Pandoc Lua Filters

Albert Krewinkel John MacFarlane

January 10, 2020

Introduction

Pandoc has long supported filters, which allow the pandoc abstract syntax tree (AST) to be manipulated between the parsing and the writing phase. Traditional pandoc filters accept a JSON representation of the pandoc AST and produce an altered JSON representation of the AST. They may be written in any programming language, and invoked from pandoc using the `--filter` option.

Although traditional filters are very flexible, they have a couple of disadvantages. First, there is some overhead in writing JSON to stdout and reading it from stdin (twice, once on each side of the filter). Second, whether a filter will work will depend on details of the user's environment. A filter may require an interpreter for a certain programming language to be available, as well as a library for manipulating the pandoc AST in JSON form. One cannot simply provide a filter that can be used by anyone who has a certain version of the pandoc executable.

Starting with version 2.0, pandoc makes it possible to write filters in Lua without any external dependencies at all. A Lua interpreter (version 5.3) and a Lua library for creating pandoc filters is built into the pandoc executable. Pandoc data types are marshaled to Lua directly, avoiding the overhead of writing JSON to stdout and reading it from stdin.

Here is an example of a Lua filter that converts strong emphasis to small caps:

```lua
return {
  {
    Strong = function (elem)
      return pandoc.SmallCaps(elem.c)
    end,
  }
}
```

or equivalently,

```lua
function Strong(elem)
  return pandoc.SmallCaps(elem.c)
end
```

This says: walk the AST, and when you find a Strong element, replace it with a SmallCaps element with the same content.

To run it, save it in a file, say `smallcaps.lua`, and invoke pandoc with `--lua-filter=smallcaps.lua`.

Here's a quick performance comparison, converting the pandoc manual (MANUAL.txt) to HTML, with versions of the same JSON filter written in compiled Haskell (`smallcaps`) and interpreted Python (`smallcaps.py`):

Command	Time
`pandoc`	1.01s
`pandoc --filter ./smallcaps`	1.36s
`pandoc --filter ./smallcaps.py`	1.40s
`pandoc --lua-filter ./smallcaps.lua`	1.03s

As you can see, the Lua filter avoids the substantial overhead associated with marshaling to and from JSON over a pipe.

Lua filter structure

Lua filters are tables with element names as keys and values consisting of functions acting on those elements.

Filters are expected to be put into separate files and are passed via the `--lua-filter` command-line argument. For example, if a filter is defined in a file `current-date.lua`, then it would be applied like this:

```
pandoc --lua-filter=current-date.lua -f markdown MANUAL.txt
```

The `--lua-filter` option may be supplied multiple times. Pandoc applies all filters (including JSON filters specified via `--filter` and Lua filters specified via `--lua-filter`) in the order they appear on the command line.

Pandoc expects each Lua file to return a list of filters. The filters in that list are called sequentially, each on the result of the previous filter. If there is no value returned by the filter script, then pandoc will try to generate a single filter by collecting all top-level functions whose names correspond to those of pandoc elements (e.g., `Str`, `Para`, `Meta`, or `Pandoc`). (That is why the two examples above are equivalent.)

For each filter, the document is traversed and each element subjected to the filter. Elements for which the filter contains an entry (i.e. a function of the same name) are passed to Lua element filtering function. In other words, filter entries will be called for each corresponding element in the document, getting the respective element as input.

The return value of a filter function must be one of the following:

- nil: this means that the object should remain unchanged.
- a pandoc object: this must be of the same type as the input and will replace the original object.
- a list of pandoc objects: these will replace the original object; the list is merged with the neighbors of the original objects (spliced into the list the original object belongs to); returning an empty list deletes the object.

The function's output must result in an element of the same type as the input. This means a filter function acting on an inline element must return either nil, an inline, or a list of inlines, and a function filtering a block element must return one of nil, a block, or a list of block elements. Pandoc will throw an error if this condition is violated.

If there is no function matching the element's node type, then the filtering system will look for a more general fallback function. Two fallback functions are supported, `Inline` and `Block`. Each matches elements of the respective type.

Elements without matching functions are left untouched.

See module documentation for a list of pandoc elements.

Filters on element sequences

For some filtering tasks, it is necessary to know the order in which elements occur in the document. It is not enough then to inspect a single element at a time.

There are two special function names, which can be used to define filters on lists of blocks or lists of inlines.

`Inlines (inlines)` If present in a filter, this function will be called on all lists of inline elements, like the content of a [Para] (paragraph) block, or the description of an [Image]. The `inlines` argument passed to the function will be a [List] of [Inline] elements for each call.

`Blocks (blocks)` If present in a filter, this function will be called on all lists of block elements, like the content of a [MetaBlocks] meta element block, on each item of a list, and the main content of the [Pandoc] document. The `blocks` argument passed to the function will be a [List] of [Block] elements for each call.

These filter functions are special in that the result must either be nil, in which case the list is left unchanged, or must be a list of the correct type, i.e., the same type as the input argument. Single elements are **not** allowed as return values, as a single element in this context usually hints at a bug.

See "Remove spaces before normal citations" for an example.

This functionality has been added in pandoc 2.9.2.

Traversal order

The traversal order of filters can be selected by setting the key `traverse` to either `'topdown'` or `'typewise'`; the default is `'typewise'`.

Example:

```
local filter = {
  traverse = 'topdown',
  -- ... filter functions ...
}
return {filter}
```

Support for this was added in pandoc 2.17; previous versions ignore the `traverse` setting.

Typewise traversal

Element filter functions within a filter set are called in a fixed order, skipping any which are not present:

1. functions for *Inline* elements,
2. the `Inlines` filter function,
3. functions for *Block* elements ,
4. the `Blocks` filter function,
5. the `Meta` filter function, and last
6. the `Pandoc` filter function.

It is still possible to force a different order by explicitly returning multiple filter sets. For example, if the filter for *Meta* is to be run before that for *Str*, one can write

```
-- ... filter definitions ...

return {
  { Meta = Meta },  -- (1)
  { Str = Str }     -- (2)
}
```

Filter sets are applied in the order in which they are returned. All functions in set (1) are thus run before those in (2), causing the filter function for *Meta* to be run before the filtering of *Str* elements is started.

Topdown traversal

It is sometimes more natural to traverse the document tree depth-first from the root towards the leaves, and all in a single run.

For example, a block list `[Plain [Str "a"], Para [Str "b"]]` will try the following filter functions, in order: `Blocks`, `Plain`, `Inlines`, `Str`, `Para`, `Inlines`, `Str`.

Topdown traversals can be cut short by returning `false` as a second value from the filter function. No child-element of the returned element is processed in that case.

For example, to exclude the contents of a footnote from being processed, one might write

```lua
traverse = 'topdown'
function Note (n)
  return n, false
end
```