Filters, sources, sinks, and pumps

or Functional programming for the rest of us

Diego Nehab

August 16, 2016

Abstract

Certain data processing operations can be implemented in the form of filters. A filter is a function that can process data received in consecutive invocations, returning partial results each time it is called. Examples of operations that can be implemented as filters include the end-of-line normalization for text, Base64 and Quoted-Printable transfer content encodings, the breaking of text into lines, SMTP dot-stuffing, and there are many others. Filters become even more powerful when we allow them to be chained together to create composite filters. In this context, filters can be seen as the internal links in a chain of data transformations. Sources and sinks are the corresponding end points in these chains. A source is a function that produces data, chunk by chunk, and a sink is a function that takes data, chunk by chunk. Finally, pumps are procedures that actively drive data from a source to a sink, and indirectly through all intervening filters. In this article, we describe the design of an elegant interface for filters, sources, sinks, chains, and pumps, and we illustrate each step with concrete examples.

1 Introduction

Within the realm of networking applications, we are often required to apply transformations to streams of data. Examples include the end-of-line normalization for text, Base64 and Quoted-Printable transfer content encodings, breaking text into lines with a maximum number of columns, SMTP dot-stuffing, `gzip` compression, HTTP chunked transfer coding, and the list goes on.

Many complex tasks require a combination of two or more such transformations, and therefore a general mechanism for promoting reuse is desirable. In the process of designing `LuaSocket 2.0`, we repeatedly faced this problem. The solution we reached proved to be very general and convenient. It is based on the concepts of filters, sources, sinks, and pumps, which we introduce below.

Filters are functions that can be repeatedly invoked with chunks of input, successively returning processed chunks of output. Naturally, the result of concatenating all the output chunks must be the same as the result of applying the

filter to the concatenation of all input chunks. In fancier language, filters *commute* with the concatenation operator. More importantly, filters must handle input data correctly no matter how the stream has been split into chunks.

A *chain* is a function that transparently combines the effect of one or more filters. The interface of a chain is indistinguishable from the interface of its component filters. This allows a chained filter to be used wherever an atomic filter is accepted. In particular, chains can be themselves chained to create arbitrarily complex operations.

Filters can be seen as internal nodes in a network through which data will flow, potentially being transformed many times along the way. Chains connect these nodes together. The initial and final nodes of the network are *sources* and *sinks*, respectively. Less abstractly, a source is a function that produces new chunks of data every time it is invoked. Conversely, sinks are functions that give a final destination to the chunks of data they receive in sucessive calls. Naturally, sources and sinks can also be chained with filters to produce filtered sources and sinks.

Finally, filters, chains, sources, and sinks are all passive entities: they must be repeatedly invoked in order for anything to happen. *Pumps* provide the driving force that pushes data through the network, from a source to a sink, and indirectly through all intervening filters.

In the following sections, we start with a simplified interface, which we later refine. The evolution we present is not contrived: it recreates the steps we ourselves followed as we consolidated our understanding of these concepts within our application domain.

1.1 A simple example

The end-of-line normalization of text is a good example to motivate our initial filter interface. Assume we are given text in an unknown end-of-line convention (including possibly mixed conventions) out of the commonly found Unix (`LF`), Mac OS (`CR`), and DOS (`CR LF`) conventions. We would like to be able to use the folowing code to normalize the end-of-line markers:

```
local CRLF = "\013\010"
local input = source.chain(source.file(io.stdin), normalize(CRLF))
local output = sink.file(io.stdout)
pump.all(input, output)
```

This program should read data from the standard input stream and normalize the end-of-line markers to the canonic `CR LF` marker, as defined by the MIME standard. Finally, the normalized text should be sent to the standard output stream. We use a *file source* that produces data from standard input, and chain it with a filter that normalizes the data. The pump then repeatedly obtains data from the source, and passes it to the *file sink*, which sends it to the standard output.

In the code above, the `normalize` *factory* is a function that creates our normalization filter, which replaces any end-of-line marker with the canonic

marker. The initial filter interface is trivial: a filter function receives a chunk of input data, and returns a chunk of processed data. When there are no more input data left, the caller notifies the filter by invoking it with a `nil` chunk. The filter responds by returning the final chunk of processed data (which could of course be the empty string).

Although the interface is extremely simple, the implementation is not so obvious. A normalization filter respecting this interface needs to keep some kind of context between calls. This is because a chunk boundary may lie between the `CR` and `LF` characters marking the end of a single line. This need for contextual storage motivates the use of factories: each time the factory is invoked, it returns a filter with its own context so that we can have several independent filters being used at the same time. For efficiency reasons, we must avoid the obvious solution of concatenating all the input into the context before producing any output chunks.

To that end, we break the implementation into two parts: a low-level filter, and a factory of high-level filters. The low-level filter is implemented in C and does not maintain any context between function calls. The high-level filter factory, implemented in Lua, creates and returns a high-level filter that maintains whatever context the low-level filter needs, but isolates the user from its internal details. That way, we take advantage of C's efficiency to perform the hard work, and take advantage of Lua's simplicity for the bookkeeping.

1.2 The Lua part of the filter

Below is the complete implementation of the factory of high-level end-of-line normalization filters:

```
function filter.cycle(lowlevel, context, extra)
  return function(chunk)
    local ret
    ret, context = lowlevel(context, chunk, extra)
    return ret
  end
end

function normalize(marker)
  return filter.cycle(eol, 0, marker)
end
```

The `normalize` factory simply calls a more generic factory, the `cycle` factory, passing the low-level filter `eol`. The `cycle` factory receives a low-level filter, an initial context, and an extra parameter, and returns a new high-level filter. Each time the high-level filer is passed a new chunk, it invokes the low-level filter with the previous context, the new chunk, and the extra argument. It is the low-level filter that does all the work, producing the chunk of processed data and a new context. The high-level filter then replaces its internal context, and returns the processed chunk of data to the user. Notice that we take advantage of Lua's lexical scoping to store the context in a closure between function calls.

1.3 The C part of the filter

As for the low-level filter, we must first accept that there is no perfect solution to the end-of-line marker normalization problem. The difficulty comes from an inherent ambiguity in the definition of empty lines within mixed input. However, the following solution works well for any consistent input, as well as for non-empty lines in mixed input. It also does a reasonable job with empty lines and serves as a good example of how to implement a low-level filter.

The idea is to consider both `CR` and `LF` as end-of-line *candidates*. We issue a single break if any candidate is seen alone, or if it is followed by a different candidate. In other words, `CR CR` and `LF LF` each issue two end-of-line markers, whereas `CR LF` and `LF CR` issue only one marker each. It is easy to see that this method correctly handles the most common end-of-line conventions.

With this in mind, we divide the low-level filter into two simple functions. The inner function `pushchar` performs the normalization itself. It takes each input character in turn, deciding what to output and how to modify the context. The context tells if the last processed character was an end-of-line candidate, and if so, which candidate it was. For efficiency, we use Lua's auxiliary library's buffer interface:

```
#define candidate(c) (c == CR || c == LF)
static int pushchar(int c, int last, const char *marker,
    luaL_Buffer *buffer) {
  if (candidate(c)) {
    if (candidate(last)) {
      if (c == last)
        luaL_addstring(buffer, marker);
      return 0;
    } else {
      luaL_addstring(buffer, marker);
      return c;
    }
  } else {
    luaL_pushchar(buffer, c);
    return 0;
  }
}
```

The outer function `eol` simply interfaces with Lua. It receives the context and input chunk (as well as an optional custom end-of-line marker), and returns the transformed output chunk and the new context. Notice that if the input chunk is `nil`, the operation is considered to be finished. In that case, the loop will not execute a single time and the context is reset to the initial state. This allows the filter to be reused many times:

```
static int eol(lua_State *L) {
  int context = luaL_checkint(L, 1);
  size_t isize = 0;
  const char *input = luaL_optlstring(L, 2, NULL, &isize);
  const char *last = input + isize;
  const char *marker = luaL_optstring(L, 3, CRLF);
  luaL_Buffer buffer;
  luaL_buffinit(L, &buffer);
  if (!input) {
    lua_pushnil(L);
    lua_pushnumber(L, 0);
    return 2;
  }
  while (input < last)
    context = pushchar(*input++, context, marker, &buffer);
  luaL_pushresult(&buffer);
  lua_pushnumber(L, context);
  return 2;
}
```

When designing filters, the challenging part is usually deciding what to store in the context. For line breaking, for instance, it could be the number of bytes that still fit in the current line. For Base64 encoding, it could be a string with the bytes that remain after the division of the input into 3-byte atoms. The MIME module in the `LuaSocket` distribution has many other examples.

2 Filter chains

Chains greatly increase the power of filters. For example, according to the standard for Quoted-Printable encoding, text should be normalized to a canonic end-of-line marker prior to encoding. After encoding, the resulting text must be broken into lines of no more than 76 characters, with the use of soft line breaks (a line terminated by the = sign). To help specifying complex transformations like this, we define a chain factory that creates a composite filter from one or more filters. A chained filter passes data through all its components, and can be used wherever a primitive filter is accepted.

The chaining factory is very simple. The auxiliary function `chainpair` chains two filters together, taking special care if the chunk is the last. This is because the final `nil` chunk notification has to be pushed through both filters in turn:

```
local function chainpair(f1, f2)
  return function(chunk)
    local ret = f2(f1(chunk))
    if chunk then return ret
    else return ret .. f2() end
  end
end
```

```
function filter.chain(...)
  local f = select(1, ...)
  for i = 2, select('#', ...) do
    f = chainpair(f, select(i, ...))
  end
  return f
end
```

Thanks to the chain factory, we can define the Quoted-Printable conversion as such:

```
local qp = filter.chain(normalize(CRLF), encode("quoted-printable"),
  wrap("quoted-printable"))
local input = source.chain(source.file(io.stdin), qp)
local output = sink.file(io.stdout)
pump.all(input, output)
```

3 Sources, sinks, and pumps

The filters we introduced so far act as the internal nodes in a network of transformations. Information flows from node to node (or rather from one filter to the next) and is transformed along the way. Chaining filters together is our way to connect nodes in this network. As the starting point for the network, we need a source node that produces the data. In the end of the network, we need a sink node that gives a final destination to the data.

3.1 Sources

A source returns the next chunk of data each time it is invoked. When there is no more data, it simply returns `nil`. In the event of an error, the source can inform the caller by returning `nil` followed by the error message.

Below are two simple source factories. The `empty` source returns no data, possibly returning an associated error message. The `file` source yields the contents of a file in a chunk by chunk fashion:

```
function source.empty(err)
  return function()
    return nil, err
  end
end

function source.file(handle, io_err)
  if handle then
    return function()
      local chunk = handle:read(2048)
      if not chunk then handle:close() end
      return chunk
    end
  else return source.empty(io_err or "unable to open file") end
end
```

3.2 Filtered sources

A filtered source passes its data through the associated filter before returning it to the caller. Filtered sources are useful when working with functions that get their input data from a source (such as the pumps in our examples). By chaining a source with one or more filters, such functions can be transparently provided with filtered data, with no need to change their interfaces. Here is a factory that does the job:

```
function source.chain(src, f)
  return function()
    if not src then
      return nil
    end
    local chunk, err = src()
    if not chunk then
      src = nil
      return f(nil)
    else
      return f(chunk)
    end
  end
end
```

3.3 Sinks

Just as we defined an interface for a source of data, we can also define an interface for a data destination. We call any function respecting this interface a sink. In our first example, we used a file sink connected to the standard output.

Sinks receive consecutive chunks of data, until the end of data is signaled by a `nil` input chunk. A sink can be notified of an error with an optional extra argument that contains the error message, following a `nil` chunk. If a sink detects an error itself, and wishes not to be called again, it can return `nil`, followed by an error message. A return value that is not `nil` means the sink will accept more data.

Below are two useful sink factories. The table factory creates a sink that stores individual chunks into an array. The data can later be efficiently concatenated into a single string with Lua's `table.concat` library function. The `null` sink simply discards the chunks it receives:

```
function sink.table(t)
  t = t or {}
  local f = function(chunk, err)
    if chunk then table.insert(t, chunk) end
    return 1
  end
  return f, t
end
```

```
local function null()
  return 1
end
function sink.null()
  return null
end
```

Naturally, filtered sinks are just as useful as filtered sources. A filtered sink passes each chunk it receives through the associated filter before handing it down to the original sink. In the following example, we use a source that reads from the standard input. The input chunks are sent to a table sink, which has been coupled with a normalization filter. The filtered chunks are then concatenated from the output array, and finally sent to standard out:

```
local input = source.file(io.stdin)
local output, t = sink.table()
output = sink.chain(normalize(CRLF), output)
pump.all(input, output)
io.write(table.concat(t))
```

3.4 Pumps

Although not on purpose, our interface for sources is compatible with Lua iterators. That is, a source can be neatly used in conjunction with `for` loops. Using our file source as an iterator, we can write the following code:

```
for chunk in source.file(io.stdin) do
  io.write(chunk)
end
```

Loops like this will always be present because everything we designed so far is passive. Sources, sinks, filters: none of them can do anything on their own. The operation of pumping all data a source can provide into a sink is so common that it deserves its own function:

```
function pump.step(src, snk)
  local chunk, src_err = src()
  local ret, snk_err = snk(chunk, src_err)
  if chunk and ret then return 1
  else return nil, src_err or snk_err end
end

function pump.all(src, snk, step)
    step = step or pump.step
    while true do
        local ret, err = step(src, snk)
        if not ret then
            if err then return nil, err
            else return 1 end
        end
    end
end
```

The `pump.step` function moves one chunk of data from the source to the sink. The `pump.all` function takes an optional `step` function and uses it to pump all the data from the source to the sink. Here is an example that uses the Base64 and the line wrapping filters from the `LuaSocket` distribution. The program reads a binary file from disk and stores it in another file, after encoding it to the Base64 transfer content encoding:

```
local input = source.chain(
  source.file(io.open("input.bin", "rb")),
  encode("base64"))
local output = sink.chain(
  wrap(76),
  sink.file(io.open("output.b64", "w")))
pump.all(input, output)
```

The way we split the filters here is not intuitive, on purpose. Alternatively, we could have chained the Base64 encode filter and the line-wrap filter together, and then chain the resulting filter with either the file source or the file sink. It doesn't really matter.

4 Exploding filters

Our current filter interface has one serious shortcoming. Consider for example a `gzip` decompression filter. During decompression, a small input chunk can be exploded into a huge amount of data. To address this problem, we decided to change the filter interface and allow exploding filters to return large quantities of output data in a chunk by chunk manner.

More specifically, after passing each chunk of input to a filter, and collecting the first chunk of output, the user must now loop to receive other chunks from the filter until no filtered data is left. Within these secondary calls, the caller passes an empty string to the filter. The filter responds with an empty string when it is ready for the next input chunk. In the end, after the user passes a `nil` chunk notifying the filter that there is no more input data, the filter might still have to produce too much output data to return in a single chunk. The user has to loop again, now passing `nil` to the filter each time, until the filter itself returns `nil` to notify the user it is finally done.

Fortunately, it is very easy to modify a filter to respect the new interface. In fact, the end-of-line translation filter we presented earlier already conforms to it. The complexity is encapsulated within the chaining functions, which must now include a loop. Since these functions only have to be written once, the user is rarely affected. Interestingly, the modifications do not have a measurable negative impact in the performance of filters that do not need the added flexibility. On the other hand, for a small price in complexity, the changes make exploding filters practical.

5 A complex example

The LTN12 module in the `LuaSocket` distribution implements all the ideas we have described. The MIME and SMTP modules are tightly integrated with LTN12, and can be used to showcase the expressive power of filters, sources, sinks, and pumps. Below is an example of how a user would proceed to define and send a multipart message, with attachments, using `LuaSocket`:

```
local smtp = require"socket.smtp"
local mime = require"mime"
local ltn12 = require"ltn12"

local message = smtp.message{
  headers = {
    from = "Sicrano <sicrano@example.com>",
    to = "Fulano <fulano@example.com>",
    subject = "A message with an attachment"},
  body = {
    preamble = "Hope you can see the attachment" .. CRLF,
    [1] = {
      body = "Here is our logo" .. CRLF},
    [2] = {
      headers = {
        ["content-type"] = 'image/png; name="luasocket.png"',
        ["content-disposition"] =
          'attachment; filename="luasocket.png"',
        ["content-description"] = 'LuaSocket logo',
        ["content-transfer-encoding"] = "BASE64"},
      body = ltn12.source.chain(
        ltn12.source.file(io.open("luasocket.png", "rb")),
        ltn12.filter.chain(
          mime.encode("base64"),
          mime.wrap()))}}}

assert(smtp.send{
  rcpt = "<fulano@example.com>",
  from = "<sicrano@example.com>",
  source = message})
```

The `smtp.message` function receives a table describing the message, and returns a source. The `smtp.send` function takes this source, chains it with the SMTP dot-stuffing filter, connects a socket sink with the server, and simply pumps the data. The message is never assembled in memory. Everything is produced on demand, transformed in small pieces, and sent to the server in chunks, including the file attachment which is loaded from disk and encoded on the fly. It just works.

6 Conclusions

In this article, we introduced the concepts of filters, sources, sinks, and pumps to the Lua language. These are useful tools for stream processing in general. Sources provide a simple abstraction for data acquisition. Sinks provide an abstraction for final data destinations. Filters define an interface for data transformations. The chaining of filters, sources and sinks provides an elegant way to create arbitrarily complex data transformations from simpler components. Pumps simply push the data through.

7 Acknowledgements

The concepts described in this text are the result of long discussions with David Burgess. A version of this text has been released on-line as the Lua Technical Note 012, hence the name of the corresponding LuaSocket module, LTN12. Wim Couwenberg contributed to the implementation of the module, and Adrian Sietsma was the first to notice the correspondence between sources and Lua iterators.