Memory Management in the the XML Package

Duncan Temple Lang, University of California at Davis

Abstract

We describe some of the complexities of memory management in the XML package. We outline various strategies that the R programmer can use with this package to make computations more efficient. We also explain how to reason about managing nodes and documents in this and related packages (e.g. Sxslt).

Table of Contents

In R, garbage collection just works as the user expects - when an object is no longer referenced, it is available to be cleaned up and the memory reused. This is simple to understand in the case of variables. For example, when we assign a value to a variable `x`, the value is referenced and so cannot be garbage collected. If we assign another value to `x` or remove `x`, the original value is now no longer referenced and becomes available for collection. What about when we have an object in a list, e.g. `x$y = 1:10`. The vector from 1 through 10 is referenced. If we assign a new value to the element `y` in `x`, then it will be avalable for garbage collection.

Another more complex situation is when we have two or more R objects that reference a shared value, e.g.

```
a = 1:10
x = a
y = a
```

R's computational model is that when this happens, the assignment to `x` and `y` causes the right hand side (the value of `a`) to be copied. But R does not do this in an effort to be efficient with memory. R only copies the value when it is changed in a variable that references it. So if we change an element of `y`, then we force a copy of the entire vector and change just that. `a` and `x` will still point to the same R vector.

How does this relate to the XML package ? The only thing of concern to us here is when we are dealing with internal/C-representations. The critical issue is simply this. Suppose we have an XML document created by either *`xmlParse`*() or *`newXMLDoc`*(), e.g.

```
doc = xmlParse("<foo><bar>Some text</bar></foo>", asText = TRUE)
```

Then we use some mechanism (e.g. XPath or direct manipulation of the nodes) to get access to one or more nodes, e.g.

```
nodes = getNodeSet(doc, "//bar")
```

Now, if we remove `doc` or simply return `nodes` from a function that has `doc` as a local variable, we have a problem. We are referencing individual nodes in a XML tree, but are no longer referencing the document

or overall tree. What's the issue? Well, since the document consumes memory, we must arrange to release it when we no longer need it. However, we do need part of the document, specifically the individual nodes for which we have a reference in an R variable. If we released the tree, we would release the nodes along with it. We cannot take a copy of the individual nodes as we need to be able to find, e.g., their ancestors, i.e. other parts of the tree. So we cannot free the memory associated with the document/entire tree while we still have references to the .

Now, while we have references to the individual nodes, all is well. But when we release all of these, we need to release the memory associated with the document. When we release of these nodes, we must not release the document unless there are no other nodes in the document being referenced in R variables.

The situation is further complicated when we create nodes outside of a document, e.g. with *newXMLNode*(). We might have

```
makeRow =
function(tbl = newXMLNode("table"))
{
  newXMLNode("tr", parent = tbl)
}
```

and the call

```
r = makeRow()
```

causes the **<table>** node to be unreachable from R, but needed as we can reach its child **<tr>**. Until `r` is released, we must ensure that `tbl` is not released. But when `tr` is released, `tbl` should be considered for release.

So we need a mechanism which can track these references to sub-nodes, determine when none exist and a document can be released. We must also be able to handle nodes which are not part of a document, but just a collection of related nodes that form a "virtual" document but have no C-level xmlDoc object associated with them.

There is a yet more complicated issue. While we may hold a reference to a node or a document in R, C code can chose to free the document (or a node). We should be in control of this and in most cases we are. However, when we are using another package that uses these libxml2 nodes, those libraries can chose to free documents or nodes.

A solution

An approach that we have implemented to address this issue is as follows. Let's take the case where we have an actual XML document (i.e. a C-level xmlDoc object with associated nodes). This and all its nodes are C-level objects, initially unrelated to R. When we parse this document, we add a C-level finalizer for the external pointer representing the C-level data structure in R. The finalizer is charged with freeing the document (via a call to *xmlFreeDoc*) if that is "appropriate". This finalizer routine is invoked when R has no more references to the pointer to this *xmlDoc* object. So when we explicitly remove the variable representing the parse document or it goes out of scope, this will be called during the next garbage collection R runs.

So when this finalizer is invoked, how can it determine whether it is "appropriate" to release the *xmlDoc*'s memory and hence all its nodes. The answer is that it looks at a field in this structure that gives the count of the number of nodes in the tree that are currently referenced by R variables. Each time we return a node from this document from C-level code back to R, we create a *new* external pointer containing the address of the node. Then we register a finalizer for that external pointer. The critical step is to increment the counter

for this node and also for the document in which it is housed. These counters are stored in the `_private` field of the *xmlNode* and *xmlDoc* structures. This is a pointer to a generic type and we allocate memory to store both the count and also a marker to allow us to identify that it is our private field and not one being used by another software layer on the nodes and documents. So three steps to return a node are

1. create an `externalptr` object
2. update the count (allocating space if necessary) in the node
3. update the count (allocating space if necessary) in the document
4. register the finalizer

The finalizer for a node decrements the counter for the node (if it exists), and decrements the counter for the document (if it exists). If the document has a zero count after this, the document can be freed. If the document still has a positive count, then there must be other nodes that are referenced by R variables and so we cannot release the document. When the finalizers for those other nodes are invoked (when the references are removed in R), the document will be decremented. Eventually, when the final node is released, the document count will be 0 and the document will be freed.

When the count for a given node gets to 0, the `_private` field is freed and set to NULL so we recognize that R is no longer managing that node.

Note that if we never have any references to specific nodes in a document, then the finalizer for the document will guarantee that it is released when the final R reference to the document is removed.

The following illustrates some of the aspects of this system.

```
library(XML)
doc = xmlParse("data/mtcars.xml")  # finalizer registered.
nodes1 = getNodeSet(doc, "//record[@id='Mazda RX4']") # get one node - finalizer r
.Call("R_getXMLRefCount", nodes1[[1]])  # ask for its reference count
nodes2 = getNodeSet(doc, "//record[@id='Mazda RX4']") # get the same node again, f
.Call("R_getXMLRefCount", nodes2[[1]])
.Call("R_getXMLRefCount", nodes1[[1]])

rm(doc)
gc()
rm(nodes1)
gc()
.Call("R_getXMLRefCount", nodes2[[1]])
rm(nodes2)
gc()

rm(nodes2)
gc()
```

Manging Nodes with No Document

We can create nodes without a document and add them as children, siblings and even the parent of other nodes. In this case, we don't have a convenient central location to which all nodes have a reference, i.e. the

doc field which is part of each *xmlNode*. Instead, when we return a reference to a node, we repeat the steps of creating the external pointer, incrementing the counter for that node, registering the finalizer, but do not of course increment the counter for the non existent document.

When we remove an R reference to a node, the finalizer is run for that unique `externalptr` object in R. That decrements the count. If that results in the count being 0, then we remove the counter information (i.e. free the value of the `_private` field and set it to NULL). Next we see if it is appropriate to free the node. We determine this by seeing if a) it has no parent, and b) if none of its descendants (children, their children, and their children, etc.) have a count that it is non-zero. (For large trees, this can involve traversing many nodes.) If there is no parent and no referenced node, then we can free that node.

Problems

Reference counting can be problematic for garbage collection. Most problematic is the potential for cycles where a variable **a** references **b** and **b** refers to **a** and neither is garbage collected eventhough neither is needed. Fortunately, XML documents are trees and not graphs and cycles don't occur.

We can get into trouble with this management scheme in quite specific circumstances. Because the reference counting mechanism is known only to the XML package (and R indirectly), other code may not honor the conditional freeing of the nodes and document. For example, if we create an XML document (an *xmlDoc*) and return it to R as an `externalptr` with a finalizer. We can ignore the case that any nodes are referenced from R. Suppose this document is explicitly freed (via *xmlFreeDoc*) by some code. Later, when the R variable referencing the document goes out of scope, the `externalptr` will be garbage collected and the finalizer run. This will check the count on the *xmlDoc* in its private field to determine if it should free the document. Unfortunately, at this point, the memory pointed to by the `externalptr` will be made up of random content since it was freed. So we will be interpreting nonsense. There may be a (very small) memory leak as if we had allocated memory to hold the count in the `_private` field, that won't have been released in the call to *xmlFreeDoc*.

The same issue arises for references in R to *xmlNode* objects. If we have such references and the document or node is explicitly freed by a 3rd party, the finalizer will be looking at random bits in memory and is likely to interpret them incorrectly and crash or seriously corrupt memory.

These disastrous situations only occur when we mix releasing memory across two regimes - the R garbage collector and another system. As long as this does not happen, the reference counting mechanism makes certain to avoid freeing memory still in use by R and does release memory that is not in use by R.

There is the possibility of using R's memory manager and garbage collector to allocated and manage memory in libxml2. This would put the two systems under one regime, including libxslt. This would avoid these problems. We need to investigate the precise details of this.

By-passing the Memory Management

The reference counting memory management approach works well and removes the need for the R programmer to concern herself with any of the details of working with C-level/native/internal data structures. One issue that can arise however is efficiency. Consider the worst case situation where we have nodes without a document. We might construct a top-level node first and assign it to an R variable. Then we would use `newXMLNode()` and related functions to create other nodes. It is common to specify the `parent` argument in such calls to immediately add the newly created node as a child of the parent. Regardless, by default,

each of these functions will return an *externalptr* and add a finalizer for the node. Then, as these are garbage collected, they will gradually reduce the counts to 0 for each node and finally the top-level node can be released. There is a great deal of traversing of this sub-tree to determine if the top-most node can be released. Yet this is often done in a context in which the R programmer knows that the top-most object is to be retained and the descendant nodes are of no direct interest outside the scope of the computation. We may want the *externalptr*, but not the extra computations of the finalizers and garbage collection. Instead, we may just know that the top node will be added to a document and that there is no need for all this checking of reference counts.

This same concern arises when extracting nodes from a document with *getNodeSet*() when we know that the document reference in R will continue to exist after we have finished with the nodes. Similarly, when we retrieve a list of child nodes via *xmlChildren*(), we may not want to put these references under our reference counting scheme as we may know that the parent will not be released before we discard these references. So a function such as

```
function(node)
{
   sapply(xmlChildren(node), xmlAttrs))
}
```

does not need to use reference counting for the child nodes returned by *xmlChildren*(). However,

```
function(node, labels = c("book", "article"))
{
   xmlChildren(node)[ names(node) %in% labels ]
}
```

potentially does as we are returning child nodes which may persist pass the lifetime of their parent node.

In order to deal with this, many of the R functions that can return an *externalptr* referencing an *xmlNode* have a *addFinalizer* parameter. This can take various different values:

1. **TRUE** to use reference counting on the node for this reference
2. **FALSE** to not use reference counting for this reference
3. **NA** to use the current default
4. a character string naming a C routine to use as the finalizer, or a *NativeSymbolInfo* object that contains the resolved C routine to use

When we know that we don't need to put the returned references under reference counting control, we can explicitly specify **FALSE** for this argument. So we could write our first function above as

```
function(node)
{
   sapply(xmlChildren(node, addFinalizer = FALSE), xmlAttrs))
}
```

Basically, if you know when you are getting a reference to an XML node (an *externalptr*) that it is guaranteed not to be freed for the duration of its lifespan in R, then you can use `addFinalizer = FALSE`. This saves a lot of (admittedly fast) computations.

When writing functions that manipulate XML nodes and documents, it is good practice to add an *addFinalizer* parameter to the function and use this in the calls to the functions in the XML package that accept

such a parameter. This allows callers of your functions to control how the references are managed and to make use of knowledge of their context about the lifespan of the references.

If your function creates nodes, it is good to allow the caller specify a parent for the node(s) and have this default to **NULL**. In this case, you can often have an *addFinalizer* parameter also that has a default of `is.null(parent)`. So a function might look like

```
foo =
function(..., parent = NULL, addFinalizer = is.null(parent))
{
   lapply(list(...),
            function(x)
             newXMLNode("li", x, parent = parent,
                        addFinalizer = addFinalizer))
}
```

When in doubt, use the default memory management and do not override it.

Removing Finalizers

There are occasions when we have established reference counting and finalizers for *externalptr* objects referencing *xmlNode* or *xmlDoc* objects and need to remove these. This occurs when we are calling functions that use the reference counting approach and do not allow us to override it. After we use these functions, we might pass the document or node to software that simply frees it. Then we have the situation discussed above in the section called "Problems"(page 4) In some situations, one can clone the document or the nodes and pass those to the other software. However this is not always possible as we might have references to nodes within a document that is created earlier by that other software and to which we have references via callbacks, e.g. Sxslt and processing an XML document with an XSL transformation involving callbacks to R functions that implement XSL extension functions.

So now that we know we can get into such a situation, we have a problem. When the other software frees the document and its sub-nodes, our finalizers will be invoked subsequently and be interpreting random memory contents. The key to this situation is to remove the finalizers on these R *externalptr* objects. Prior to version 2.14.0 (or more to the point the development version of R post-dating 2.13.2), there was only a mechanism to add finalizers. We have added[1] routines in R to do this. One can remove a finalizer by providing an *externalptr* object or by providing an external pointer value (which has to be in a new *externalptr* object). We provide a routine (*R_xmlNode_removeFinalizers*) in the XML package to use the latter approach to remove finalizers, if they exist, for each of the nodes in an XML document or top-level node. This works recursively, removing finalizers on itself and each of the sub-nodes.

[1] Code has not been committed yet, Mar 24 2011