

第二十二章：扩展示例 —— Web 客户端编程

到目前为止，我们已经了解过如何与数据库进行交互、如何进行语法分析（parse）以及如何处理错误。接下来，让我们更进一步，通过引入一个 web 客户端库来将这些知识结合在一起。

在这一章，我们将要构建一个实际的程序：一个播客下载器（podcast downloader），或者叫“播客抓取器”（podcatcher）。这个播客抓取器的概念非常简单，它接受一系列 URL 作为输入，通过下载这些 URL 来得到一些 RSS 格式的 XML 文件，然后在这些 XML 文件里面找到下载音频文件所需的 URL。

播客抓取器常常会让用户通过将 RSS URL 添加到配置文件里面的方法来订阅播客，之后用户就可以定期地进行更新操作：播客抓取器会下载 RSS 文档，对它们进行检查以寻找音频文件的下载链接，并为用户下载所有目前尚未存在的音频文件。

Tip

用户通常将 RSS 文件称之为“广播”（podcast）或是“广播源”（podcast feed），而每个单独的音频文件则是播客的其中一集（episode）。

为了实现具有类似功能的播客抓取器，我们需要以下几样东西：

- 一个用于下载文件的 HTTP 客户端库；
- 一个 XML 分析器；
- 一种能够记录我们感兴趣的广播，并将这些记录永久地储存起来的方法；
- 一种能够永久地记录已下载广播分集（episodes）的方法。

这个列表的后两样可以通过使用 HDBC 设置的数据库来完成，而前两样则可以通过本章介绍的其他库模块来完成。

Tip

本章的代码是专为本书而写的，但这些代码实际上是基于 hpodder —— 一个使用 Haskell 编写的播客抓取器来编写的。hpodder 拥有的特性比本书展示的播客抓取器要多得多，因此本书不太可能详细地对它进行介绍。如果读者对 hpodder 感兴趣的话，可以在 <http://software.complete.org/hpodder> 找到 hpodder 的源代码。

本章的所有代码都是以自成一体的方式来编写的，每段代码都是一个独立的 Haskell 模块，读者可以通过 **ghci** 独立地运行这些模块。本章的最后会写出一段代码，将这些模块全部结合起来，构成一个完整的程序。我们首先要做的就是写出构建播客抓取器需要用到的基本类型。

基本类型

为了构建播客抓取器，我们首先需要思考抓取器需要引入（important）的基本信息有那些。一般来说，抓取器关心的都是记录用户感兴趣的博文的信息，以及那些记录了用户已经看过和处理过的分集的信息。在有需要的时候改变这些信息并不困难，但是因为我们整个抓取器里面都要用到这些信息，所以我们最好还是先定义它们：

```
-- file: ch22/PodTypes.hs
module PodTypes where

data Podcast =
  Podcast {castId :: Integer, -- ^ 这个播客的数字 ID
           castURL :: String -- ^ 这个播客的源 URL
          }
  deriving (Eq, Show, Read)

data Episode =
  Episode {epId :: Integer,    -- ^ 这个分集的数字 ID
           epCast :: Podcast,  -- ^ 这个分集所属播客的 ID
           epURL :: String,    -- ^ 下载这一集所使用的 URL
           epDone :: Bool      -- ^ 记录用户是否已经看过这一集
          }
  deriving (Eq, Show, Read)
```

 v: latest ▾

这些信息将被储存在数据库里面。通过为每个播客和博客的每一集都创建一个独一无二的 ID，程序可以更容易找到分集所属的播客，也可以更容易地从一个特定的播客或者分集里面载入信息，并且更好地应对将来可能会出现“博客 URL 改变”这类情况。

数据库

接下来，我们需要编写代码，以便将信息永久地储存在数据库里面。我们最感兴趣的，就是通过数据库，将 `PodTypes.hs` 文件定义的 Haskell 结构中的数据储存在硬盘里面。并在用户首次运行程序的时候，创建储存数据所需的数据库表。

我们将使用 21 章介绍过的 HDBC 与 Sqlite 数据库进行交互。Sqlite 非常轻量，并且是自包含的（self-contained），因此它对于这个小项目来说简直是再合适不过了。HDBC 和 Sqlite 的安装方法可以在 21 章的《安装 HDBC 和驱动》一节看到。

```
-- file: ch22/PodDB.hs
module PodDB where

import Database.HDBC
import Database.HDBC.Sqlite3
import PodTypes
import Control.Monad(when)
import Data.List(sort)

-- | Initialize DB and return database Connection
connect :: FilePath -> IO Connection
connect fp =
    do dbh <- connectSqlite3 fp
       prepDB dbh
       return dbh

{- | 对数据库进行设置，做好储存数据的准备。

这个程序会创建两个表，并要求数据库引擎为我们检查某些数据的一致性：

* castid 和 epid 都是独一无二的主键（unique primary keys），它们的值不能重复
* castURL 的值也应该是独一无二的
* 在记录分集的表里面，对于一个给定的播客（epcast），每个给定的 URL 或者分集 ID 只能出现一次
-}

prepDB :: IConnection conn => conn -> IO ()
prepDB dbh =
    do tables <- getTables dbh
       when (not ("podcasts" `elem` tables)) $
           do run dbh "CREATE TABLE podcasts (\n
                        \castid INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT,\n
                        \castURL TEXT NOT NULL UNIQUE)" []
              return ()
       when (not ("episodes" `elem` tables)) $
           do run dbh "CREATE TABLE episodes (\n
                        \epid INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT,\n
                        \epcastid INTEGER NOT NULL,\n
                        \epurl TEXT NOT NULL,\n
                        \epdone INTEGER NOT NULL,\n
                        \UNIQUE(epcastid, epurl),\n
                        \UNIQUE(epcastid, epid)" []
              return ()
       commit dbh

{- | 将一个新的播客添加到数据库里面。
在创建播客时忽略播客的 castid，并返回一个包含了 castid 的新对象。

尝试添加一个已经存在的播客将引发一个错误。 -}

addPodcast :: IConnection conn => conn -> Podcast -> IO Podcast
addPodcast dbh podcast =
    handleSql errorHandler $
        do -- Insert the castURL into the table. The database
           -- will automatically assign a cast ID.
           run dbh "INSERT INTO podcasts (castURL) VALUES (?)"
              [toSql (castURL podcast)]
           -- Find out the castID for the URL we just added.
           r <- quickQuery' dbh "SELECT castid FROM podcasts WHERE castURL = ?"
              [toSql (castURL podcast)]
           case r of
```

```

[[x]] -> return $ podcast {castId = fromSql x}
y -> fail $ "addPodcast: unexpected result: " ++ show y
where errorHandler e =
    do fail $ "Error adding podcast; does this URL already exist?\n"
      ++ show e

```

{- | 将一个新的分集添加到数据库里面。 -}

因为这一操作是自动执行而非用户请求执行的，我们将简单地忽略创建重复分集的请求。这样的话，在对播客源进行处理的时候，我们就可以把遇到的所有 URL 到传给这个函数，而不必先检查这个 URL 是否已经存在于数据库当中。

这个函数在创建新的分集时同样不会考虑如何创建新的 ID，因此它也没有必要去考虑如何去获取这个 ID。 -}

```

addEpisode :: IConnection conn => conn -> Episode -> IO ()
addEpisode dbh ep =
    run dbh "INSERT OR IGNORE INTO episodes (epCastId, epURL, epDone) \
            \VALUES (?, ?, ?)"
    [toSql (castId . epCast $ ep), toSql (epURL ep),
     toSql (epDone ep)]
>> return ()

```

{- | 对一个已经存在的播客进行修改。 -}

根据 ID 来查找指定的播客，并根据传入的 Podcast 结构对数据库记录进行修改。 -}

```

updatePodcast :: IConnection conn => conn -> Podcast -> IO ()
updatePodcast dbh podcast =
    run dbh "UPDATE podcasts SET castURL = ? WHERE castId = ?"
    [toSql (castURL podcast), toSql (castId podcast)]
>> return ()

```

{- | 对一个已经存在的分集进行修改。 -}

根据 ID 来查找指定的分集，并根据传入的 episode 结构对数据库记录进行修改。 -}

```

updateEpisode :: IConnection conn => conn -> Episode -> IO ()
updateEpisode dbh episode =
    run dbh "UPDATE episodes SET epCastId = ?, epURL = ?, epDone = ? \
            \WHERE epId = ?"
    [toSql (castId . epCast $ episode),
     toSql (epURL episode),
     toSql (epDone episode),
     toSql (epId episode)]
>> return ()

```

{- | 移除一个播客。 这个操作在执行之前会先移除这个播客已有的所有分集。 -}

```

removePodcast :: IConnection conn => conn -> Podcast -> IO ()
removePodcast dbh podcast =
    do run dbh "DELETE FROM episodes WHERE epcastid = ?"
       [toSql (castId podcast)]
    run dbh "DELETE FROM podcasts WHERE castid = ?"
    [toSql (castId podcast)]
    return ()

```

{- | 获取一个包含所有播客的列表。 -}

```

getPodcasts :: IConnection conn => conn -> IO [Podcast]
getPodcasts dbh =
    do res <- quickQuery' dbh
       "SELECT castid, casturl FROM podcasts ORDER BY castid" []
    return (map convPodcastRow res)

```

{- | 获取特定的广播。 -}

函数在成功执行时返回 Just Podcast；在 ID 不匹配时返回 Nothing。 -}

```

getPodcast :: IConnection conn => conn -> Integer -> IO (Maybe Podcast)
getPodcast dbh wantedId =
    do res <- quickQuery' dbh
       "SELECT castid, casturl FROM podcasts WHERE castid = ?"
       [toSql wantedId]
    case res of
        [x] -> return (Just (convPodcastRow x))
        [] -> return Nothing
        x -> fail $ "Really bad error; more than one podcast with ID"

```

{- | 将 SELECT 语句的执行结果转换为 Podcast 记录 -}

```

convPodcastRow :: [SqlValue] -> Podcast

```

```
convPodcastRow [svId, svURL] =
    Podcast {castId = fromSql svId,
             castURL = fromSql svURL}
convPodcastRow x = error $ "Can't convert podcast row " ++ show x

{- | 获取特定播客的所有分集。 -}
getPodcastEpisodes :: IConnection conn => conn -> Podcast -> IO [Episode]
getPodcastEpisodes dbh pc =
    do r <- quickQuery' dbh
        "SELECT epId, epURL, epDone FROM episodes WHERE epCastId = ?"
        [toSql (castId pc)]
    return (map convEpisodeRow r)
    where convEpisodeRow [svId, svURL, svDone] =
        Episode {epId = fromSql svId, epURL = fromSql svURL,
                  epDone = fromSql svDone, epCast = pc}
```

PodDB 模块定义了连接数据库的函数、创建所需数据库表的函数、将数据添加到数据库里面的函数、查询数据库的函数以及从数据库里面移除数据的函数。以下代码展示了一个与数据库进行交互的 **ghci** 会话，这个会话将在当前目录里面创建一个名为 `poddbtest.db` 的数据库文件，并将广播和分集添加到这个文件里面。

```
ghci> :load PodDB.hs
[1 of 2] Compiling PodTypes      ( PodTypes.hs, interpreted )
[2 of 2] Compiling PodDB        ( PodDB.hs, interpreted )
Ok, modules loaded: PodDB, PodTypes.

ghci> dbh <- connect "poddbtest.db"

ghci> :type dbh
dbh :: Connection

ghci> getTables dbh
["episodes", "podcasts", "sqlite_sequence"]

ghci> let url = "http://feeds.thisamericanlife.org/talpodcast"

ghci> pc <- addPodcast dbh (Podcast {castId=0, castURL=url})
Podcast {castId = 1, castURL = "http://feeds.thisamericanlife.org/talpodcast"}

ghci> getPodcasts dbh
[Podcast {castId = 1, castURL = "http://feeds.thisamericanlife.org/talpodcast"}]

ghci> addEpisode dbh (Episode {epId = 0, epCast = pc, epURL = "http://www.example.com/foo.mp3", epDone = False})

ghci> getPodcastEpisodes dbh pc
[Episode {epId = 1, epCast = Podcast {castId = 1, castURL = "http://feeds.thisamericanlife.org/talpodcast"}, epURL = "http://www.example.com/fo

ghci> commit dbh

ghci> disconnect dbh
```

分析器

在实现了抓取器的数据库部分之后，我们接下来就需要实现抓取器中负责对广播源进行语法分析的部分，这个部分要分析的是一些包含着多种信息的 XML 文件，例子如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<rss xmlns:itunes="http://www.itunes.com/DTDs/Podcast-1.0.dtd" version="2.0">
<channel>
<title>Haskell Radio</title>
<link>http://www.example.com/radio/</link>
<description>Description of this podcast</description>
<item>
<title>Episode 2: Lambdas</title>
<link>http://www.example.com/radio/lambdas</link>
<enclosure url="http://www.example.com/radio/lambdas.mp3"
type="audio/mpeg" length="10485760"/>
</item>
```

 v: latest ▾

```
<item>
<title>Episode 1: Parsec</title>
<link>http://www.example.com/radio/parsec</link>
<enclosure url="http://www.example.com/radio/parsec.mp3"
type="audio/mpeg" length="10485150"/>
</item>
</channel>
</rss>
```

在这些文件里面，我们最关心的是两样东西：广播的标题以及它们的附件（enclosure）URL。我们将使用 **HaXml 工具包** 来对 XML 文件进行分析，以下代码就是这个工具包的源码：

```
-- file: ch22/PodParser.hs
module PodParser where

import PodTypes
import Text.XML.HaXml
import Text.XML.HaXml.Parse
import Text.XML.HaXml.Html.Generate(showattr)
import Data.Char
import Data.List

data PodItem = PodItem {itemtitle :: String,
                        enclosureurl :: String
                      }
    deriving (Eq, Show, Read)

data Feed = Feed {channeltitle :: String,
                  items :: [PodItem]}
    deriving (Eq, Show, Read)

{- | 根据给定的广播和 PodItem，产生一个分集。 -}
item2ep :: Podcast -> PodItem -> Episode
item2ep pc item =
    Episode {epId = 0,
             epCast = pc,
             epURL = enclosureurl item,
             epDone = False}

{- | 从给定的字符串里面分析出数据，给定的名字在有需要的时候会被用在错误消息里面。 -}
parse :: String -> String -> Feed
parse content name =
    Feed {channeltitle = getTitle doc,
          items = getEnclosures doc}

    where parseResult = xmlParse name (stripUnicodeBOM content)
          doc = getContent parseResult

          getContent :: Document -> Content
          getContent (Document _ _ e _) = CElem e

          {- | Some Unicode documents begin with a binary sequence;
              strip it off before processing. -}
          stripUnicodeBOM :: String -> String
          stripUnicodeBOM ('\xef' : '\xbb' : '\xbf' : x) = x
          stripUnicodeBOM x = x

{- | 从文档里面提取出频道部分 (channel part)

注意 HaXml 会将 CFilter 定义为:

> type CFilter = Content -> [Content]
-}
channel :: CFilter
channel = tag "rss" /> tag "channel"

getTitle :: Content -> String
getTitle doc =
    contentToStringDefault "Untitled Podcast"
        (channel /> tag "title" /> txt $ doc)
```

 v: latest ▾

```

getEnclosures :: Content -> [PodItem]
getEnclosures doc =
    concatMap procPodItem $ getPodItems doc
    where procPodItem :: Content -> [PodItem]
          procPodItem item = concatMap (procEnclosure title) enclosure
              where title = contentToStringDefault "Untitled Episode"
                    (keep /> tag "title" /> txt $ item)
                    enclosure = (keep /> tag "enclosure") item

getPodItems :: CFilter
getPodItems = channel /> tag "item"

procEnclosure :: String -> Content -> [PodItem]
procEnclosure title enclosure =
    map makePodItem (showattr "url" enclosure)
    where makePodItem :: Content -> PodItem
          makePodItem x = PodItem {itemtitle = title,
                                   enclosureurl = contentToString [x]}

{- | 将 [Content] 转换为可打印的字符串，
   如果传入的 [Content] 为 []，那么向用户说明此次匹配未成功。 -}
contentToStringDefault :: String -> [Content] -> String
contentToStringDefault msg [] = msg
contentToStringDefault _ x = contentToString x

{- | 将 [Content] 转换为可打印的字符串，并且小心地对它进行反解码 (unescape)。

   一个没有反解码实现的实现可以简单地定义为：

> contentToString = concatMap (show . content)

   因为 HaXml 的反解码操作只能对 Elements 使用，
   我们必须保证每个 Content 都被包裹为 Element，
   然后使用 txt 函数去将 Element 内部的数据提取出来。 -}
contentToString :: [Content] -> String
contentToString =
    concatMap procContent
    where procContent x =
          verbatim $ keep /> txt $ CElem (unesc (fakeElem x))

fakeElem :: Content -> Element
fakeElem x = Elem "fake" [] [x]

unesc :: Element -> Element
unesc = xmlUnEscape stdXmlEscaper

```

让我们好好看看这段代码。它首先定义了两种类型：PodItem 和 Feed。程序会将 XML 文件转换为 Feed，而每个 Feed 可以包含多个 PodItem。此外，程序还提供了一个函数，它可以将 PodItem 转换为 PodTypes.hs 文件中定义的 Episode。

接下来，程序开始定义与语法分析有关的函数。parse 函数接受两个参数，一个是 String 表示的 XML 文本，另一个则是用于展示错误信息的 String 表示的名字，这个函数也会返回一个 Feed。

HaXml 被设计成一个将数据从一种类型转换为另一种类型的“过滤器”，它是一个简单直接的转换操作，可以将 XML 转换为 XML、将 XML 转换为 Haskell 数据、或者将 Haskell 数据转换为 XML。HaXml 拥有一种名为 CFilter 的数据类型，它的定义如下：

```
type CFilter = Content -> [Content]
```

一个 CFilter 接受一个 XML 文档片段 (fragments)，然后返回 0 个或多个片段。CFilter 可能会被要求找出指定标签 (tag) 的所有子标签、所有具有指定名字的标签、XML 文档某一部分包含的文本，又或者其他几样东西 (a number of other things)。操作符 (/>) 可以将多个 CFilter 函数组合在一起。抓取器想要的是那些包围在 <channel> 标签里面的数据，所以我们首先要做的就是找出这些数据。以下是实现这一操作的一个简单的 CFilter：

```
channel = tag "rss" /> tag "channel"
```

当我们将一个文档传递给 channel 函数时，函数会从文档的顶层 (top level) 查找名为 rss 的标签。并在发现这些标签时，找到 channel 标签。

余下的程序也会遵循这一基本方法进行。 `txt` 函数会从标签中提取出文本， 然后通过使用 `CFilter` 函数， 程序可以取得文档的任意部分。

下载

构建抓取器的下一个步骤是完成用于下载数据的模块。 抓取器需要下载两种不同类型的数据： 它们分别是广播的内容以及每个分集的音频。 对于前者， 程序需要对数据进行语法分析并更新数据库； 而对于后者， 程序则需要将数据写入到文件里面并储存到硬盘上。

抓取器将通过 HTTP 服务器进行下载， 所以我们需要使用一个 Haskell HTTP 库。 为了下载广播源， 抓取器需要下载文档、对文档进行语法分析并更新数据库。 对于分集音频， 程序会下载文件、将它写入到硬盘并在数据库里面将该分集标记为“已下载”。 以下是执行这一工作的代码：

```
-- file: ch22/PodDownload.hs
module PodDownload where
import PodTypes
import PodDB
import PodParser
import Network.HTTP
import System.IO
import Database.HDBC
import Data.Maybe
import Network.URI

{- | 下载 URL 。
函数在发生错误时返回 (Left errorMessage) ;
下载成功时返回 (Right doc) 。 -}
downloadURL :: String -> IO (Either String String)
downloadURL url =
    do resp <- simpleHTTP request
    case resp of
        Left x -> return $ Left ("Error connecting: " ++ show x)
        Right r ->
            case rspCode r of
                (2,_,_) -> return $ Right (rspBody r)
                (3,_,_) -> -- A HTTP redirect
                    case findHeader HdrLocation r of
                        Nothing -> return $ Left (show r)
                        Just url' -> downloadURL url'
                _ -> return $ Left (show r)
    where request = Request {rqURI = uri,
                             rqMethod = GET,
                             rqHeaders = [],
                             rqBody = ""}
          uri = fromJust $ parseURI url

{- | 对数据库中的广播源进行更新。 -}
updatePodcastFromFeed :: IConnection conn => conn -> Podcast -> IO ()
updatePodcastFromFeed dbh pc =
    do resp <- downloadURL (castURL pc)
    case resp of
        Left x -> putStrLn x
        Right doc -> updateDB doc

    where updateDB doc =
        do mapM_ (addEpisode dbh) episodes
        commit dbh
        where feed = parse doc (castURL pc)
              episodes = map (item2ep pc) (items feed)

{- | 下载一个分集，并以 String 表示的形式，将储存该分集的文件名返回给调用者。
函数在发生错误时返回一个 Nothing 。 -}
getEpisode :: IConnection conn => conn -> Episode -> IO (Maybe String)
getEpisode dbh ep =
    do resp <- downloadURL (epURL ep)
    case resp of
        Left x -> do putStrLn x
                     return Nothing
        Right doc ->
            do file <- openBinaryFile filename WriteMode
```



```

        hPutStr file doc
        hClose file
        updateEpisode dbh (ep {epDone = True})
        commit dbh
        return (Just filename)
    -- This function ought to apply an extension based on the filetype
where filename = "pod." ++ (show . castId . epCast $ ep) ++ "." ++
    (show (epId ep)) ++ ".mp3"

```

这个函数定义了三个函数：

`downloadURL` 函数对 URL 进行下载，并以 `String` 形式返回它；
`updatePodcastFromFeed` 函数对 XML 源文件进行下载，对文件进行分析，并更新数据库；
`getEpisode` 下载一个给定的分集，并在数据库里面将该分集标记为“已下载”。

Warning

这里使用的 HTTP 库并不会以惰性的方式读取 HTTP 结果，因此在下载诸如广播这样的大文件的时候，这个库可能会消耗掉大量的内容。其他一些 HTTP 库并没有这一限制。我们之所以在这里使用这个有缺陷的库，是因为它稳定、易于安装并且也易于使用。对于正式的 HTTP 需要，我们推荐使用 `mini-http` 库，这个库可以从 `Hackage` 里面获得。

主程序

最后，我们需要编写一个程序来将上面展示的各个部分结合在一起。以下是这个主模块（main module）：

```

-- file: ch22/PodMain.hs
module Main where

import PodDownload
import PodDB
import PodTypes
import System.Environment
import Database.HDBC
import Network.Socket (withSocketsDo)

main = withSocketsDo $ handleSqlError $
    do args <- getArgs
       dbh <- connect "pod.db"
       case args of
           ["add", url] -> add dbh url
           ["update"] -> update dbh
           ["download"] -> download dbh
           ["fetch"] -> do update dbh
                          download dbh
           _ -> syntaxError
       disconnect dbh

add dbh url =
    do addPodcast dbh pc
       commit dbh
    where pc = Podcast {castId = 0, castURL = url}

update dbh =
    do pclist <- getPodcasts dbh
       mapM_ procPodcast pclist
    where procPodcast pc =
        do putStrLn $ "Updating from " ++ (castURL pc)
           updatePodcastFromFeed dbh pc

download dbh =
    do pclist <- getPodcasts dbh
       mapM_ procPodcast pclist
    where procPodcast pc =
        do putStrLn $ "Considering " ++ (castURL pc)
           episodelist <- getPodcastEpisodes dbh pc
           let dleps = filter (\ep -> epDone ep == False)
                               episodelist

```

 v: latest ▾


```

        mapM_ procEpisode dleps
    procEpisode ep =
        do putStrLn $ "Downloading " ++ (epURL ep)
           getEpisode dbh ep

syntaxError = putStrLn
    "Usage: pod command [args]\n\
    \\\n\
    \pod add url      Adds a new podcast with the given URL\n\
    \pod download     Downloads all pending episodes\n\
    \pod fetch        Updates, then downloads\n\
    \pod update       Downloads podcast feeds, looks for new episodes\n"
```

这个程序使用了一个非常简单的命令行解释器，并且这个解释器还包含了一个用于展示命令行语法错误的函数，以及一些用于处理不同命令行参数的小函数。

通过以下命令，可以对这个程序进行编译：

```
ghc --make -O2 -o pod -package HTTP -package HaXml -package network \
    -package HDBC -package HDBC-sqlite3 PodMain.hs
```

你也可以通过《创建包》一节介绍的方法，使用 Cabal 文件来构建这个项目：

```
-- ch23/pod.cabal
Name: pod
Version: 1.0.0
Build-type: Simple
Build-Depends: HTTP, HaXml, network, HDBC, HDBC-sqlite3, base

Executable: pod
Main-Is: PodMain.hs
GHC-Options: -O2
```

除此之外，我们还需要一个简单的 Setup.hs 文件：


```
import Distribution.Simple
main = defaultMain
```

如果你是使用 Cabal 进行构建的话，那么只要运行以下代码即可：

```
runghc Setup.hs configure
runghc Setup.hs build
```


程序的输出将被放到一个名为 dist 的文件及里面。要将程序安装到系统里面的话，可以运行 `run runghc Setup.hs install`。

讨论



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

2条评论 • 6年前
yadsun — 校正：ghc是生成快速本底代码的优化编译器。中 本底 - ->本地

Pearls of Functional Algorithm Design

1条评论 • 6年前
Tonghua Su — where is the content?