

第二章：Socket 和 Syslog

基本网络

本书的前几张，我们讨论了在网络上进行操作的服务。其中两个例子是数据库客户端/服务器和 web 服务。当需要设计新的协议，或者使用没有现成 Haskell 库的协议通信时，将需要使用 Haskell 库函数提供的底层网络工具。

本章中，我们将讨论这些底层工具。网络通讯是个大题目，可以用一整本书来讨论。本章中，我们将展示如何使用 Haskell 应用你已经掌握的底层网络知识。

Haskell 的网络函数几乎始终与常见的 C 函数调用相符。像其他在 C 上层语言一样，你将发现其接口很眼熟。

使用 UDP 通信

UDP 将数据拆散为数据包。其不保证数据到达目的地，也不确保同一个数据包到达的次数。其用校验和的方式确保到达的数据包没有损坏。UDP 适合用在对性能和延迟敏感的应用中，此类场景中系统的整体性能比单个数据包更重要。也可以用在 TCP 表现性能不高的场景，比如发送互不相关的短消息。适合使用 UDP 的系统的例子包括音频和视频会议、时间同步、网络文件系统、以及日志系统。

UDP 客户端例子：syslog

传统 Unix syslog 服务允许程序通过网络向某个负责记录的中央服务器发送日志信息。某些程序对性能非常敏感，而且可能会生成大量日志消息。这样的程序，将日志的开销最小化以确保每条日志被记录更重要。此外，在日志服务器无法访问时，使程序依旧可以操作或许是一种可取的设计。因此，UDP 是一种 syslog 支持的日志传输协议。这种协议比较简单，这里有一个 Haskell 实现的客户端：

```
-- file: ch27/syslogclient.hs
import Data.Bits
import Network.Socket
import Network.BSD
import Data.List
import SyslogTypes

data SyslogHandle =
  SyslogHandle {slSocket :: Socket,
                 slProgram :: String,
                 slAddress :: SockAddr}

openlog :: HostName          -- ^ Remote hostname, or localhost
        -> String           -- ^ Port number or name; 514 is default
        -> String           -- ^ Name to log under
        -> IO SyslogHandle   -- ^ Handle to use for logging
openlog hostname port progname =
  do -- Look up the hostname and port. Either raises an exception
    -- or returns a nonempty list. First element in that list
    -- is supposed to be the best option.
    addrinfos <- getAddrInfo Nothing (Just hostname) (Just port)
    let serveraddr = head addrinfos

    -- Establish a socket for communication
    sock <- socket (addrFamily serveraddr) Datagram defaultProtocol

    -- Save off the socket, program name, and server address in a handle
    return $ SyslogHandle sock progname (addrAddress serveraddr)

syslog :: SyslogHandle -> Facility -> Priority -> String -> IO ()
syslog syslogh fac pri msg =
  sendstr sendmsg
  where code = makeCode fac pri
        sendmsg = "<" ++ show code ++ ">" ++ (slProgram syslogh) ++
                  ": " ++ msg

    -- Send until everything is done
    sendstr :: String -> IO ()
    sendstr [] = return ()
    sendstr omsg = do sent <- sendTo (slSocket syslogh) omsg
```

 v: latest ▾

```

                (slAddress syslogh)
            sendstr (genericDrop sent omsg)

closelog :: SyslogHandle -> IO ()
closelog syslogh = sClose (slSocket syslogh)

{- | Convert a facility and a priority into a syslog code -}
makeCode :: Facility -> Priority -> Int
makeCode fac pri =
    let faccode = codeOffFac fac
        pricode = fromEnum pri
    in
        (faccode `shiftL` 3) .|. pricode

```

这段程序需要 SyslogTypes.hs , 代码如下:

```

-- file: ch27/SyslogTypes.hs
module SyslogTypes where

{- | Priorities define how important a log message is. -}

data Priority =
    DEBUG          -- ^ Debug messages
  | INFO           -- ^ Information
  | NOTICE        -- ^ Normal runtime conditions
  | WARNING        -- ^ General Warnings
  | ERROR          -- ^ General Errors
  | CRITICAL       -- ^ Severe situations
  | ALERT          -- ^ Take immediate action
  | EMERGENCY      -- ^ System is unusable
    deriving (Eq, Ord, Show, Read, Enum)

{- | Facilities are used by the system to determine where messages
are sent. -}

data Facility =
    KERN           -- ^ Kernel messages
  | USER          -- ^ General userland messages
  | MAIL           -- ^ E-Mail system
  | DAEMON         -- ^ Daemon (server process) messages
  | AUTH           -- ^ Authentication or security messages
  | SYSLOG         -- ^ Internal syslog messages
  | LPR            -- ^ Printer messages
  | NEWS           -- ^ Usenet news
  | UUCP           -- ^ UUCP messages
  | CRON           -- ^ Cron messages
  | AUTHPRIV       -- ^ Private authentication messages
  | FTP            -- ^ FTP messages
  | LOCAL0
  | LOCAL1
  | LOCAL2
  | LOCAL3
  | LOCAL4
  | LOCAL5
  | LOCAL6
  | LOCAL7
    deriving (Eq, Show, Read)

facToCode = [
    (KERN, 0),
    (USER, 1),
    (MAIL, 2),
    (DAEMON, 3),
    (AUTH, 4),
    (SYSLOG, 5),
    (LPR, 6),
    (NEWS, 7),
    (UUCP, 8),
    (CRON, 9),
    (AUTHPRIV, 10),
    (FTP, 11),
    (LOCAL0, 16),

```

```

        (LOCAL1, 17),
        (LOCAL2, 18),
        (LOCAL3, 19),
        (LOCAL4, 20),
        (LOCAL5, 21),
        (LOCAL6, 22),
        (LOCAL7, 23)
    ]

codeToFac = map (\(x, y) -> (y, x)) facToCode

{- | We can't use enum here because the numbering is discontinuous -}
codeOfFac :: Facility -> Int
codeOfFac f = case lookup f facToCode of
    Just x -> x
    _ -> error $ "Internal error in codeOfFac"

facOfCode :: Int -> Facility
facOfCode f = case lookup f codeToFac of
    Just x -> x
    _ -> error $ "Invalid code in facOfCode"

```

可以用 `ghci` 向本地的 `syslog` 服务器发送消息。服务器可以使用本章实现的例子，也可以使用其它的在 Linux 或者 POSIX 系统中的 `syslog` 服务器。注意，这些服务器默认禁用了 UDP 端口，你需要启用 UDP 以使 `syslog` 接收 UDP 消息。

可以使用下面这样的命令向本地 `syslog` 服务器发送一条消息：

```

ghci> :load syslogclient.hs
[1 of 2] Compiling SyslogTypes      ( SyslogTypes.hs, interpreted )
[2 of 2] Compiling Main              ( syslogclient.hs, interpreted )
Ok, modules loaded: SyslogTypes, Main.
ghci> h <- openlog "localhost" "514" "testprog"
Loading package parsec-2.1.0.0 ... linking ... done.
Loading package network-2.1.0.0 ... linking ... done.
ghci> syslog h USER INFO "This is my message"
ghci> closelog h

```

UDP Syslog 服务器

UDP 服务器会在服务器上绑定某个端口。其接收直接发到这个端口的包，并处理它们。UDP 是无状态的，面向包的协议，程序员通常使用 `recvFrom` 这个调用接收消息和发送机信息，在发送响应时会用到发送机信息。

```

-- file: ch27/syslogserver.hs
import Data.Bits
import Network.Socket
import Network.BSD
import Data.List

type HandlerFunc = SocketAddr -> String -> IO ()

serveLog :: String           -- ^ Port number or name; 514 is default
         -> HandlerFunc      -- ^ Function to handle incoming messages
         -> IO ()

serveLog port handlerfunc = withSocketsDo $
    do -- Look up the port. Either raises an exception or returns
       -- a nonempty list.
       addrinfos <- getAddrInfo
           (Just (defaultHints {addrFlags = [AI_PASSIVE]}))
           Nothing (Just port)
       let serveraddr = head addrinfos

       -- Create a socket
       sock <- socket (addrFamily serveraddr) Datagram defaultProtocol

       -- Bind it to the address we're listening to
       bindSocket sock (addrAddress serveraddr)

```

 v: latest ▾

```

-- Loop forever processing incoming data. Ctrl-C to abort.
procMessages sock
where procMessages sock =
    do -- Receive one UDP packet, maximum length 1024 bytes,
      -- and save its content into msg and its source
      -- IP and port into addr
      (msg, _, addr) <- recvFrom sock 1024
      -- Handle it
      handlerfunc addr msg
      -- And process more messages
      procMessages sock

-- A simple handler that prints incoming packets
plainHandler :: HandlerFunc
plainHandler addr msg =
    putStrLn $ "From " ++ show addr ++ ": " ++ msg

```

这段程序可以在 ghci 中执行。执行 `serveLog "1514" plainHandler` 将建立一个监听 1514 端口的 UDP 服务器。其使用 `plainHandler` 将每条收到的 UDP 包打印出来。按下 `Ctrl-C` 可以终止这个程序。

Note

处理错误。执行时收到了 `bind: permission denied` 消息？要确保端口值比 1024 大。某些操作系统不允许 root 之外的用户使用小于 1024 的端口。

使用 TCP 通信

TCP 被设计为确保互联网上的数据尽可能可靠地传输。TCP 是数据流传输。虽然流在传输时会被操作系统拆散为一个个单独的包，但是应用程序并不需要关心包的边界。TCP 负责确保如果流被传送到应用程序，它就是完整的、无改动、仅传输一次且保证顺序。显然，如果线缆被破坏会导致流量无法送达，任何协议都无法克服这类限制。

与 UDP 相比，这带来一些折衷。首先，在 TCP 会话开始必须传递一些包以建立连接。其次，对于每个短会话，UDP 将有性能优势。另外，TCP 会努力确保数据到达。如果会话的一端尝试向远端发送数据，但是没有收到响应，它将周期性的尝试重新传输数据直至放弃。这使得 TCP 面对丢包时比较健壮可靠。可是，它同样意味着 TCP 不是实时传输协议（如实况音频或视频传输）的最佳选择。

处理多个 TCP 流

TCP 的连接是有状态的。这意味着每个客户机和服务器之间都有一条专用的逻辑“频道”，而不是像 UDP 一样只是处理一次性的数据包。这简化了客户端开发者的工作。服务器端程序几乎总是需要同时处理多条 TCP 连接。如何做到这一点呢？

在服务器端，首先需要创建一个 socket 并绑定到某个端口，就像 UDP 一样。但这回不是重复监听从任意地址发来的数据，取而代之，你的主循环将围绕 `accept` 调用编写。每当有一个客户机连接，服务器操作系统为其分配一个新的 socket。所以我们的主 socket 只用来监听进来的连接，但从不发送数据。我们也获得了多个子 socket 可以同时使用，每个子 socket 从属于一个逻辑上的 TCP 会话。

在 Haskell 中，通常使用 `forkIO` 创建一个单独的轻量级线程以处理与子 socket 的通信。对此，Haskell 拥有一个高效的内部实现，执行得非常好。

TCP Syslog 服务器

让我们使用 TCP 的实现来替换 UDP 的 syslog 服务器。假设一条消息并不是定义为单独的包，而是以一个尾部的字符 'n' 结束。任意客户端可以使用 TCP 连接向服务器发送 0 或多条消息。我们可以像下面这样实现：

```

-- file: ch27/syslogtcpserver.hs
import Data.Bits
import Network.Socket
import Network.BSD
import Data.List
import Control.Concurrent
import Control.Concurrent.MVar
import System.IO

type HandlerFunc = SocketAddr -> String -> IO ()

```

 v: latest ▾

```

serveLog :: String           -- ^ Port number or name; 514 is default
         -> HandlerFunc      -- ^ Function to handle incoming messages
         -> IO ()

serveLog port handlerfunc = withSocketsDo $
  do -- Look up the port. Either raises an exception or returns
    -- a nonempty list.
    addrrinfos <- getAddrInfo
                (Just (defaultHints {addrFlags = [AI_PASSIVE]}))
                Nothing (Just port)
    let serveraddr = head addrrinfos

    -- Create a socket
    sock <- socket (addrFamily serveraddr) Stream defaultProtocol

    -- Bind it to the address we're listening to
    bindSocket sock (addrAddress serveraddr)

    -- Start listening for connection requests. Maximum queue size
    -- of 5 connection requests waiting to be accepted.
    listen sock 5

    -- Create a lock to use for synchronizing access to the handler
    lock <- newMVar ()

    -- Loop forever waiting for connections. Ctrl-C to abort.
    procRequests lock sock

where
  -- | Process incoming connection requests
  procRequests :: MVar () -> Socket -> IO ()
  procRequests lock mastersock =
    do (connsock, clientaddr) <- accept mastersock
       handle lock clientaddr
         "syslogtcpserver.hs: client connected"
       forkIO $ procMessages lock connsock clientaddr
       procRequests lock mastersock

  -- | Process incoming messages
  procMessages :: MVar () -> Socket -> SockAddr -> IO ()
  procMessages lock connsock clientaddr =
    do connhdl <- socketToHandle connsock ReadMode
       hSetBuffering connhdl LineBuffering
       messages <- hGetContents connhdl
       mapM_ (handle lock clientaddr) (lines messages)
       hClose connhdl
       handle lock clientaddr
         "syslogtcpserver.hs: client disconnected"


  -- Lock the handler before passing data to it.
  handle :: MVar () -> HandlerFunc
  -- This type is the same as
  -- handle :: MVar () -> SockAddr -> String -> IO ()
  handle lock clientaddr msg =
    withMVar lock
      (\a -> handlerfunc clientaddr msg >> return a)

  -- A simple handler that prints incoming packets
  plainHandler :: HandlerFunc
  plainHandler addr msg =
    putStrLn $ "From " ++ show addr ++ ": " ++ msg

```

SyslogTypes 的实现，见 **UDP 客户端例子: syslog**。

让我们读一下源码。主循环是 `procRequests`，这是一个死循环，用于等待来自客户端的新连接。`accept` 调用将一直阻塞，直到一个客户端来连接。当有客户端连接，我们获得一个新 `socket` 和客户端地址。我们向处理函数发送一条关于新连接的消息，接着使用 `forkIO` 建立一个线程处理来自客户端的数据。这条线程执行 `procMessages`。

处理 TCP 数据时，为了方便，通常将 `socket` 转换为 Haskell 句柄。我们也同样处理，并明确设置了缓冲 – 一个 TCP  `v: latest` ◿ 接着，设置惰性读取 `socket` 句柄。对每个传入的行，我们都将其传给 `handle`。当没有更多数据时 – 远端已经关闭了 `socket` – 我们输出一条会话结束的消息。

因为可能同时收到多条消息，我们需要确保没有将多条消息同时写入一个处理函数。那将导致混乱的输出。我们使用了一个简单的锁以序列化对处理函数的访问，并且编写了一个简单的 `handle` 函数处理它。

你可以使用下面我们将展示的客户机代码测试，或者直接使用 `telnet` 程序来连接这个服务器。你向其发送的每一行输入都将被服务器原样返回。我们来试一下：

```
ghci> :load syslogtcpserver.hs
[1 of 1] Compiling Main             ( syslogtcpserver.hs, interpreted )
Ok, modules loaded: Main.
ghci> serveLog "10514" plainHandler
Loading package parsec-2.1.0.0 ... linking ... done.
Loading package network-2.1.0.0 ... linking ... done.
```

此处，服务器从 10514 端口监听新连接。在有某个客户机过来连接之前，它什么事儿都不做。我们可以使用 `telnet` 来连接这个服务器：

```
~$ telnet localhost 10514
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^['.
Test message
^]
telnet> quit
Connection closed.
```

于此同时，在我们运行 TCP 服务器的终端上，你将看到如下输出：

```
From 127.0.0.1:38790: syslogtcpserver.hs: client connected
From 127.0.0.1:38790: Test message
From 127.0.0.1:38790: syslogtcpserver.hs: client disconnected
```

其显示一个客户端从本机 (127.0.0.1) 的 38790 端口连上了主机。连接之后，它发送了一条消息，然后断开。当你扮演一个 TCP 客户端时，操作系统将分配一个未被使用的端口给你。通常这个端口在你每次运行程序时都不一样。

TCP Syslog 客户端

现在，为我们的 TCP syslog 协议编写一个客户端。这个客户端与 UDP 客户端类似，但是有一些变化。首先，因为 TCP 是流式协议，我们可以使用句柄传输数据而不需要使用底层的 `socket` 操作。其次，不在需要在 `SyslogHandle` 中保存目的地址，因为我们将使用 `connect` 建立 TCP 连接。最后，我们需要一个途径，以区分不同的消息。UDP 中，这很容易，因为每条消息都是不相关的逻辑包。TCP 中，我们将仅使用换行符 `'n'` 来作为消息结尾的标识，尽管这意味着不能在单条消息中发送多行信息。这是代码：

```
-- file: ch27/syslogtcpclient.hs
import Data.Bits
import Network.Socket
import Network.BSD
import Data.List
import SyslogTypes
import System.IO

data SyslogHandle =
  SyslogHandle {slHandle :: Handle,
                slProgram :: String}

openlog :: HostName          -- ^ Remote hostname, or localhost
        -> String           -- ^ Port number or name; 514 is default
        -> String           -- ^ Name to log under
        -> IO SyslogHandle  -- ^ Handle to use for logging
openlog hostname port progname =
  do -- Look up the hostname and port. Either raises an exception
    -- or returns a nonempty list. First element in that list
    -- is supposed to be the best option.
    addrfinfos <- getAddrInfo Nothing (Just hostname) (Just port)
    let serveraddr = head addrfinfos

    -- Establish a socket for communication
```

 v: latest ▾

```

sock <- socket (addrFamily serveraddr) Stream defaultProtocol

-- Mark the socket for keep-alive handling since it may be idle
-- for long periods of time
setSocketOption sock KeepAlive 1

-- Connect to server
connect sock (addrAddress serveraddr)

-- Make a Handle out of it for convenience
h <- socketToHandle sock WriteMode

-- We're going to set buffering to BlockBuffering and then
-- explicitly call hFlush after each message, below, so that
-- messages get logged immediately
hSetBuffering h (BlockBuffering Nothing)

-- Save off the socket, program name, and server address in a handle
return $ SyslogHandle h progname

syslog :: SyslogHandle -> Facility -> Priority -> String -> IO ()
syslog syslogh fac pri msg =
  do hPutStrLn (slHandle syslogh) sendmsg
    -- Make sure that we send data immediately
    hFlush (slHandle syslogh)
  where code = makeCode fac pri
        sendmsg = "<" ++ show code ++ ">" ++ (slProgram syslogh) ++
                  ": " ++ msg

closelog :: SyslogHandle -> IO ()
closelog syslogh = hClose (slHandle syslogh)

{- / Convert a facility and a priority into a syslog code -}
makeCode :: Facility -> Priority -> Int
makeCode fac pri =
  let faccode = codeOffFac fac
      pricode = fromEnum pri
  in
    (faccode `shiftL` 3) .|. pricode

```

可以在 ghci 中试着运行它。如果还没有关闭之前的 TCP 服务器，你的会话看上去可能会是这样：

```

ghci> :load syslogtcpclient.hs
Loading package base ... linking ... done.
[1 of 2] Compiling SyslogTypes      ( SyslogTypes.hs, interpreted )
[2 of 2] Compiling Main              ( syslogtcpclient.hs, interpreted )
Ok, modules loaded: Main, SyslogTypes.
ghci> openlog "localhost" "10514" "tcpctest"
Loading package parsec-2.1.0.0 ... linking ... done.
Loading package network-2.1.0.0 ... linking ... done.
ghci> sl <- openlog "localhost" "10514" "tcpctest"
ghci> syslog sl USER INFO "This is my TCP message"
ghci> syslog sl USER INFO "This is my TCP message again"
ghci> closelog sl

```

结束时，服务器上将看到这样的输出：

```

From 127.0.0.1:46319: syslogtcpserver.hs: client connected
From 127.0.0.1:46319: <9>tcpctest: This is my TCP message
From 127.0.0.1:46319: <9>tcpctest: This is my TCP message again
From 127.0.0.1:46319: syslogtcpserver.hs: client disconnected

```

<9> 是优先级和设施代码，和之前 UDP 例子中的意思一样。

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



通过以下方式登录

或注册一个 DISQUS 帐号 ?

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 — ...

Pearls of Functional Algorithm Design

1条评论 • 6年前

Tonghua Su — where is the content?
- 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

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