

## 第二十章：使用 Haskell 进行系统编程

目前为止，我们讨论的大多数是高阶概念。Haskell 也可以用于底层系统编程。完全可以使用 Haskell 编写使用操作系统底层接口的程序。

本章中，我们将尝试一些很有野心的东西：编写一种类似 Perl 实际上是合法的 Haskell 的“语言”，完全使用 Haskell 实现，用于简化编写 shell 脚本。我们将实现管道，简单命令调用，和一些简单的工具用于执行由 `grep` 和 `sed` 处理的任务。

有些模块是依赖操作系统的。本章中，我们将尽可能使用不依赖特殊操作系统的通用模块。不过，本章将有很多内容着眼于 POSIX 环境。POSIX 是一种类 Unix 标准，如 Linux，FreeBSD，MacOS X，或 Solaris。Windows 默认情况下不支持 POSIX，但是 Cygwin 环境为 Windows 提供了 POSIX 兼容层。

### 调用外部程序

Haskell 可以调用外部命令。为了这么做，我们建议使用 `System.Cmd` 模块中的 `rawSystem`。其用特定的参数调用特定的程序，并将返回程序的退出状态码。你可以在 `ghci` 中练习一下。

```
ghci> :module System.Cmd
ghci> rawSystem "ls" ["-l", "/usr"]
Loading package old-locale-1.0.0.0 ... linking ... done.
Loading package old-time-1.0.0.0 ... linking ... done.
Loading package filepath-1.1.0.0 ... linking ... done.
Loading package directory-1.0.0.0 ... linking ... done.
Loading package unix-2.3.0.0 ... linking ... done.
Loading package process-1.0.0.0 ... linking ... done.
total 124
drwxr-xr-x  2 root root  49152 2008-08-18 11:04 bin
drwxr-xr-x  2 root root   4096 2008-03-09 05:53 games
drwxr-sr-x 10 jimb guile  4096 2006-02-04 09:13 guile
drwxr-xr-x 47 root root   8192 2008-08-08 08:18 include
drwxr-xr-x 107 root root 32768 2008-08-18 11:04 lib
lrwxrwxrwx  1 root root      3 2007-09-24 16:55 lib64 -> lib
drwxrwsr-x 17 root staff  4096 2008-06-24 17:35 local
drwxr-xr-x  2 root root   8192 2008-08-18 11:03 sbin
drwxr-xr-x 181 root root   8192 2008-08-12 10:11 share
drwxrwsr-x  2 root src   4096 2007-04-10 16:28 src
drwxr-xr-x  3 root root   4096 2008-07-04 19:03 X11R6
ExitSuccess
```

此处，我们相当于执行了 shell 命令 `ls -l /usr`。`rawSystem` 并不从字符串解析输入参数或是扩展通配符 [43]。取而代之，其接受一个包含所有参数的列表。如果不想提供参数，可以像这样简单地输入一个空列表。

```
ghci> rawSystem "ls" []
calendartime.ghci  modtime.ghci  rp.ghci  RunProcessSimple.hs
cmd.ghci          posixtime.hs  rps.ghci  timediff.ghci
dir.ghci          rawSystem.ghci  RunProcess.hs  time.ghci
ExitSuccess
```

### 目录和文件信息

`System.Directory` 模块包含了相当多可以从文件系统获取信息的函数。你可以获取某目录包含的文件列表，重命名或删除文件，复制文件，改变当前工作路径，或者建立新目录。`System.Directory` 是可移植的，在可以跑 GHC 的平台都可以使用。

**System.Directory 的库文档** 中含有一份详尽的函数列表。让我们通过 `ghci` 来对其中一些进行演示。这些函数大多数简单的等价于其对应的 C 语言库函数或 shell 命令。

```
ghci> :module System.Directory
ghci> setCurrentDirectory "/etc"
Loading package old-locale-1.0.0.0 ... linking ... done.
Loading package old-time-1.0.0.0 ... linking ... done.
Loading package filepath-1.1.0.0 ... linking ... done.
Loading package directory-1.0.0.0 ... linking ... done.
```

 v: latest ▾

```
ghci> getCurrentDirectory
"/etc"
ghci> setCurrentDirectory ".."
ghci> getCurrentDirectory
"/"
```

此处我们看到了改变工作目录和获取当前工作目录的命令。它们类似 POSIX shell 中的 `cd` 和 `pwd` 命令。

```
ghci> getDirectoryContents "/"
[".", "..", "lost+found", "boot", "etc", "media", "initrd.img", "var", "usr", "bin", "dev", "home", "lib", "mnt", "proc", "root", "sbin", "tmp", "sys", "lib64", "s
```

`getDirectoryContents` 返回一个列表，包含给定目录的所有内容。注意，在 POSIX 系统中，这个列表通常包含特殊值 `."` 和 `.."`。通常在处理目录内容时，你可能会希望将他们过滤出去，像这样：

```
ghci> getDirectoryContents "/" >>= return . filter (notElem `[".", ".."]` )
["lost+found", "boot", "etc", "media", "initrd.img", "var", "usr", "bin", "dev", "home", "lib", "mnt", "proc", "root", "sbin", "tmp", "sys", "lib64", "srv", "opt"
```

### Tip

更细致的讨论如何过滤 `getDirectoryContents` 函数的结果，请参考 [第八章：高效文件处理、正则表达式、文件名匹配](#)

`filter (notElem `[".", ".."]` )` 这段代码是否有点莫名其妙？也可以写作 `filter (c -> not $ elem c `[".", ".."]` )`。反引号让我们更有效的将第二个参数传给 `notElem`；在“中序函数”一节中有关于反引号更详细的信息。

也可以向系统查询某些路径的位置。这将向底层操作系统发起查询相关信息。

```
ghci> getHomeDirectory
"/home/bos"
ghci> getAppUserDataDirectory "myApp"
"/home/bos/. myApp"
ghci> getUserDocumentsDirectory
"/home/bos"
```

## 终止程序

开发者经常编写独立的程序以完成特定任务。这些独立的部分可能会被组合起来完成更大的任务。一段 shell 脚本或者其他程序将会执行它们。发起调用的脚本需要获知被调用程序是否执行成功。Haskell 自动为异常退出的程序分配一个“不成功”的状态码。

不过，你需要对状态码进行更细粒度的控制。可能你需要对不同类型的错误返回不同的代码。`System.Exit` 模块提供一个途径可以在程序退出时返回特定的状态码。通过调用 `exitWith ExitSuccess` 表示程序执行成功（POSIX 系统中的 0）。或者可以调用 `exitWith (ExitFailure 5)`，表示将在程序退出时向系统返回 5 作为状态码。

## 日期和时间

从文件时间戳到商业事务的很多事情都涉及到日期和时间。除了从系统获取日期时间信息之外，Haskell 提供了很多关于时间日期的操作方法。

### ClockTime 和 CalendarTime

在 Haskell 中，日期和时间主要由 `System.Time` 模块处理。它定义了两个类型：`ClockTime` 和 `CalendarTime`。

`ClockTime` 是传统 POSIX 中时间戳的 Haskell 版本。`ClockTime` 表示一个相对于 UTC 1970 年 1 月 1 日 零点的时间。负值的 `ClockTime` 表示在其之前的秒数，正值表示在其之后的秒数。

`ClockTime` 便于计算。因为它遵循协调世界时（Coordinated Universal Time，UTC），其不必调整本地时区、夏令时或其他时间处理中的特例。每天是精确的  $(60 * 60 * 24)$  或 86,400 秒 [44]，这易于计算时间间隔。举个例子，你可以简单的记录某个程序 `v: latest` 时间和其结束的时间，相减即可确定程序的执行时间。如果需要的话，还可以除以 3600，这样就可以按小时显示。

使用 `ClockTime` 的典型场景：

经过了多长时间？  
相对此刻 14 天前是什么时间？  
文件的最后修改时间是何时？  
当下的精确时间是何时？

`ClockTime` 善于处理这些问题，因为它们使用无法混淆的精确时间。但是，`ClockTime` 不善于处理下列问题：

今天是周一吗？  
明年 5 月 1 日是周几？  
在我的时区当前是什么时间，考虑夏令时。

`CalendarTime` 按人类的方式存储时间：年，月，日，小时，分，秒，时区，夏令时信息。很容易的转换为便于显示的字符串，或者以上问题的答案。

你可以任意转换 `ClockTime` 和 `CalendarTime`。Haskell 将 `ClockTime` 可以按本地时区转换为 `CalendarTime`，或者按 `CalendarTime` 格式表示的 UTC 时间。

## 使用 `ClockTime`

`ClockTime` 在 `System.Time` 中这样定义：

```
data ClockTime = TOD Integer Integer
```

第一个 `Integer` 表示从 Unix 纪元开始经过的秒数。第二个 `Integer` 表示附加的皮秒数。因为 Haskell 中的 `ClockTime` 使用无边界的 `Integer` 类型，所以其能够表示的数据范围仅受计算资源限制。

让我们看看使用 `ClockTime` 的一些方法。首先是按系统时钟获取当前时间的 `getClockTime` 函数。

```
ghci> :module System.Time
ghci> getClockTime
Loading package old-locale-1.0.0.0 ... linking ... done.
Loading package old-time-1.0.0.0 ... linking ... done.
Mon Aug 18 12:10:38 CDT 2008
```

如果一秒钟再次运行 `getClockTime`，它将返回一个更新后的时间。这条命令会输出一个便于观察的字符串，补全了周相关的信息。这是由于 `ClockTime` 的 `Show` 实例。让我们从更底层看一下 `ClockTime`：

```
ghci> TOD 1000 0
Wed Dec 31 18:16:40 CST 1969
ghci> getClockTime >>= (\(TOD sec _) -> return sec)
1219079438
```

这里我们先构建一个 `ClockTime`，表示 UTC 时间 1970 年 1 月 1 日午夜后 1000 秒这个时间点。在你的时区这个时间相当于 1969 年 12 月 31 日晚。

第二个例子演示如何从 `getClockTime` 返回值中将秒数取出来。我们可以像这样操作它：

```
ghci> getClockTime >>= (\(TOD sec _) -> return (TOD (sec + 86400) 0))
Tue Aug 19 12:10:38 CDT 2008
```

这将显精确示你的时区 24 小时后的时间，因为 24 小时等于 86,400 秒。

## 使用 `CalendarTime`

正如其名字暗示的，`CalendarTime` 按日历上的方式表示时间。它包括年、月、日等信息。`CalendarTime` 和其相关类型定义如下：

```
data CalendarTime = CalendarTime
  {ctYear :: Int,           -- Year (post-Gregorian)
   ctMonth :: Month,
   ctDay :: Int,           -- Day of the month (1 to 31)}
```

 v: latest ▾

```

ctHour :: Int,           -- Hour of the day (0 to 23)
ctMin  :: Int,           -- Minutes (0 to 59)
ctSec  :: Int,           -- Seconds (0 to 61, allowing for leap seconds)
ctPicosec :: Integer,    -- Picoseconds
ctWDay :: Day,           -- Day of the week
ctYDay :: Int,           -- Day of the year (0 to 364 or 365)
ctTZName :: String,      -- Name of timezone
ctTZ    :: Int,          -- Variation from UTC in seconds
ctIsDST :: Bool          -- True if Daylight Saving Time in effect
}

data Month = January | February | March | April | May | June
           | July | August | September | October | November | December

data Day = Sunday | Monday | Tuesday | Wednesday
         | Thursday | Friday | Saturday

```

关于以上结构有些事情需要强调：

`ctWDay`, `ctYDay`, `ctTZName` 是被创建 `CalendarTime` 的库函数生成的，但是并不参与计算。如果你手工创建一个 `CalendarTime`，不必向其中填写准确的值，除非你的计算依赖于它们。

这三个类型都是 `Eq`, `Ord`, `Read`, `Show` 类型类的成员。另外，`Month` 和 `Day` 都被声明为 `Enum` 和 `Bounded` 类型类的成员。更多的信息请参考“重要的类型类”这一章节。

有几种不同的途径可以生成 `CalendarTime`。可以像这样将 `ClockTime` 转换为 `CalendarTime`：

```

ghci> :module System.Time
ghci> now <- getClockTime
Loading package old-locale-1.0.0.0 ... linking ... done.
Loading package old-time-1.0.0.0 ... linking ... done.
Mon Aug 18 12:10:35 CDT 2008
ghci> nowCal <- toCalendarTime now
CalendarTime {ctYear = 2008, ctMonth = August, ctDay = 18, ctHour = 12, ctMin = 10, ctSec = 35, ctPicosec = 804267000000, ctWDay = Monday, ctYD
ghci> let nowUTC = toUTCTime now
ghci> nowCal
CalendarTime {ctYear = 2008, ctMonth = August, ctDay = 18, ctHour = 12, ctMin = 10, ctSec = 35, ctPicosec = 804267000000, ctWDay = Monday, ctYD
ghci> nowUTC
CalendarTime {ctYear = 2008, ctMonth = August, ctDay = 18, ctHour = 17, ctMin = 10, ctSec = 35, ctPicosec = 804267000000, ctWDay = Monday, ctYD

```

用 `getClockTime` 从系统获得当前的 `ClockTime`。接下来，`toCalendarTime` 按本地时间区将 `ClockTime` 转换为 `CalendarTime`。`toUTCTime` 执行类似的转换，但其结果将以 UTC 时区表示。

注意，`toCalendarTime` 是一个 IO 函数，但是 `toUTCTime` 不是。原因是 `toCalendarTime` 依赖本地时区返回不同的结果，但是针对相同的 `ClockTime`，`toUTCTime` 将始终返回相同的结果。

很容易改变一个 `CalendarTime` 的值

```

ghci> nowCal {ctYear = 1960}
CalendarTime {ctYear = 1960, ctMonth = August, ctDay = 18, ctHour = 12, ctMin = 10, ctSec = 35, ctPicosec = 804267000000, ctWDay = Monday, ctYD
ghci> (\(TOD sec _) -> sec) (toClockTime nowCal)
1219079435
ghci> (\(TOD sec _) -> sec) (toClockTime (nowCal {ctYear = 1960}))
-295685365

```

此处，先将之前的 `CalendarTime` 年份修改为 1960。然后用 `toClockTime` 将其初始值转换为一个 `ClockTime`，接着转换新值，以便观察其差别。注意新值在转换为 `ClockTime` 后显示了一个负的秒数。这是意料中的，`ClockTime` 表示的是 UTC 时间 1970 年 1 月 1 日午夜之后的秒数。

也可以像这样手工创建 `CalendarTime`：

```

ghci> let newCT = CalendarTime 2010 January 15 12 30 0 0 Sunday 0 "UTC" 0 False
ghci> newCT
CalendarTime {ctYear = 2010, ctMonth = January, ctDay = 15, ctHour = 12, ctMin = 30, ctSec = 0, ctPicosec = 0, ctWDay = Sunday, ctYDay = 0, ctTZName = UTC, ctTZ = 0, ctIsDST = False}
ghci> (\(TOD sec _) -> sec) (toClockTime newCT)
1263558600

```

注意，尽管 2010 年 1 月 15 日并不是一个周日 – 并且也不是一年中的第 0 天 – 系统可以很好的处理这些情况。实际上，如果将其转换为 `ClockTime` 后再转回 `CalendarTime`，你将发现这些域已经被正确的处理了。

```
ghci> toUTCTime . toClockTime $ newCT
CalendarTime {ctYear = 2010, ctMonth = January, ctDay = 15, ctHour = 12, ctMin = 30, ctSec = 0, ctPicoSec = 0, ctWDay = Friday, ctYDay = 14, ctTZName = UTC, ctIsDST = False}
```

## ClockTime 的 TimeDiff

以对人类友好的方式难于处理 `ClockTime` 值之间的差异，`System.Time` 模块包括了一个 `TimeDiff` 类型。`TimeDiff` 用于方便的处理这些差异。其定义如下：

```
data TimeDiff = TimeDiff
  {tdYear :: Int,
   tdMonth :: Int,
   tdDay :: Int,
   tdHour :: Int,
   tdMin :: Int,
   tdSec :: Int,
   tdPicoSec :: Integer}
```

`diffClockTimes` 和 `addToClockTime` 两个函数接收一个 `ClockTime` 和一个 `TimeDiff` 并在内部将 `ClockTime` 转换为一个 UTC 时区的 `CalendarTime`，在其上执行 `TimeDiff`，最后将结果转换回一个 `ClockTime`。

看看它怎样工作：

```
ghci> :module System.Time
ghci> let feb5 = toClockTime $ CalendarTime 2008 February 5 0 0 0 0 Sunday 0 "UTC" 0 False
Loading package old-locale-1.0.0.0 ... linking ... done.
Loading package old-time-1.0.0.0 ... linking ... done.
ghci> feb5
Mon Feb  4 18:00:00 CST 2008
ghci> addToClockTime (TimeDiff 0 1 0 0 0 0 0) feb5
Tue Mar  4 18:00:00 CST 2008
ghci> toUTCTime $ addToClockTime (TimeDiff 0 1 0 0 0 0 0) feb5
CalendarTime {ctYear = 2008, ctMonth = March, ctDay = 5, ctHour = 0, ctMin = 0, ctSec = 0, ctPicoSec = 0, ctWDay = Wednesday, ctYDay = 64, ctTZName = UTC, ctIsDST = False}
ghci> let jan30 = toClockTime $ CalendarTime 2009 January 30 0 0 0 0 Sunday 0 "UTC" 0 False
ghci> jan30
Thu Jan 29 18:00:00 CST 2009
ghci> addToClockTime (TimeDiff 0 1 0 0 0 0 0) jan30
Sun Mar  1 18:00:00 CST 2009
ghci> toUTCTime $ addToClockTime (TimeDiff 0 1 0 0 0 0 0) jan30
CalendarTime {ctYear = 2009, ctMonth = March, ctDay = 2, ctHour = 0, ctMin = 0, ctSec = 0, ctPicoSec = 0, ctWDay = Monday, ctYDay = 60, ctTZName = UTC, ctIsDST = False}
ghci> diffClockTimes jan30 feb5
TimeDiff {tdYear = 0, tdMonth = 0, tdDay = 0, tdHour = 0, tdMin = 0, tdSec = 31104000, tdPicoSec = 0}
ghci> normalizeTimeDiff $ diffClockTimes jan30 feb5
TimeDiff {tdYear = 0, tdMonth = 12, tdDay = 0, tdHour = 0, tdMin = 0, tdSec = 0, tdPicoSec = 0}
```

首先我们生成一个 `ClockTime` 表示 UTC 时间 2008 年 2 月 5 日。注意，若你的时区不是 UTC，按你本地时区的格式，当其被显示的时候可能是 2 月 4 日晚。

其次，我们用 `addToClockTime` 在其上加一个月。2008 是闰年，但系统可以正确的处理，然后我们得到了一个月后的相同日期。使用 `toUTCTime`，我们可以看到以 UTC 时间表示的结果。

第二个实验，设定一个表示 UTC 时间 2009 年 1 月 30 日午夜的时间。2009 年不是闰年，所以我们可能很好奇其加上一个月是什么结果。因为 2009 年没有 2 月 29 日和 2 月 30 日，所以我们得到了 3 月 2 日。

最后，我们可以看到 `diffClockTimes` 怎样通过两个 `ClockTime` 值得到一个 `TimeDiff`，尽管其只包含秒和皮秒。`normalizeTimeDiff` 函数接受一个 `TimeDiff` 将其重新按照人类的习惯格式化。

 v: latest ▾

## 文件修改日期

很多程序需要找出某些文件的最后修改日期。ls 和图形化的文件管理器是典型的需要显示文件最后变更时间的程序。System.Directory 模块包含一个跨平台的 getModificationTime 函数。其接受一个文件名，返回一个表示文件最后变更日期的 ClockTime。例如：

```
ghci> :module System.Directory
ghci> getModificationTime "/etc/passwd"
Loading package old-locale-1.0.0.0 ... linking ... done.
Loading package old-time-1.0.0.0 ... linking ... done.
Loading package filepath-1.1.0.0 ... linking ... done.
Loading package directory-1.0.0.0 ... linking ... done.
Fri Aug 15 08:29:48 CDT 2008
```

POSIX 平台不仅维护变更时间 (被称为 mtime)，还有最后读或写访问时间 (atime) 以及最后状态变更时间 (ctime)。这是 POSIX 平台独有的，所以跨平台的 System.Directory 模块无法访问它。取而代之，需要使用 System.Posix.Files 模块中的函数。下面有一个例子：

```
-- file: ch20/posixtime.hs
-- posixtime.hs

import System.Posix.Files
import System.Time
import System.Posix.Types

-- / Given a path, returns (atime, mtime, ctime)
getTimes :: FilePath -> IO (ClockTime, ClockTime, ClockTime)
getTimes fp =
    do stat <- getFileStatus fp
       return (toct (accessTime stat),
               toct (modificationTime stat),
               toct (statusChangeTime stat))

-- / Convert an EpochTime to a ClockTime
toct :: EpochTime -> ClockTime
toct et =
    TOD (truncate (toRational et)) 0
```

注意对 getFileStatus 的调用。这个调用直接映射到 C 语言的 stat() 函数。其返回一个包含了大量不同种类信息的值，包括文件类型、权限、属主、组、和我们感性去的三种时间值。System.Posix.Files 提供了 accessTime 等多个函数，可以将我们感兴趣的时间从 getFileStatus 返回的 FileStatus 类型中提取出来。

accessTime 等函数返回一个 POSIX 平台特有的类型，称为 EpochTime，可以通过 toct 函数转换 ClockTime。System.Posix.Files 模块同样提供了 setFileTimes 函数，以设置文件的 atime 和 mtime。[45]

## 延伸的例子: 管道

我们已经了解了如何调用外部程序。有时候需要更多的控制。比如获得程序的标准输出、提供输入，甚至将不同的外部程序串起来调用。管道有助于实现所有这些需求。管道经常用在 shell 脚本中。在 shell 中设置一个管道，会调用多个程序。第一个程序的输入会做为第二个程序的输入。其输出又会作为第三个的输入，以此类推。最后一个程序通常将输出打印到终端，或者写入文件。下面是一个 POSIX shell 的例子，演示如何使用管道：

```
$ ls /etc | grep 'm.*ap' | tr a-z A-Z
IDMAPD.CONF
MAILCAP
MAILCAP.ORDER
MEDIAPRM
TERMCAP
```

这条命令运行了三个程序，使用管道在它们之间传输数据。它以 ls/etc 开始，输出是 /etc 目录下全部文件和目录的列表。ls 的输出被作为 grep 的输入。我们想 grep 输入一条正则则使其只输出以 'm' 开头并且在某处包含 "ap" 的行。最后，其结果被传入 tr。我们给 tr 设置一个选项，使其将所有字符转换为大写。tr 的输出没有特殊的去处，所以直接在屏幕显示。

这种情况下，程序之间的管道线路由 shell 设置。我们可以使用 Haskell 中的 POSIX 工具实现同样的事情。

 v: latest ▾



在讲解如何实现之前，要提醒你一下，`System.Posix` 模块提供的是很低阶的 Unix 系统接口。无论使用何种编程语言，这些接口都可以相互组合，组合的结果也可以相互组合。这些低阶接口的完整性质可以用一整本书来讨论，这章中我们只会简单介绍。

## 使用管道做重定向

POSIX 定义了一个函数用于创建管道。这个函数返回两个文件描述符 (FD)，与 Haskell 中的句柄概念类似。一个 FD 用于读端，另一个用于写端。任何从写端写入的东西，都可以从读端读取。这些数据就是“通过管道推送”的。在 Haskell 中，你可以通过 `createPipe` 使用这个接口。

在外部程序之间传递数据之前，要做的第一步是建立一个管道。同时还要将一个程序的输出重定向到管道，并将管道做为另一个程序的输入。Haskell 的 `dupTo` 函数就是做这个的。其接收一个 FD 并将其拷贝为另一个 FD 号。POSIX 的标准输入、标准输出和标准错误的 FD 分别被预定义为 0, 1, 2。将管道的某一端设置为这些 FD 号，我们就可以有效的重定向程序的输入和输出。

不过还有问题需要解决。我们不能简单的只是在某个调用比如 `rawSystem` 之前使用 `dupTo`，因为这回混淆我们的 Haskell 主程序的输入和输出。此外，`rawSystem` 会一直阻塞直到被调用的程序执行完毕，这让我们无法启动并行执行的进程。为了解决这个问题，可以使用 `forkProcess`。这是一个很特殊的函数。它实际上生成了一份当前进程的拷贝，并使这两份进程同时运行。Haskell 的 `forkProcess` 函数接收一个函数，使其在新进程（称为子进程）中运行。我们让这个函数调用 `dupTo`。之后，其调用 `executeFile` 调用真正希望执行的命令。这同样也是一个特殊的函数：如果一切顺利，他将不会返回。这是因为 `executeFile` 使用一个不同的程序替换了当前执行的进程。最后，初始的 Haskell 进程调用 `getProcessStatus` 以等待子进程结束，并获得其状态码。

在 POSIX 系统中，无论何时你执行一条命令，不关是在命令上敲 `ls` 还是在 Haskell 中使用 `rawSystem`，其内部机理都是调用 `forkProcess`，`executeFile`，和 `getProcessStatus` (或是它们对应的 C 函数)。为了使用管道，我们复制了系统启动程序的进程，并且加入了一些调用和重定向管道的步骤。

还有另外一些辅助步骤需要注意。当调用 `forkProcess` 时，“几乎”和程序有关的一切都被复制 [46]。包括所有已经打开的文件描述符（句柄）。程序通过检查管道是否传来文件结束符判断数据接收是否结束。写端进程关闭管道时，读端程序将收到文件结束符。然而，如果同一个写端文件描述符在多个进程中同时存在，则文件结束符要在所有进程中都被关闭才会发送文件结束符。因此，我们必须在子进程中追踪打开了哪些文件描述符，以便关闭它们。同样，也必须尽早在主进程中关闭子进程的写管道。

下面是一个用 Haskell 编写的管道系统的初始实现：

```
-- file: ch20/RunProcessSimple.hs

{-# OPTIONS_GHC -XDatatypeContexts #-}
{-# OPTIONS_GHC -XTypeSynonymInstances #-}
{-# OPTIONS_GHC -XFlexibleInstances #-}

module RunProcessSimple where

--import System.Process
import Control.Concurrent
import Control.Concurrent.MVar
import System.IO
import System.Exit
import Text.Regex.Posix
import System.Posix.Process
import System.Posix.IO
import System.Posix.Types
import Control.Exception

{- | The type for running external commands. The first part
of the tuple is the program name. The list represents the
command-line parameters to pass to the command. -}
type SysCommand = (String, [String])

{- | The result of running any command -}
data CommandResult = CommandResult {
  cmdOutput :: IO String,      -- ^ IO action that yields the output
  getExitStatus :: IO ProcessStatus -- ^ IO action that yields exit result
}

{- | The type for handling global lists of FDs to always close in the clients
-}
type CloseFDs = MVar [Fd]

{- | Class representing anything that is a runnable command -}
```

```

class CommandLike a where
  {- | Given the command and a String representing input,
       invokes the command. Returns a String
       representing the output of the command. -}
  invoke :: a -> CloseFDs -> String -> IO CommandResult

-- Support for running system commands
instance CommandLike SysCommand where
  invoke (cmd, args) closefds input =
    do -- Create two pipes: one to handle stdin and the other
       -- to handle stdout. We do not redirect stderr in this program.
       (stdinread, stdinwrite) <- createPipe
       (stdoutread, stdoutwrite) <- createPipe

       -- We add the parent FDs to this list because we always need
       -- to close them in the clients.
       addCloseFDs closefds [stdinwrite, stdoutread]

       -- Now, grab the closed FDs list and fork the child.
       childPID <- withMVar closefds (\fds ->
         forkProcess (child fds stdinread stdoutwrite))

       -- Now, on the parent, close the client-side FDs.
       closeFd stdinread
       closeFd stdoutwrite

       -- Write the input to the command.
       stdinhdl <- fdToHandle stdinwrite
       forkIO $ do hPutStr stdinhdl input
                  hClose stdinhdl

       -- Prepare to receive output from the command
       stdouthdl <- fdToHandle stdoutread

       -- Set up the function to call when ready to wait for the
       -- child to exit.
       let waitfunc =
         do status <- getProcessStatus True False childPID
         case status of
           Nothing -> fail $ "Error: Nothing from getProcessStatus"
           Just ps -> do removeCloseFDs closefds
                        [stdinwrite, stdoutread]
                        return ps
       return $ CommandResult {cmdOutput = hGetContents stdouthdl,
                              getExitStatus = waitfunc}

-- Define what happens in the child process
where child closefds stdinread stdoutwrite =
  do -- Copy our pipes over the regular stdin/stdout FDs
     dupTo stdinread stdInput
     dupTo stdoutwrite stdOutput

     -- Now close the original pipe FDs
     closeFd stdinread
     closeFd stdoutwrite

     -- Close all the open FDs we inherited from the parent
     mapM_ (\fd -> catch (closeFd fd) (\(SomeException e) -> return ())) closefds

     -- Start the program
     executeFile cmd True args Nothing

-- Add FDs to the list of FDs that must be closed post-fork in a child
addCloseFDs :: CloseFDs -> [Fd] -> IO ()
addCloseFDs closefds newfds =
  modifyMVar_ closefds (\oldfds -> return $ oldfds ++ newfds)

-- Remove FDs from the list
removeCloseFDs :: CloseFDs -> [Fd] -> IO ()
removeCloseFDs closefds removethem =
  modifyMVar_ closefds (\fdlist -> return $ procfdlist fdlist removethem)

```



```

where
procfdlist fdlist [] = fdlist
procfdlist fdlist (x:xs) = procfdlist (removefd fdlist x) xs

-- We want to remove only the first occurrence of any given fd
removefd [] _ = []
removefd (x:xs) fd
  | fd == x = xs
  | otherwise = x : removefd xs fd

{- | Type representing a pipe. A 'PipeCommand' consists of a source
and destination part, both of which must be instances of
'CommandLike'. -}
data (CommandLike src, CommandLike dest) =>
  PipeCommand src dest = PipeCommand src dest

{- | A convenient function for creating a 'PipeCommand'. -}
(-|-) :: (CommandLike a, CommandLike b) => a -> b -> PipeCommand a b
(-|-) = PipeCommand

{- | Make 'PipeCommand' runnable as a command -}
instance (CommandLike a, CommandLike b) =>
  CommandLike (PipeCommand a b) where
  invoke (PipeCommand src dest) closefds input =
    do res1 <- invoke src closefds input
       output1 <- cmdOutput res1
       res2 <- invoke dest closefds output1
       return $ CommandResult (cmdOutput res2) (getEC res1 res2)

{- | Given two 'CommandResult' items, evaluate the exit codes for
both and then return a "combined" exit code. This will be ExitSuccess
if both exited successfully. Otherwise, it will reflect the first
error encountered. -}
getEC :: CommandResult -> CommandResult -> IO ProcessStatus
getEC src dest =
  do sec <- getExitStatus src
     dec <- getExitStatus dest
     case sec of
       Exited ExitSuccess -> return dec
       x -> return x

{- | Execute a 'CommandLike'. -}
runIO :: CommandLike a => a -> IO ()
runIO cmd =
  do -- Initialize our closefds list
     closefds <- newMVar []

     -- Invoke the command
     res <- invoke cmd closefds []

     -- Process its output
     output <- cmdOutput res
     putStr output

     -- Wait for termination and get exit status
     ec <- getExitStatus res
     case ec of
       Exited ExitSuccess -> return ()
       x -> fail $ "Exited: " ++ show x

```

在研究这个函数的运作原理之前，让我们先来在 `ghci` 里面尝试运行它一下：

```

ghci> runIO $ ("pwd", []::[String])
/Users/Blade/sandbox

ghci> runIO $ ("ls", ["/usr"])
NX
X11
X11R6
bin
include

```

 v: latest ▾

```
lib
libexec
local
sbin
share
standalone

ghci> runIO $ ("ls", ["/usr"]) -|- ("grep", ["^l"])
lib
libexec
local

ghci> runIO $ ("ls", ["/etc"]) -|- ("grep", ["m.*ap"]) -|- ("tr", ["a-z", "A-Z"])
COM. APPLE. SCREENSHARING. AGENT. LAUNCHD
```

我们从一个简单的命令 `pwd` 开始，它会打印当前工作目录。我们将 `[]` 做为参数列表，因为 `pwd` 不需要任何参数。由于使用了类型类，Haskell 无法自动推导出 `[]` 的类型，所以我们说明其类型为字符串组成的列表。

下面是一个更复杂些的例子。我们执行了 `ls`，将其输出传入 `grep`。最后我们通过管道，调用了一个与本节开始处 `shell` 内置管道的例子中相同的命令。不像 `shell` 中那样舒服，但是相对于 `shell` 我们的程序始终相对简单。

让我们读一下程序。起始处的 `OPTIONS_GHC` 语句，作用与 `ghc` 或 `ghci` 开始时传入 `-fglasgow-exts` 参数相同。我们使用了一个 GHC 扩展，以允许使用 `(String, [String])` 类型作为一个类型类的实例 [47]。将此类声明加入源码文件，就不用每次调用这个模块的时候都要记得手工打开编译器开关。

在载入了所需模块之后，定义了一些类型。首先，定义 `type SysCommand = (String, [String])` 作为一个别名。这是系统将接收并执行的命令的类型。例子中的每条命令都要用到这个类型的数据。`CommandResult` 命令用于表示给定命令的执行结果，`CloseFDs` 用于表示必须在新子进程中关闭的文件描述符列表。

接着，定义一个类称为 `CommandLike`。这个类用来跑“东西”，这个“东西”可以是独立的程序，可以是两个程序之间的管道，未来也可以跑纯 Haskell 函数。任何一个类型想为这个类的成员，只需实现一个函数 `- invoke`。这将允许以 `runIO` 启动一个独立命令或者一个管道。这在定义管道时也很有用，因为我们可以拥有某个管道的读写两端的完整调用栈。

我们的管道基础设施将使用字符串在进程间传递数据。我们将通过 `hGetContents` 获得 Haskell 在延迟读取方面的优势，并使用 `forkIO` 在后台写入。这种设计工作得不错，尽管传输速度不像将两个进程的管道读写端直接连接起来那样快 [48]。但这让实现很简单。我们仅需要小心，不要做任何会让整个字符串被缓冲的操作，把接下来的工作完全交给 Haskell 的延迟特性。

接下来，为 `SysCommand` 定义一个 `CommandLike` 实例。我们创建两个管道：一个用来作为新进程的标准输入，另一个用于其标准输出。将产生两个读端两个写端，四个文件描述符。我们将要在子进程中关闭的文件描述符加入列表。这包括子进程标准输入的写端，和子进程标准输出的读端。接着，我们 `fork` 出子进程。然后可以在父进程中关闭相关的子进程文件描述符。`fork` 之前不能这样做，因为那时子进程还不可用。获取 `stdinwrite` 的句柄，并通过 `forkIO` 启动一个现成向其写入数据。接着定义 `waitfunc`，其中定义了调用这在准备好等待子进程结束时要执行的动作。同时，子进程使用 `dupTo`，关闭其不需要的文件描述符。并执行命令。

然后定义一些工具函数用来管理文件描述符。此后，定义一些工具用于建立管道。首先，定义一个新类型 `PipeCommand`，其有源和目的两个属性。源和目的都必须是 `CommandLike` 的成员。为了方便，我们还定义了 `-|-` 操作符。然后使 `PipeCommand` 成为 `CommandLike` 的实例。它调用第一个命令并获得输出，将其传入第二个命令。之后返回第二个命令的输出，并调用 `getExitStatus` 函数等待命令执行结束并检查整组命令执行之后的状态码。

最后以定义 `runIO` 结束。这个函数建立了需要在子进程中关闭的 FDS 列表，执行程序，显示输出，并检查其退出状态。

## 更好的管道

上个例子中解决了一个类似 `shell` 的管道系统的基本需求。但是为它加上下面这些特点之后就更好了：

- 支持更多的 `shell` 语法。
- 使管道同时支持外部程序和正规 Haskell 函数，并使二者可以自由的混合使用。
- 以易于 Haskell 程序利用的方式返回标准输出和退出状态码。

幸运的是，支持这些功能的代码片段已经差不多就位了。只需要为 `CommandLike` 多加入几个实例，以及一些类似 `runIO` 的函数。下面是修订后实现了以上功能的例子代码：

```
-- file: ch20/RunProcess.hs
{-# OPTIONS_GHC -XDatatypeContexts #-}
{-# OPTIONS_GHC -XTypeSynonymInstances #-}
```

 v: latest ▾

```

{-# OPTIONS_GHC -XFlexibleInstances #-}

module RunProcess where

import System.Process
import Control.Concurrent
import Control.Concurrent.MVar
import Control.Exception
import System.Posix.Directory
import System.Directory(setCurrentDirectory)
import System.IO
import System.Exit
import Text.Regex
import System.Posix.Process
import System.Posix.IO
import System.Posix.Types
import Data.List
import System.Posix.Env(getEnv)

{- | The type for running external commands. The first part
of the tuple is the program name. The list represents the
command-line parameters to pass to the command. -}
type SysCommand = (String, [String])

{- | The result of running any command -}
data CommandResult = CommandResult {
    cmdOutput :: IO String,          -- ^ IO action that yields the output
    getExitStatus :: IO ProcessStatus -- ^ IO action that yields exit result
}

{- | The type for handling global lists of FDs to always close in the clients
-}
type CloseFDs = MVar [Fd]

{- | Class representing anything that is a runnable command -}
class CommandLike a where
    {- | Given the command and a String representing input,
        invokes the command. Returns a String
        representing the output of the command. -}
    invoke :: a -> CloseFDs -> String -> IO CommandResult

-- Support for running system commands
instance CommandLike SysCommand where
    invoke (cmd, args) closefds input =
        do -- Create two pipes: one to handle stdin and the other
            -- to handle stdout. We do not redirect stderr in this program.
            (stdinread, stdinwrite) <- createPipe
            (stdoutread, stdoutwrite) <- createPipe

            -- We add the parent FDs to this list because we always need
            -- to close them in the clients.
            addCloseFDs closefds [stdinwrite, stdoutread]

            -- Now, grab the closed FDs list and fork the child.
            childPID <- withMVar closefds (\fds ->
                forkProcess (child fds stdinread stdoutwrite))

            -- Now, on the parent, close the client-side FDs.
            closeFd stdinread
            closeFd stdoutwrite

            -- Write the input to the command.
            stdinhdl <- fdToHandle stdinwrite
            forkIO $ do hPutStr stdinhdl input
                      hClose stdinhdl

            -- Prepare to receive output from the command
            stdouthdl <- fdToHandle stdoutread

            -- Set up the function to call when ready to wait for the
            -- child to exit.
            let waitfunc =

```

```

do status <- getProcessStatus True False childPID
    case status of
        Nothing -> fail $ "Error: Nothing from getProcessStatus"
        Just ps -> do removeCloseFDs closefds
                        [stdinwrite, stdoutread]
                        return ps
return $ CommandResult {cmdOutput = hGetContents stdouthdl,
                        getExitStatus = waitfunc}

-- Define what happens in the child process
where child closefds stdinread stdoutwrite =
    do -- Copy our pipes over the regular stdin/stdout FDs
        dupTo stdinread stdInput
        dupTo stdoutwrite stdOutput

        -- Now close the original pipe FDs
        closeFd stdinread
        closeFd stdoutwrite

        -- Close all the open FDs we inherited from the parent
        mapM_ (\fd -> catch (closeFd fd) (\(SomeException e) -> return ())) closefds

        -- Start the program
        executeFile cmd True args Nothing

{- | An instance of 'CommandLike' for an external command. The String is
passed to a shell for evaluation and invocation. -}
instance CommandLike String where
    invoke cmd closefds input =
        do -- Use the shell given by the environment variable SHELL,
            -- if any. Otherwise, use /bin/sh
            esh <- getEnv "SHELL"
            let sh = case esh of
                Nothing -> "/bin/sh"
                Just x -> x
            invoke (sh, ["-c", cmd]) closefds input

-- Add FDs to the list of FDs that must be closed post-fork in a child
addCloseFDs :: CloseFDs -> [Fd] -> IO ()
addCloseFDs closefds newfds =
    modifyMVar_ closefds (\oldfds -> return $ oldfds ++ newfds)

-- Remove FDs from the list
removeCloseFDs :: CloseFDs -> [Fd] -> IO ()
removeCloseFDs closefds removethem =
    modifyMVar_ closefds (\fdlist -> return $ procfldlist fdlist removethem)

where
    procfldlist fdlist [] = fdlist
    procfldlist fdlist (x:xs) = procfldlist (removefd fdlist x) xs

-- We want to remove only the first occurrence of any given fd
removefd [] _ = []
removefd (x:xs) fd
    | fd == x = xs
    | otherwise = x : removefd xs fd

-- Support for running Haskell commands
instance CommandLike (String -> IO String) where
    invoke func _ input =
        return $ CommandResult (func input) (return (Exited ExitSuccess))

-- Support pure Haskell functions by wrapping them in IO
instance CommandLike (String -> String) where
    invoke func = invoke iofunc
    where iofunc :: String -> IO String
          iofunc = return . func

-- It's also useful to operate on lines. Define support for line-based
-- functions both within and without the IO monad.

instance CommandLike ([String] -> IO [String]) where

```

```

    invoke func _ input =
        return $ CommandResult linedfunc (return (Exited ExitSuccess))
    where linedfunc = func (lines input) >>= (return . unlines)

instance CommandLike ([String] -> [String]) where
    invoke func = invoke (unlines . func . lines)

{- | Type representing a pipe. A 'PipeCommand' consists of a source
and destination part, both of which must be instances of
'CommandLike'. -}
data (CommandLike src, CommandLike dest) =>
    PipeCommand src dest = PipeCommand src dest

{- | A convenient function for creating a 'PipeCommand'. -}
(-|-) :: (CommandLike a, CommandLike b) => a -> b -> PipeCommand a b
(-|-) = PipeCommand

{- | Make 'PipeCommand' runnable as a command -}
instance (CommandLike a, CommandLike b) =>
    CommandLike (PipeCommand a b) where
    invoke (PipeCommand src dest) closefds input =
        do res1 <- invoke src closefds input
           output1 <- cmdOutput res1
           res2 <- invoke dest closefds output1
           return $ CommandResult (cmdOutput res2) (getEC res1 res2)

{- | Given two 'CommandResult' items, evaluate the exit codes for
both and then return a "combined" exit code. This will be ExitSuccess
if both exited successfully. Otherwise, it will reflect the first
error encountered. -}
getEC :: CommandResult -> CommandResult -> IO ProcessStatus
getEC src dest =
    do sec <- getExitStatus src
       dec <- getExitStatus dest
       case sec of
           Exited ExitSuccess -> return dec
           x -> return x

{- | Different ways to get data from 'run'.

* IO () runs, throws an exception on error, and sends stdout to stdout

* IO String runs, throws an exception on error, reads stdout into
  a buffer, and returns it as a string.

* IO [String] is same as IO String, but returns the results as lines

* IO ProcessStatus runs and returns a ProcessStatus with the exit
  information. stdout is sent to stdout. Exceptions are not thrown.

* IO (String, ProcessStatus) is like IO ProcessStatus, but also
  includes a description of the last command in the pipe to have
  an error (or the last command, if there was no error)

* IO Int returns the exit code from a program directly. If a signal
  caused the command to be reaped, returns 128 + SIGNUM.

* IO Bool returns True if the program exited normally (exit code 0,
  not stopped by a signal) and False otherwise.

-}
class RunResult a where
    {- | Runs a command (or pipe of commands), with results presented
    in any number of different ways. -}
    run :: (CommandLike b) => b -> a

-- | Utility function for use by 'RunResult' instances
setUpCommand :: CommandLike a => a -> IO CommandResult
setUpCommand cmd =
    do -- Initialize our closefds list
       closefds <- newMVar []

```

```

    -- Invoke the command
    invoke cmd closefds []

instance RunResult (IO ()) where
    run cmd = run cmd >>= checkResult

instance RunResult (IO ProcessStatus) where
    run cmd =
        do res <- setUpCommand cmd

        -- Process its output
        output <- cmdOutput res
        putStr output

        getExitStatus res

instance RunResult (IO Int) where
    run cmd = do rc <- run cmd
               case rc of
                 Exited (ExitSuccess) -> return 0
                 Exited (ExitFailure x) -> return x
                 (Terminated x _) -> return (128 + (fromIntegral x))
                 Stopped x -> return (128 + (fromIntegral x))

instance RunResult (IO Bool) where
    run cmd = do rc <- run cmd
               return ((rc::Int) == 0)

instance RunResult (IO [String]) where
    run cmd = do r <- run cmd
               return (lines r)

instance RunResult (IO String) where
    run cmd =
        do res <- setUpCommand cmd

        output <- cmdOutput res

        -- Force output to be buffered
        evaluate (length output)

        ec <- getExitStatus res
        checkResult ec
        return output

checkResult :: ProcessStatus -> IO ()
checkResult ps =
    case ps of
      Exited (ExitSuccess) -> return ()
      x -> fail (show x)

{- | A convenience function. Refers only to the version of 'run'
that returns @IO ()@. This prevents you from having to cast to it
all the time when you do not care about the result of 'run'.
-}
runIO :: CommandLike a => a -> IO ()
runIO = run

-----

-- Utility Functions
-----

cd :: FilePath -> IO ()
cd = setCurrentDirectory

{- | Takes a string and sends it on as standard output.
The input to this function is never read. -}
echo :: String -> String -> String
echo inp _ = inp

-- | Search for the regexp in the lines. Return those that match.
grep :: String -> [String] -> [String]
grep pat = filter (ismatch regex)

```



```

where regex = mkRegex pat
      ismatch r inp = case matchRegex r inp of
                        Nothing -> False
                        Just _  -> True

{- | Creates the given directory. A value of 0o755 for mode would be typical.
An alias for System.Posix.Directory.createDirectory. -}
mkdir :: FilePath -> FileMode -> IO ()
mkdir = createDirectory

{- | Remove duplicate lines from a file (like Unix uniq).
Takes a String representing a file or output and plugs it through
lines and then nub to uniqify on a line basis. -}
uniq :: String -> String
uniq = unlines . nub . lines

-- | Count number of lines. wc -l
wcL, wcW :: [String] -> [String]
wcL inp = [show (genericLength inp :: Integer)]

-- | Count number of words in a file (like wc -w)
wcW inp = [show ((genericLength $ words $ unlines inp) :: Integer)]

sortLines :: [String] -> [String]
sortLines = sort

-- | Count the lines in the input
countLines :: String -> IO String
countLines = return . (++) "\n" . show . length . lines

```

主要改变是：

- String 的 CommandLike 实例，以便在 shell 中对字符串进行求值和调用。
- String -> IO String 的实例，以及其它几种相关类型的实现。这样就可以像处理命令一样处理 Haskell 函数。
- RunResult 类型类，定义了一个 run 函数，其可以用多种不同方式返回命令的相关信息。
- 一些工具函数，提供了用 Haskell 实现的类 Unix shell 命令。

现在来试试这些新特性。首先确定一下之前例子中的命令是否还能工作。然后，使用新的类 shell 语法运行一下。

```

ghci> :load RunProcess.hs
[1 of 1] Compiling RunProcess      ( RunProcess.hs, interpreted )
Ok, modules loaded: RunProcess.

ghci> runIO $ ("ls", ["/etc"]) -|- ("grep", ["m.*ap"]) -|- ("tr", ["a-z", "A-Z"])
Loading package array-0.5.0.0 ... linking ... done.
Loading package deepseq-1.3.0.2 ... linking ... done.
Loading package bytestring-0.10.4.0 ... linking ... done.
Loading package containers-0.5.5.1 ... linking ... done.
Loading package filepath-1.3.0.2 ... linking ... done.
Loading package old-locale-1.0.0.6 ... linking ... done.
Loading package time-1.4.2 ... linking ... done.
Loading package unix-2.7.0.1 ... linking ... done.
Loading package directory-1.2.1.0 ... linking ... done.
Loading package process-1.2.0.0 ... linking ... done.
Loading package transformers-0.3.0.0 ... linking ... done.
Loading package mtl-2.1.3.1 ... linking ... done.
Loading package regex-base-0.93.2 ... linking ... done.
Loading package regex-posix-0.95.2 ... linking ... done.
Loading package regex-compatible-0.95.1 ... linking ... done.
COM. APPLE. SCREENSHARING. AGENT. LAUNCHED

ghci> runIO $ "ls /etc" -|- "grep 'm.*ap'" -|- "tr a-z A-Z"
COM. APPLE. SCREENSHARING. AGENT. LAUNCHED

```

输入起来容易多了。试试使用 Haskell 实现的 grep 来试一下其它的新特性：

```

ghci> runIO $ "ls /usr/local/bin" -|- grep "m.*ap" -|- "tr a-z A-Z"
DUMPCAP
MERGECAP

```

 v: latest ▾

```
NMAP

ghci> run $ "ls /usr/local/bin" <-|> grep "m.*ap" <-|> "tr a-z A-Z" :: IO String
"DUMPCAP\nMERGECAP\nNMAP\n"

ghci> run $ "ls /usr/local/bin" <-|> grep "m.*ap" <-|> "tr a-z A-Z" :: IO [String]
["DUMPCAP","MERGECAP","NMAP"]

ghci> run $ "ls /usr" :: IO String
"X11\nX11R6\nbin\ninclude\nlib\nlibexec\nlocal\nsbin\nshare\nstandalone\ntexbin\n"

ghci> run $ "ls /usr" :: IO Int
X11
X11R6
bin
include
lib
libexec
local
sbin
share
standalone
texbin
0

ghci> runIO $ echo "Line1\nHi, test\n" <-|> "tr a-z A-Z" <-|> sortLines
HI, TEST
LINE1
```

## 关于管道，最后说几句

我们开发了一个精巧的系统。前面时醒过，POSIX 有时会很复杂。另外要强调一下：要始终注意确保先将这些函数返回的字符串求值，然后再尝试获取子进程的退出状态码。子进程经常要等待写出其所有输出之后才能退出，如果搞错了获取输出和退出状态码的顺序，你的程序会挂住。

本章中，我们从零开始开发了一个精简版的 HSH。如果你希望使程序具有这样类 shell 的功能，我们推荐使用 HSH 而非上面开发的例子，因为 HSH 的实现更加优化。HSH 还有一个数量庞大的工具函数集和更多功能，但其背后的代码也更加庞大和复杂。其实例子中很多工具函数的代码我们是直接从 HSH 抄过来的。可以从 <http://software.complete.org/hsh> 访问 HSH 的源码。

## 注

- [43]** 也有一个 `system` 函数，接受单个字符串为参数，并将其传入 shell 解析。我们推荐使用 `rawSystem`，因为某些字符在 shell 中具有特殊含义，可能会导致安全隐患或者意外的行为。
- [44]** 可能有人会注意到 UTC 定义了不规则的闰秒。在 Haskell 使用的 POSIX 标准中，规定了在其表示的时间中，每天必须都是精确的 86,400 秒，所以在执行日常计算时无需担心闰秒。精确的处理闰秒依赖于系统而且复杂，不过通常其可以被解释为一个“长秒”。这个问题大体上只是在执行精确的亚秒级计算时才需要关心。
- [45]** POSIX 系统上通常无法设置 `ctime`。
- [46]** 线程是一个主要例外，其不会被复制，所以说“几乎”。
- [47]** Haskell 社区对这个扩展支持得很好。Hugs 用户可以通过 `hugs -98 +o` 使用。
- [48]** Haskell 的 HSH 库提供了与此相近的 API，使用了更高效（也更复杂）的机构将外部进程使用管道直接连接起来，没有要传给 Haskell 处理的数据。shell 采用了相同的方法，而且这样可以降低处理管道的 CPU 负载。

## 讨论

0条评论

Real World Haskell 中文版


1 登录

推荐

推文

分享

最新发布



开始讨论...

通过以下方式登录

或注册一个 DISQUS 帐号

姓名

来做第一个留言的人吧!

- 在 REAL WORLD HASKELL 中文版 上还有

Pearls of Functional Algorithm Design — Pearls of Functional Algorithm Design

1条评论 • 6年前

forlice — ...

第十章：代码案例学习：解析二进制数据格式 — Real World Haskell 中文版

1条评论 • 4年前

Y — 此处少个"+": instance Show Greymap where show (Greymap w h m \_) = "Greymap " ++ show w ++ "x" ++ show h ++ " " ++ show m

第八章：高效文件处理、正则表达式、文件名匹配 — Real World Haskell 中文版

1条评论 • 4年前

Yutong Zhang —

Real World Haskell 中文版 — Real World Haskell 中文版

4条评论 • 6年前

Jojo — 更新好慢呀