。你总是可以确定程序中纯的部分不会有意想不到的结果。这样就帮助你思考程序，也帮助编译器思考程序。比如最新版本的 **ghc** 可以自动给你代码纯的部分提供一定程度的并行化 – 一个计算的神圣目标。

对于这个主题，你可以在 *_惰性I/O副作用* 一节看更多的讨论。

使用文件和句柄（Handle）

到目前为止，我们已经看了在计算机的终端里怎么和用户交互。当然，你经常会需要去操作某个特定文件，这个也很简单。

Haskell位I/O定义了一些基本函数，其中很多和你在其他语言里面见到的类似。 `System.IO` 的参考手册为这些函数提供了很好的概要。你会用到这里面某个我们在这里没有提及的某个函数。

通常开始的时候你会用到 `openFile` ，这个函数给你一个文件句柄，这个句柄用来对这个文件做特定的操作。Haskell提供了像 `hPutStrLn` 这样的函数，它用起来和 `putStrLn` 很像，但是多一个参数（句柄），指定操作哪个文件。当操作完成之后，需要用 `hClose` 来关闭这个句柄 。这些函数都是定义在 `System.IO` 中的，所以当你操作文件的时候你要引入这个模块。几乎每一个非“h”的函数都有一个对应的“h”函数，比如，`print` 打印到显示器，有一个对应的 `hPrint` 打印到文件。

我们用一种命令式的方式来开始读写文件。这有点像一个其他语言中 `while` 循环，这在Haskell中不是最好的方法。接着我们会看几个更加Haskell风格的例子。

```
-- file: ch07/toupper-imp.hs
import System.IO
import Data.Char(toUpper)

main :: IO ()
main = do
    inh <- openFile "input.txt" ReadMode
    outh <- openFile "output.txt" WriteMode
    mainloop inh outh
    hClose inh
    hClose outh

mainloop :: Handle -> Handle -> IO ()
```

v: latest

```
mainloop inh outh =
    do ineof <- hIsEOF inh
       if ineof
       then return ()
       else do inpStr <- hGetLine inh
               hPutStrLn outh (map toUpper inpStr)
               mainloop inh outh
```

像每一个Haskell程序一样，程序在 `main` 那里开始执行。两个文件被打开： `input.txt` 被打开用来读，还有一个 `output.txt` 被打开用来写。然后我们调用 `mainloop` 来处理这个文件。

`mainloop` 开始的时候检查看看我们是否在输入文件的结尾（EOF）。如果不是，我们从输入文件读取一行，把这一行转成大写，再把它写到输出文件。然后我们递归调用 `mainloop` 继续处理这个文件。

注意那个 `return` 调用。这个和C或者Python中的 `return` 不一样。在那些语言中， `return` 用来立即退出当前函数的执行，并且给调用者返回一个值。在Haskell中， `return` 是和 `<-` 相反。也就是说， `return` 接受一个纯的值，把它包装进IO。因为每个I/O动作必须返回某个 `IO` 类型，如果你的结果来自纯的计算，你必须用 `return` 把它包装进IO。举一个例子，如果 `7` 是一个 `Int` ，然后 `return 7` 会创建一个动作，里面保存了一个 `IO Int` 类型的值。在执行的时候，这个动作将会产生结果 `7` 。关于 `return` 的更多细节，可以参见 **Return的本色** 一节。

我们来尝试运行这个程序。我们已经有一个像这样的名字叫 `input.txt` 的文件：

```
This is ch08/input.txt

Test Input
I like Haskell
Haskell is great
I/O is fun

123456789
```

现在，你可以执行 `runghc toupper-imp.hs`，你会在你的目录里找到 `output.txt` 。它看起来应该是这样：

```
THIS IS CH08/INPUT.TXT

TEST INPUT
I LIKE HASKELL
HASKELL IS GREAT
I/O IS FUN

123456789
```

关于 openFile 的更多信息

我们用 **ghci** 来检查 `openFifle` 的类型：

```
ghci> :module System.IO
ghci> :type openFile
openFile :: FilePath -> IOMode -> IO Handle
```

`FilePath` 就是 `String` 的另一个名字。它在I/O函数的类型中使用，用来阐明那个参数是用来表示文件名的，而不是其他通常的数据。

`IOMode` 指定文件是怎么被管理的， `IOMode` 的可能值在表格7.2中列出来了。

表格7.2. IOMode 可能值

IOMode	可读	可写	开始位置	备注
ReadMode	是	否	文件开头	文件必须存在
WriteMode	否	是	文件开头	如果存在，文件会被截断（完全清空）
ReadWriteMode	是	是	文件开头	如果不存在会新建文件，如果存在不会损害原来的数据
AppendMode	否	是	文件结尾	如果不存在会新建文件，如果存在不会损害原来的数据

我们在这一章里大多数是操作文本文件，二进制文件同样可以在Haskell里使用。如果你在操作一个二进制文件，你要用 `openBinaryFile` 替代 `openFile` 。你当做二进制文件打开，而不是当做文本文件打开的话，像Windows这样的操作系统会用不同的方式来处理文件。在

Linux这类操作系统中， `openFile` 和 `openBinaryFile` 执行相同的操作。不过为了移植性，当你处理二进制数据的时候总是用 `openBinaryFile` 还是明智的。

关闭句柄

你已经看到 `hClose` 用来关闭文件句柄 。我们花点时间思考下为什么这个很重要。

就和你将在 **缓冲区（Buffering）** 一节看到的一样，Haskell为文件维护内部缓冲区，这提供了一个重要的性能提升。然而，也就是说，直到你在一个打开来写的文件上调用 `hClose` ，你的数据不会被清理出操作系统。

确保 `hClose` 的另一个理由是，打开的文件会占用系统资源。如果你的程序运行很长一段时间，并且打开了很多文件，但是没有关闭他们，你的程序很有可能因为资源耗尽而崩溃。所有这些Haskell和其他语言没有什么不同。

当一个程序退出的时候，Haskell通常会小心地关闭所以还打开着的文件。然而在一些情况下Haskell可能不会帮你做这些。所以再一次强调，最好任何时候由你负责调用 `hClose` 。

Haskell给你提供了一些工具，不管出现什么错误，用来简单地确保这些工作。你可以阅读在 **扩展例子：函数式I/O和临时文件** 一节的 `finally` 和 **`获取-使用-回收 周期`_** 一节的 `bracket` 。

Seek and Tell

当从一个对应硬盘上某个文件句柄上读写的时候，操作系统维护了一个当前硬盘位置的内部记录。每次你做另一次读的时候，操作系统返回下一个从当前位置开始的数据块，并且增加这个位置，反映出你正在读的数据。

你可以用 `hTell` 来找出你文件中的当前位置。当文件刚新建的时候，文件是空的，这个位置为0。在你写入5个字节之后，位置会变成5，诸如此类。 `hTell` 接受一个 `Handle` 并返回一个带有位置的 `IO Integer` 。

`hTell` 的伙伴是 `hSeek` 。 `hSeek` 让你可以改变文件位置，它有3个参数：一个 `Handle` ， 一个 `seekMode` ，还有一个位置。

`SeekMode` 可以是三个不同值中的一个，这个值指定怎么去解析这个给的位置。 `AbsoluteSeek` 表示这个位置是在文件中的精确位置，这个和 `hTell` 给你的是同样的信息。 `RelativeSeek` 表示从当前位置开始寻找，一个正数要求在文件中向前推进，一个负数要求向后倒退。最后，`SeekFromEnd` 会寻找文件结尾之前特定数目的字节。 `hSeek handle SeekFromEnd 0` 把你带到文件结尾。举一个 `hSeek` 的例子，参考 **扩展例子：函数式I/O和临时文件** 一节。

不是所有句柄都是可以定位的。一个句柄通常对应于一个文件，但是它也可以对应其他东西，比如网络连接，磁带机或者终端。你可以用 `hIsSeekable` 去看给定的句柄是不是可定位的。

标准输入，输出和错误

先前我们指出对于每一个非“h”函数通常有一个对应的“h”函数用在句柄上的。实际上，非“h”的函数就是他们的“h”函数的一个快捷方式。

在 `System.IO` 里有3个预定义的句柄，这些句柄总是可用的。他们是 `stdin` ，对应标准输入； `stdout` ，对应标准输出；和 `stderr` 对应标准错误。标准输入一般对应键盘，标准输出对应显示器，标准错误一般输出到显示器。

像 `getLine` 的这些函数可以简单地这样定义：

```
getLine = hGetLine stdin
putStrLn = hPutStrLn stdout
print = hPrint stdout
```

Tip

我们这里使用了局部应用。如果不明白，可以参考 **`局部函数应用和柯里化`_**

之前我们告诉你这3个标准文件句柄一般对应什么。那是因为一些操作系统可以让你重定向这个文件句柄到不同的地方-文件，设备，甚至是其他程序。这个功能在POSIX（Linux，BSD，Mac）操作系统Shell编程中广泛使用，在Windows中也能使用。

使用标准输入输出经常是很有用的，这让你和终端前的用户交互。它也能让你操作输入输出文件，或者甚至让你的代码和其他程序组合在一起。

举一个例子，我们可以像这样在前面提供标准输入给 `callingpure.hs` ：

```
$ echo John|runghc callingpure.hs
Greetings once again.  What is your name?
Pleased to meet you, John.
Your name contains 4 characters.
```

当 callingpure.hs 运行的时候，它不用等待键盘的输入，而是从 echo 程序接收 John 。注意输出也没有把 John 这个词放在一个分开的行，这和用键盘运行程序一样。终端一般回显所有你输入的东西给你，但这是一个技术上的输入，不会包含在输出流中。

删除和重命名文件

这一章到目前为止，我们已经讨论了文件的内容。现在让我们说一点文件自己的东西。 System.Directory 提供了两个你可能觉得有用的函数。 removeFile 接受一个参数，一个文件名，然后删除那个文件。 renameFile 接受两个文件名：第一个是老的文件名，第二个是新的文件名。如果新的文件名在另外一个目录中，你也可以把它想象成移动文件。在调用 renameFile 之前老的文件必须存在。如果新的文件已经存在了，它在重命名之前会被删除掉。

像很多其他接受文件名的函数一样，如果老的文件名不存在， renameFile 会引发一个异常。更多关于异常处理的信息你可以在 **`第十九章，错误处理`_** 中找到。

在 System.Directory 中有很多其他函数，用来创建和删除目录，查找目录中文件列表，和测试文件是否存在。它们在 **`目录和文件信息`_** 一节中讨论。

临时文件

程序员频繁需要用到临时文件。临时文件可能用来存储大量需要计算的数据，其他程序要使用的数据，或者很多其他的用法。

当你想一个办法来手动打开同名的多个文件，安全地做到这一点的细节在各个平台上都不相同。Haskell提供了一个方便的函数叫做 openTempFile （还有一个对应的 openBinaryTempFile ）来为你处理这个难点。

openTempFile 接受两个参数：创建文件所在的目录，和一个命名文件的“模板”。这个目录可以简单是“.”，表示当前目录。或者你可以用 System.Directory.getTemporaryDirectory 去找指定机器上存放临时文件最好的地方。这个模板用做文件名的基础，它会添加一些随机的字符来保证文件名是唯一的，从实际上保证被操作的文件具有独一无二的文件名。

openTempFile 返回类型是 IO (FilePath, Handle) 。元组的第一部分是创建的文件的名字，第二部分是用 ReadWriteMode 打开那个文件的一个句柄 。当你处理完这个文件，你要 hClose 它并且调用 removeFile 删除它。看下面的例子中一个样本函数的使用。

扩展例子：函数式I/O和临时文件

这里有一个大一点的例子，它把很多这一章的还有前面几章的概念放在一起，还包含了一些没有介绍过的概念。看一下这个程序，看你是否能知道它是干什么的，是怎么做的。

```
-- file: ch07/tempfile.hs
import System.IO
import System.Directory(getTemporaryDirectory, removeFile)
import System.IO.Error(catch)
import Control.Exception(finally)

-- The main entry point.  Work with a temp file in myAction.
main :: IO ()
main = withTempFile "mytemp.txt" myAction

{- The guts of the program.  Called with the path and handle of a temporary
file.  When this function exits, that file will be closed and deleted
because myAction was called from withTempFile. -}
myAction :: FilePath -> Handle -> IO ()
myAction tempname temph =
    do -- Start by displaying a greeting on the terminal
       putStrLn "Welcome to tempfile.hs"
       putStrLn $ "I have a temporary file at " ++ tempname

       -- Let's see what the initial position is
       pos <- hTell temph
       putStrLn $ "My initial position is " ++ show pos
```

v: latest

```
        -- Now, write some data to the temporary file
        let tempdata = show [1..10]
        putStrLn $ "Writing one line containing " ++
            show (length tempdata) ++ " bytes: " ++
                tempdata
        hPutStrLn temph tempdata

        -- Get our new position.  This doesn't actually modify pos
        -- in memory, but makes the name "pos" correspond to a different
        -- value for the remainder of the "do" block.
        pos <- hTell temph
        putStrLn $ "After writing, my new position is " ++ show pos

        -- Seek to the beginning of the file and display it
        putStrLn $ "The file content is: "
        hSeek temph AbsoluteSeek 0

        -- hGetContents performs a lazy read of the entire file
        c <- hGetContents temph

        -- Copy the file byte-for-byte to stdout, followed by \n
        putStrLn c

        -- Let's also display it as a Haskell literal
        putStrLn $ "Which could be expressed as this Haskell literal:"
        print c

{- This function takes two parameters: a filename pattern and another
function.  It will create a temporary file, and pass the name and Handle
of that file to the given function.

The temporary file is created with openTempFile.  The directory is the one
indicated by getTemporaryDirectory, or, if the system has no notion of
a temporary directory, "." is used.  The given pattern is passed to
openTempFile.

After the given function terminates, even if it terminates due to an
exception, the Handle is closed and the file is deleted. -}
withTempFile :: String -> (FilePath -> Handle -> IO a) -> IO a
withTempFile pattern func =
    do -- The library ref says that getTemporaryDirectory may raise on
       -- exception on systems that have no notion of a temporary directory.
       -- So, we run getTemporaryDirectory under catch.  catch takes
       -- two functions: one to run, and a different one to run if the
       -- first raised an exception.  If getTemporaryDirectory raised an
       -- exception, just use "." (the current working directory).
       tempdir <- catch (getTemporaryDirectory) (\_ -> return ".")
       (tempfile, temph) <- openTempFile tempdir pattern

       -- Call (func tempfile temph) to perform the action on the temporary
       -- file.  finally takes two actions.  The first is the action to run.
       -- The second is an action to run after the first, regardless of
       -- whether the first action raised an exception.  This way, we ensure
       -- the temporary file is always deleted.  The return value from finally
       -- is the first action's return value.
       finally (func tempfile temph)
               (do hClose temph
                   removeFile tempfile)
```

让我们从结尾开始看这个程序。 `writeTempFile` 函数证明Haskell当I/O被引入的时候没有忘记它的函数式特性。这个函数接受一个 `String` 和另外一个函数，传给 `withTempFile` 的函数使用这个名字和一个临时文件的句柄调用。当函数退出时，这个临时文件被关闭和删除。所以甚至在处理I/O时，我们仍然可以发现为了方便传递函数作为参数的习惯。Lisp程序员可能看到我们的 `withTempFile` 函数有点类似Lisp的 `with-open-file` 函数。

为了让程序能够更好地处理错误，我们需要为它添加一些异常处理代码。你一般需要临时文件在处理完成之后被删除，就算有错误发生。所以我们要确保删除发生。关于异常处理的更多信息，请看 **`第十九章：错误处理`_** 。

让我们回到这个程序的开头， `main` 被简单定义成 `withTempFile "mytemp.txt" myAction` 。然后, `myAction` 将会被调用，使用名 v: latest 文件的句柄作为参数。

`myAction` 显示一些信息到终端，写一些数据到文件，寻找文件的开头，并且使用 `hGetContents` 把数据读取回来。然后把文件的内容按字节地，通过 `print c` 当做Haskell字面量显示出来。这和 `putStrLn (show c)` 一样。

我们看一下输出：

```
$ runhaskell tempfile.hs
Welcome to tempfile.hs
I have a temporary file at /tmp/mytemp8572.txt
My initial position is 0
Writing one line containing 22 bytes: [1,2,3,4,5,6,7,8,9,10]
After writing, my new position is 23
The file content is:
[1,2,3,4,5,6,7,8,9,10]

Which could be expressed as this Haskell literal:
"[1,2,3,4,5,6,7,8,9,10]\n"
```

每次你运行这个程序，你的临时文件的名字应该有点细微的差别，因为它包含了一个随机生成的部分。看一下这个输出，你可能会问一些问题?

1. 为什么写入一行22个字节之后你的位置是23?
2. 为什么文件内容显示之后有一个空行?
3. 为什么Haskell字面量显示的最后有一个 `\n` ?

你可能能猜到这三个问题的答案都是相关的。看看你能不能在一会内答出这些题。如果你需要帮助，这里有解释：

1. 是因为我们用 `hPutStrLn` 替代 `hPutStr` 来写这个数据。 `hPutStrLn` 总是在结束一行的时候在结尾处写上一个 `\n` ，而这个没有出现在 `tempdata` 。
2. 我们用 `putStrLn c` 来显示文件内容 `c` 。因为数据原来使用 `hPutStrLn` 来写的，`c` 结尾处有一个换行符，并且 `putStrLn` 又添加了第二个换行符，结果就是多了一个空行。
3. 这个 `\n` 是来自原始的 `hPutStrLn` 的换行符。

最后一个注意事项，字节数目可能在一些操作系统上不一样。比如Windows，使用连个字节序列 `\r\n` 作为行结束标记，所以在Windows平台你可能会看到不同。

惰性I/O

这一章到目前为止，你已经看了一些相当传统的I/O例子。单独请求和处理每一行或者每一块数据。

Haskell还为你准备了另一种方法。因为Haskell是一种惰性语言，意思是任何给定的数据片只有在它的值必须要知道的情况下才会被计算。有一些新奇的方法来处理I/O。

hGetContents

一种新奇的处理I/O的办法是 `hGetContents` 函数，这个函数类型是 `Handle -> IO String` 。这个返回的 `String` 表示 `Handle` 所给文件里的所有数据。

在一个严格求值（strictly-evaluated）的语言中，使用这样的函数不是一件好事情。读取一个2KB文件的所有内容可能没事，但是如果你尝试去读取一个500GB文件的所有内容，你很可能因为缺少内存去存储这些数据而崩溃。在这些语言中，传统上你会采用循环去处理文件的全部数据的机制。

但是 `hGetContents` 不一样。它返回的 `String` 是惰性估值的。在你调用 `hGetContents` 的时刻，实际上没有读任何东西。数据只从句柄读取，作为处理的一个元素（字符）列表。 `String` 的元素一直都用不到，Haskell的垃圾收集器会自动释放那块内存。所有这些都是完全透明地发生的。因为函数的返回值是一个如假包换的纯 String ，所以它可以被传递给非 I/O 的纯代码。 让我们快速看一个例子。回到 **`操作文件和句柄`_** 一节，你看到一个命令式的程序，它把整个文件内容转换成大写。它的命令式算法和你在其他语言看到的很类似。接下来展示的是一个利用了惰性求值实现的更简单的算法。

```
-- file: ch07/toupper-lazy1.hs
import System.IO
import Data.Char(toUpper)

main :: IO ()
main = do
       inh <- openFile "input.txt" ReadMode
```

v: latest

```
        outh <- openFile "output.txt" WriteMode
        inpStr <- hGetContents inh
        let result = processData inpStr
        hPutStr outh result
        hClose inh
        hClose outh

processData :: String -> String
processData = map toUpper
```

注意到 hGetContents 为我们处理所有的读取工作。看一下 processData ，它是一个纯函数，因为它没有副作用，并且每次调用的时候总是返回相同的结果。它不需要知道，也没办法告诉它，它的输入是惰性从文件读取的。不管是20个字符的字面量还是硬盘上500GB的数据它都可以很好的工作。

你可以用 **ghci** 验证一下：

```
ghci> :load toupper-lazy1.hs
[1 of 1] Compiling Main             ( toupper-lazy1.hs, interpreted )
Ok, modules loaded: Main.
ghci> processData "Hello, there!  How are you?"
"HELLO, THERE!  HOW ARE YOU?"
ghci> :type processData
processData :: String -> String
ghci> :type processData "Hello!"
processData "Hello!" :: String
```

> **Warning**
>
> 如果我们尝试去抓住上面例子中的 inpStr ，在超过它被使用的地方（ processData 调用那），内存中将没有它了。这是因为编译器会强制保存 inpStr 的值在内存里，为了以后的使用。这里我们知道 inpStr 讲不会被重用，它一被使用完就会被释放内存。只要记住：最后一次使用后释放内存。

这个程序为了清楚地表明使用了存代码，显得有点啰嗦。这里有更加简洁的版本，新版本在下一个例子里：

```
-- file: ch07/toupper-lazy2.hs
import System.IO
import Data.Char(toUpper)

main = do
        inh <- openFile "input.txt" ReadMode
        outh <- openFile "output.txt" WriteMode
        inpStr <- hGetContents inh
        hPutStr outh (map toUpper inpStr)
        hClose inh
        hClose outh
```

你在使用 hGetContents 的时候不要求去使用输入文件的所有数据。任何时候Haskell系统能决定整个 hGgetContents 返回的字符串能否被垃圾收集掉，意思就是它不会再被使用，文件会自动被关闭。同样的原理适用于从文件读取的数据。当给定的数据片不会再被使用的任何时候，Haskell会释放它保存的那块内存。严格意义上来讲，我们在这个例子中根本不必要去调用 hClose 。但是，养成习惯去调用还是个好的实践。以后对程序的修改可能让 hClose 的调用变得重要。

> **Warning**
>
> 当使用 hGetContents 的时候，记住，就算你可能在剩下的程序里面不再显式引用句柄 ，你绝不能关闭句柄 ，直到在你结束对结果的使用后， 这点很重要。提早关闭会造成丢失文件数据的部分或全部。因为Haskell是惰性的，一般地可以假定，你只有在包含输入的计算被算出结果输出之后，你才能使用这个输入。

readFile和writeFile

Haskell程序员经常使用 hGetContents 作为一个过滤器。他们从一个文件读取，在数据上做一些事情，然后把结果写到其 常见，有很多种快捷方式可以做。 readFile 和 writeFile 是把文件当做字符串处理的快捷方式。他们处理所有细节，包括打开文件，关闭文件，读取文件和写入文件。 readFile 在内部使用 hGetContents 。

你能猜到这些函数的Haskell类型吗？我们用 **ghci** 检查一下：

```
ghci> :type readFile
readFile :: FilePath -> IO String
ghci> :type writeFile
writeFile :: FilePath -> String -> IO ()
```

现在有一个例子程序使用了 `readFile` 和 `writeFile` ：

```
-- file: ch07/toupper-lazy3.hs
import Data.Char(toUpper)

main = do
       inpStr <- readFile "input.txt"
       writeFile "output.txt" (map toUpper inpStr)
```

看一下，这个程序的内部只有两行。 `readFile` 返回一个惰性 `String` ，我们保存在 `inpStr` 。然后我们拿到它，处理它，然后把它传给 `writeFile` 函数去写入。

`readFile` 和 `writeFile` 都不提供一个句柄给你操作，所以没有东西要去 `hClose` 。 `readFile` 在内部使用 `hGetContents` ，底下的句柄在返回的 `String` 被垃圾回收或者所有输入都被消费之后就会被关闭。 `writeFile` 会在供应给它的 `String` 全部被写入之后关闭它底下的句柄。

一言以蔽惰性输出

到现在为止，你应该理解了Haskell的惰性输入怎么工作的。但是在输入的时候惰性是怎么样的呢？

据你所知，Haskell中的所有东西都是在需要的时候才被求值的。因为像 `writeFile` 和 `putStr` 这样的函数写传递给它们的整个 `String` ， 所以这整个 `String` 必须被求值。所以保证 `putStr` 的参数会被完全求值。

但是输入的惰性是什么意思呢？ 在上面的例子中，对 `putStr` 或者 `writeFile` 的调用会强制一次性把整个输入字符串载入到内存中吗，直接全部写出？

答案是否定的。 `putStr` （以及所有类似的输出函数）在它变得可用时才写出数据。他们也不需要保存已经写的数据，所以只要程序中没有其他地方需要它，这块内存就可以立即释放。在某种意义上，你可以把这个在 `readFile` 和 `writeFile` 之间的 `String` 想成一个连接它们两个的管道。数据从一头进去，通过某种方式传递，然后从另外一头流出。

你可以自己验证这个，通过给 `toupper-lazy3.hs` 产生一个大的 `input.txt` 。处理它可能时间要花一点时间，但是在处理它的时候你应该能看到一个常量的并且低的内存使用。

interact

你学习了 `readFile` 和 `writeFile` 处理读文件，做个转换，然后写到不同文件的普通情形。还有一个比他还普遍的情形：从标准输入读取，做一个转换，然后把结果写到标准输出。对于这种情形，有一个函数叫做 `interact` 。 `interact` 函数的类型是 `(String -> String) -> IO ()` 。也就是说，它接受一个参数：一个类型为 `String -> String` 的函数。 `getContents` 的结果传递给这个函数，也就是，惰性读取标准输入。这个函数的结果会发送到标准输出。

我们可以使用 `interact` 来转换我们的例子程序去操作标准输入和标准输出。这里有一种方式：

```
-- file: ch07/toupper-lazy4.hs
import Data.Char(toUpper)

main = interact (map toUpper)
```

来看一下，一行就完成了我们的变换。要实现上一个例子同样的效果，你可以像这样来运行这个例子：

```
$ runghc toupper-lazy4.hs < input.txt > output.txt
```

或者，如果你想看输出打印在屏幕上的话，你可以打下面的命令：


```
$ runghc toupper-lazy4.hs < input.txt
```

如果你想看看Haskell是否真的一接收到数据块就立即写出的话，运行 `runghc toupper-lazy4.hs` ，不要其他的命令行参数。你可以看到每一个你输入的字符都会立马回显，但是都变成大写了。缓冲区可能改变这种行为，更多关于缓冲区的看这一章后面的 **`缓冲区`_** 一节。如果你看到你输入的没一行都立马回显，或者甚至一段时间什么都没有，那就是缓冲区造成的。

你也可以用 `interactive` 写一个简单的交互程序。让我们从一个简单的例子开始：

```
-- file: ch07/toupper-lazy5.hs
import Data.Char(toUpper)

main = interact (map toUpper . (++) "Your data, in uppercase, is:\n\n")
```

Tip

如果 `.` 运算符不明白的话，你可以参考 **`使用组合来重用代码`_** 一节。

这里我们在输出的开头添加了一个字符串。你可以发现这个问题吗？

因为我们在 `(++)` 的结果上调用 `map` ，这个头自己也会显示成大写。我们可以这样来解决：

```
-- file: ch07/toupper-lazy6.hs
import Data.Char(toUpper)

main = interact ((++) "Your data, in uppercase, is:\n\n" .
                 map toUpper)
```

现在把头移出了 `map` 。

interact 过滤器

`interact` 另一个通常的用法是过滤器。比如说你要写一个程序，这个程序读一个文件，并且输出所有包含字符“a”的行。你可能会这样用 `interact` 来实现：

```
-- file: ch07/filter.hs
main = interact (unlines . filter (elem 'a') . lines)
```

这里引入了三个你还不熟悉的函数。让我们在 **ghci** 里检查它们的类型：

```
ghci> :type lines
lines :: String -> [String]
ghci> :type unlines
unlines :: [String] -> String
ghci> :type elem
elem :: (Eq a) => a -> [a] -> Bool
```

你只是看它们的类型，你能猜到它们是干什么的吗？如果不能，你可以在 **`热身：快捷文本行分割`_** 一节和 **`特殊字符串处理函数`_** 一节找到解释。你会频繁看到 `lines` 和 `unlines` 和I/O一起使用。最后， `elem` 接受一个元素和一个列表，如果元素在列中中出现则返回 `True` 。

试着用我们的标准输入例子来运行：

```
$ runghc filter.hs < input.txt
I like Haskell
Haskell is great
```

果然，你得到包含“a”的两行。惰性过滤器是使用Haskell强大的方式。你想想看，一个过滤器，就像标准Unix程序 **Grep** ，听起来很像一个函数。它接受一些输入，应用一些计算，然后生成一个意料之中的输出。

The IO Monad

v: latest

这个时候你已经看了若干Haskell中I/O的例子。让我们花点时间回想一下，并且思考下I/O是怎么和更广阔的Haskell语言相关联的。

因为Haskell是一个纯的语言，如果你给特定的函数一个指定的参数，每次你给它那个参数这个函数将会返回相同的结果。此外，这个函数不会改变程序的总体状态的任何东西。

你可能想知道I/O是怎么融合到整体中去的呢？当然如果你想从键盘输入中读取一行，去读输入的那个函数肯定不可能每次都返回相同的结果。是不是？此外，I/O都是和改变状态相关的。I/O可以点亮终端上的一个像素，可以让打印机的纸开始出来，或者甚至是让一个包裹从仓库运送到另一个大洲。I/O不只是改变一个程序的状态。你可以把I/O想成可以改变世界的状态。

动作（Actions）

大多数语言在纯函数和非纯函数之间没有明确的区分。Haskell的函数有数学上的意思：它们是纯粹的计算过程，并且这些计算不会被外部所影响。此外，这些计算可以在任何时候、按需地执行。

显然，我们需要其他一些工具来使用I/O。Haskell里的这个工具叫做动作（Actions）。动作类似于函数，它们在定义的时候不做任何事情，而在它们被调用时执行一些任务。I/O动作被定义在 `IO` Monad。Monad是一种强大的将函数链在一起的方法，在 **`第十四章：Monad`_** 会讲到。为了理解I/O你不是一定要理解Monad，只要理解操作的返回类型都带有 `IO` 就行了。我们来看一些类型：

```
ghci> :type putStrLn
putStrLn :: String -> IO ()
ghci> :type getLine
getLine :: IO String
```

`putStrLn` 的类型就像其他函数一样，接受一个参数，返回一个 `IO ()` 。这个 `IO ()` 就是一个操作。如果你想你可以在纯代码中保存和传递操作，虽然我们不经常这么干。一个操作在它被调用前不做任何事情。我们看一个这样的例子：

```
-- file: ch07/actions.hs
str2action :: String -> IO ()
str2action input = putStrLn ("Data: " ++ input)

list2actions :: [String] -> [IO ()]
list2actions = map str2action

numbers :: [Int]
numbers = [1..10]

strings :: [String]
strings = map show numbers

actions :: [IO ()]
actions = list2actions strings

printitall :: IO ()
printitall = runall actions

-- Take a list of actions, and execute each of them in turn.
runall :: [IO ()] -> IO ()
runall [] = return ()
runall (firstelem:remainingelems) =
    do firstelem
       runall remainingelems

main = do str2action "Start of the program"
          printitall
          str2action "Done!"
```

`str2action` 这个函数接受一个参数并返回 `IO ()` ，就像你在 `main` 结尾看到的那样，你可以直接在另一个操作里使用这个函数，它会立刻打印出一行。或者你可以保存（不是执行）纯代码中的操作。你可以在 `list2actions` 里看到保存的例子，我们在 `str2action` 用 `map` ，返回一个操作的列表，就和操作其他纯数据一样。所有东西都通过 `printall` 显示出来， 而 `printall` 是用纯代码写的。

虽然我们定义了 `printall` ，但是直到它的操作在其他地方被求值的时候才会执行。现在注意，我们是怎么在 `main` 里把 `str2action` 当做一个I/O操作使用，并且执行了它。但是先前我们在I/O Monad外面使用它，只是把结果收集进一个列表。

你可以这样来思考： `do` 代码块中的每一个声明，除了 `let` ，都要产生一个I/O操作，这个操作在将来被执行。

对 `printall` 的调用最后会执行所有这些操作。实际上，因为Haskell是惰性的，所以这些操作直到这里才会被生成。

当你运行这个程序时，你的输出看起来像这样：

```
Data: Start of the program
Data: 1
Data: 2
Data: 3
Data: 4
Data: 5
Data: 6
Data: 7
Data: 8
Data: 9
Data: 10
Data: Done!
```

我们实际上可以写的更紧凑。来看看这个例子的修改：

```
-- file: ch07/actions2.hs
str2message :: String -> String
str2message input = "Data: " ++ input

str2action :: String -> IO ()
str2action = putStrLn . str2message

numbers :: [Int]
numbers = [1..10]

main = do str2action "Start of the program"
          mapM_ (str2action . show) numbers
          str2action "Done!"
```

注意在 `str2action` 里对标准函数组合运算符的使用。在 `main` 里面，有一个对 `mapM_` 的调用，这个函数和 `map` 类似，接受一个函数和一个列表。提供给 `mapM_` 的函数是一个I/O操作，这个操作对列表中的每一项都执行。 `mapM_` 扔掉了函数的结果，但是如果你想要 I/O的结果，你可以用 `mapM` 返回一个I/O结果的列表。来看一下它们的类型：

```
ghci> :type mapM
mapM :: (Monad m) => (a -> m b) -> [a] -> m [b]
ghci> :type mapM_
mapM_ :: (Monad m) => (a -> m b) -> [a] -> m ()
```

Tip

这些函数其实可以做I/O更多的事情，所有的Monad都可以使用他们。到现在为止，你看到“M”就把它想成“IO”。还有，那些以下划线结尾的函数一般不管它们的返回值。

为什么我们有了 `map` 还要有一个 `mapM` ，因为 `map` 是返回一个列表的纯函数，它实际上不直接执行也不能执行操作。 `maoM` 是一个 `IO` Monda里面的可以执行操作的实用程序。

现在回到 `main` ， `mapM_` 在 `numbers . show` 每个元素上应用 `(str2action . show)` ， `number . show` 把每个数字转换成一个 `String` ， `str2action` 把每个 `String` 转换成一个操作。 `mapM_` 把这些单独的操作组合成一个打的操作，然后打印出这些行。

串联化

`do` 代码块实际上是把操作连接在一起的快捷记号。有两个运算符可以用来代替 `do` 代码块： `>>` 和 `>>=` 。在 **ghci** 看一下它们的类型：

```
ghci> :type (>>)
(>>) :: (Monad m) => m a -> m b -> m b
ghci> :type (>>=)
(>>=) :: (Monad m) => m a -> (a -> m b) -> m b
```

`>>` 运算符把两个操作串联在一起：第一个操作先运行，然后是第二个。运算符的计算的结果是第二个操作的结果，第 v: latest 果被丢弃了。这和在 `do` 代码块中只有一行是类似的。你可能会写 `putStrLn "line 1" >> putStrLn "line 2"` 来测试这一点。它会打印出两行，把第一个 `putStrLn` 的结果丢掉了，值提供第二个操作的结果。

`>>=` 运算符运行一个操作，然后把它的结果传递给一个返回操作的函数。那样第二个操作可以同样运行，而且整个表达式的结果就是第二个操作的结果。例如，你写 `getLine >>= putStrLn` ，这会从键盘读取一行，然后显示出来。

让我们重写例子中的一个，不用 `do` 代码快。还记得这一章开头的这个例子吗?

```
-- file: ch07/basicio.hs
main = do
       putStrLn "Greetings!  What is your name?"
       inpStr <- getLine
       putStrLn $ "Welcome to Haskell, " ++ inpStr ++ "!"
```

我们不用 `do` 代码块来重写它:

```
-- file: ch07/basicio-nodo.hs
main =
    putStrLn "Greetings!  What is your name?" >>
    getLine >>=
    (\inpStr -> putStrLn $ "Welcome to Haskell, " ++ inpStr ++ "!")
```

你定义 `do` 代码块的时候，Haskell编译器内部会把它翻译成像这样。

Tip

忘记了怎么使用 `\` (lambda表达式)了吗? 参见 **`匿名（lambda）函数`_** 一节。

Return的本色

在这一章的前面，我们提到 `return` 很可能不是它看起来的那样。很多语言有一个关键字叫做 `return` ，它取消函数的执行并立即给调用者一个返回值。

Haskell的 `return` 函数很不一样。在Haskell中， `return` 用来在Monad里面包装数据。当说I/O的时候， `return` 用来拿到纯数据并把它带入IO Monad。

为什么我们需要那样做? 还记得结果依赖I/O的所有东西都必须在一个IO Monad里面吗? 所以如果我们在写一个执行I/O的函数，然后一个纯的计算，我们需要用 `return` 来让这个纯的计算能给函数返回一个合适的值。否则，会发生一个类型错误。这儿有一个例子:

```
-- file: ch07/return1.hs
import Data.Char(toUpper)

isGreen :: IO Bool
isGreen =
    do putStrLn "Is green your favorite color?"
       inpStr <- getLine
       return ((toUpper . head $ inpStr) == 'Y')
```

我们有一个纯的计算产生一个 `Bool` ，这个计算传给了 `return` ， `return` 把它放进了 `IO` Monad。因为它是 `do` 代码块的最后一个值，所以它变成 `isGreen` 的返回值，而不是因为我们用了 `return` 函数。

这有一个相同程序但是把纯计算移到一个单独的函数里的版本。这帮助纯代码保持分离，并且让意图更清晰。

```
-- file: ch07/return2.hs
import Data.Char(toUpper)

isYes :: String -> Bool
isYes inpStr = (toUpper . head $ inpStr) == 'Y'

isGreen :: IO Bool
isGreen =
    do putStrLn "Is green your favorite color?"
       inpStr <- getLine
       return (isYes inpStr)
```

最后，有一个人为的例子，这个例子显示了 return 确实没有在 do 代码块的结尾出现。在实践中，通常是这样的，但是不一定需要这样。

```
-- file: ch07/return3.hs
returnTest :: IO ()
returnTest =
    do one <- return 1
       let two = 2
       putStrLn $ show (one + two)
```

注意，我们用了 <- 和 return 的组合，但是 let 是和简单字面量组合的。这是因为我们需要都是纯的值才能去相加它们， <- 把东西从Monad里面拿出来，实际上就是 return 的反作用。在 **ghci** 运行一下，你会看到和预期一样显示3。

Haskell 实际上是命令式的吗?

这些 do 代码块可能开起来很像一个命令式语言？毕竟大部分时间你给了一些命令按顺序运行。

但是Haskell在它的核心上是一个惰性语言。时常在需要给I/O串联操作的时候，是由一些工具完成的，这些工具就是Haskell的一部分。Haskell通过 I/O Monad实现了出色的I/O和语言剩余部分的分离。

惰性I/O的副作用

本章前面你看到了 hGetContents ，我们解释说它返回的 String 可以在纯代码中使用。

关于副作用我们需要得到一些更具体的东西。当我们说Haskell没有副作用，这到底意味着什么?

在一定程度上，副作用总是可能的。一个写的不好的循环，就算写成纯代码形式的，也会造成系统内存耗尽和机器崩溃，或者导致数据交换到硬盘上。

当我们说没有副作用的时候，我们意思是，Haskell中的存代码不能运行那些能触发副作用的命令。纯函数不能修改全局变量，请求I/O，或者运行一条关闭系统的命令。

当你有从 hGetContents 拿到一个 String ，你把它传给一个纯函数，这个函数不知道这个 String 是由硬盘文件上来的。这个函数表现地还是和原来一样，但是处理那个 String 的时候可能造成环境发出I/O命令。纯函数是不会发出I/O命令的，它们作为处理正在运行的纯函数的一个结果，就和交换内存到磁盘的例子一样。

有时候，你在I/O发生时需要更多的控制。可能你正在从用户那里交互地读取数据，或者通过管道从另一个程序读取数据，你需要直接和用户交流。在这些时候， hGetContents 可能就不合适了。

缓冲区（Buffering）

I/O子系统是现代计算机中最慢的部分之一。完成一次写磁盘的时间是一次写内存的几千倍。在网络上的写入还要慢成百上千倍。就算你的操作没有直接和磁盘通信，可能数据被缓存了，I/O还是需要一个系统调用，这个也会减慢速度。

由于这个原因，现代操作系统和编程语言都提供了工具来帮助程序当涉及到I/O的时候更好地运行。操作系统一般采用缓存（Cache），把频繁使用的数据片段保存在内存中，这样就能更快的访问了。

编程语言通常采用缓冲区。就是说，它们可能从操作系统请求一大块数据，就算底层代码是一次一个字节地处理数据的。通过这样，它们可以实现显著的性能提升，因为每次向操作系统的I/O请求带来一次处理开销。缓冲区允许我们去读相同数量的数据可以用少得多的I/O请求。

缓冲区模式

Haskell中有3种不同的缓冲区模式，它们定义成 BufferMode 类型： NoBuffering ， LineBuffering 和 BlockBuffering 。

NoBuffering 就和它听起来那样-没有缓冲区。通过像 hGetLine 这样的函数读取的数据是从操作系统一次一个字符读取的。写入的数据会立即写入，也是一次一个字符地写入。因此， NoBuffering 通常性能很差，不适用于一般目的的使用。

LineBuffering 当换行符输出的时候会让输出缓冲区写入，或者当缓冲区太大的时候。在输入上，它通常试图去读取块 v: latest]字符，直到它首次遇到换行符。当从终端读取的时候，每次按下回车之后它会立即返回数据。这个模式经常是默认模式。

`BlockBuffering` 让Haskell在可能的时候以一个固定的块大小读取或者写入数据。这在批处理大量数据的时候是性能做好的，就算数据是以行存储的也是一样。然而，这个对于交互程序不能用，因为它会阻塞输入直到一整块数据被读取。 `BlockBuffering` 接受一个 `Maybe` 类型的参数： 如果是 `Nothing` ， 它会使用一个自定的缓冲区大小，或者你可以使用一个像 `Just 4096` 的设定，设置缓冲区大小为4096个字节。

默认的缓冲区模式依赖于操作系统和Haskell的实现。你可以通过调用 `hGetBuffering` 查看系统的当前缓冲区模式。当前的模式可以通过 `hSetBuffering` 来设置，它接受一个 `Handle` 和 `BufferMode` 。例如，你可以写 `hSetBuffering stdin (BlockBuffering Nothing)` 。

刷新缓冲区

对于任何类型的缓冲区，你可能有时候需要强制Haskell去写出所有保存在缓冲区里的数据。有些时候这个会自动发生：比如，对 `hClose` 的调用。有时候你可能需要调用 `hFlush` 作为代替， `hFlush` 会强制所有等待的数据立即写入。这在句柄是一个网络套接字的时候，你想数据被立即传输，或者你想让磁盘的数据给其他程序使用，而其他程序也正在并发地读那些数据的时候都是有用的。

读取命令行参数

很多命令行程序喜欢通过命令行来传递参数。 `System.Environment.getArgs` 返回 `IO [String]` 列出每个参数。这和C语言的 `argv` 一样，从 `argv[1]` 开始。程序的名字（C语言的 `argv[0]` ）用 `System.Environment.getProgName` 可以得到。

`System.Console.GetOpt` 模块提供了一些解析命令行选项的工具。如果你有一个程序，它有很复杂的选项，你会觉得它很有用。你可以在 **`命令行解析`_** 一节看到一个例子和使用方法。

环境变量

如果你需要阅读环境变量，你可以使用 `System.Environment` 里面两个函数中的一个： `getEnv` 或者 `getEnvironment` 。 `getEnv` 查找指定的变量，如果不存在会抛出异常。 `getEnvironment` 用一个 `[(String, String)]` 返回整个环境，然后你可以用 `lookup` 这样的函数来找你想要的环境条目。

在Haskell设置环境变量没有采用跨平台的方式来定义。如果你在像Linux这样的POSIX平台上，你可以使用 `System.Posix.Env` 模块中的 `putEnv` 或者 `setEnv` 。环境设置在Windows下面没有定义。

讨论