The XML package: Best Practices

Duncan Temple Lang, University of California at Davis

Table of Contents

I have received several emails recently (early 2011) asking me about aspects of the XML package that I no longer recommend. These questions were probably based on documentation for the package that dates back many, many years. Those approaches still work and have advantages in certain cases, but I don't recommend them.

Basically, I recommend using the "internal" C-level representation for XML documents, trees, and nodes. These involve using the `xmlParse()` function and the `newXMLDoc()` and `newXMLNode()` (`newXMLTextNode()`, `newXMLCommentNode()`, `newXMLPINode()`, `newXMLCDataNode()`, `newXMLNamespace()`.)

The other approaches are to use "pure" R-level representations of the tree and nodes. There are two primary approaches to this

1. list of lists of lists where each node is a list consisting of an element name, a character vector of attributes,

2. a collection of nodes and information associating nodes with their parents or their children

While both can be readily used to represent a fixed tree, we use the second approach within an environment to facilitate modifying the tree, e.g. adding new nodes, changing existing nodes. The `XMLHashTree` class is used for this.

Another advantage of the nodes and separate data structure that represents the relationships between the nodes (children of what node) is that we can traverse the tree in either direction, i.e. finding the parent of a node or its children. The list-of-lists approach does not allow this (without additional structures).

Why should you use the internal/libxml2/C representation? Two primary reasons: speed, and ease of manipulating and querying the resulting tree. Basically, using the internal representations avoid the time and memory involved in converting the internal XML representation to R objects. Most importantly, we can only use the XPath language to query an XML document if that document is represented as internal nodes. XPath is such a powerful and efficient means of extracting nodes in a tree that this wins the argument over representation for almost all cases.

Why *not* use the internal/libxml2/C representation? One problem is that when we serialize an R object that contains an XML tree or node in this form, by default, that will not be restored as the same R object when deserialized/load-ed back into an R session. This is because the R object is a pointer to an opaque data structure in memory rather than values that are known to R. If we use "pure" R-based representations of the tree, then they will be serialized and deserialized identically across R sessions. But why is this not a real concern? Firstly, we can serialize an XML document or node trivially be converting it to a string and write

it to a file. See `saveXML`(). This does mean we have to explicitly serialize and deserialize XML content separately from other R objects. If we do want to handle regular R objects and XML objects together, we can use R's `serialize`() or `saveRDS`() functions and their `refhook` parameter to specify a way to handle serializing these XML representations so that they can be faithfully restored. We can use `xmlSerializeHook`() and `xmlDeserializeHook`() in the XML package to do this.

Creating XML Content in R

There are two styles of creating XML (or HTML) content in R. The most common and obvious us to create the content by `paste`()ing and `sprintf`()ing strings together to represent the content. The other approach is to create node objects and piece the tree together in a more structured manner.

The node approach is more flexible and a better programming approach. It allows us to easily build individual pieces at different stages in our code without having to worry about putting the pieces together in a specific order. The string approach is simple but a little brute-force and does constrain the software development needed to create the entire document. To add content to existing content requires the use of regular expressions or of parsing the XML content into an actual tree of nodes! A significant benefit of the string approach is that it can be vectorized, while the creation of nodes is not (and unlikely to be).

Creating nodes rather than using strings works very well when the nodes being created are very different in structure/content and cannot easily be vectorized using string operations. In this case, there is no benefit to using the string approach. When the nodes being created are very similar, use the string approach. But best of all, we can mix the two approaches. Often, a collection of sub-nodes within the tree are very similar, e.g. **<Placemark>** elements in KML (the Keyhole Markup Language used for displays, e.g., Google Earth). In this case, we can create the collection of these nodes easily using

1. string methods to construct the text for the nodes
2. surround this text within a temporary, unused node
3. parse the string content as XML to create actual nodes
4. add these as children of the real XML node we constructed

A good example of this is in the RKML package.

This gives us the best of both worlds where we can write a function that returns actual XML nodes or adds those nodes to an existing parent node, but is capable of doing this using either approach and can chose whichever one is most appropriate. In this way, the implementation is encapsulated and the overall approach can use the node approach rather than being forced to use the string approach throughout.

The `xmlOutputBuffer`() and `xmlOutputDOM`() functions are means to creating an XML tree with a higher-level interface than strings made up functions for adding nodes. I suggest using the C-level internal nodes provided by `newXMLNode`() and friends.

JSON versus XML

Many REST-based Web Services are now offering to return results in either XML or JavaScript Object Notation (JSON) format.Which should you use? This is true when dealing with XML or JSON in an context,

and not just Web Services. Which format should you prefer? In many cases, you won't have a choice and then the good news is that there are tools in R for both - XML, RJSONIO and `rjson`.

The answer on which format to use depends on what you want to do with the content/document. If the result is large, the JSON representation is likely to be somewhat smaller than the XML representation and so be transferred from the server to your client machine faster. If the results are compressed, the size difference is not likely to be very larger. Furthermore, there has been a lot of silly comparisons of the size of JSON relative to XML by people (even on json.org) where the examples have used repetitious and silly XML representations.

JSON is very simple to read into R, e.g. using the `fromJSON()` function in either of the RJSONIO or `rjson` packages. (`RJSONIO()` is faster as it uses a C-based parser.) This simplicity means less control over how the results are represented in R and one might need to rearrange the results in R in a second step. This can consume more memory than if one used XML directly, but often this is not an issue.

If the document is large (either in JSON or XML), and you do not want to extract it all but only a few fields from each record, then XML and XPath is likely to be more efficient in both speed and memory. There is no standard, widely implemented equivalent of XPath for JSON. (There are implementations of JPath but that is not a standard and is implemented only in JavaScript and so not easily brought into R (although we use SpiderMonkey to call JavaScript code from R).)

Basically, JSON is slightly more succinct than XML and that is good, but not as much as is touted by some people and compression can effectively remove that distinction. JSON is less complex, but less general. It works very well for dealing with common, raw, basic data structures we use in much of computing. It does not deal well with structured aggregates and more complex, class-oriented structures.