Using the SAX Interface of LibXML

James Henstridge

james@daa.com.au

Most users of the libxml (aka gnome-xml) library tend to use the DOM style tree interface for reading documents. This is generally quite an easy interface to use, but can use quite a bit of memory. An alternative is to use the SAX interface in libxml, which is a port of the Simple API for XML library for Java.

This article is aimed at people who understand and have used the libxml DOM style interface and want to explore the SAX interface. Some examples are biased toward use in GTK+/GNOME programming.

Introduction and Concepts

Most people will agree that the most intuitive representation for arbitrary XML data is a tree. Libxml provides a nice API to construct a tree from an XML file, which makes it very easy to use. But in some cases, the tree representation does not match the internal representation you wish to use in your program, so it may not be the best choice. You may end up using libxml to construct the tree, extracting the data and then throw away the tree.

In these cases libxml's SAX API may be a better choice. Note that there are a few drawbacks to the use of this API though:

- SAX parsers generally require you to write a bit more code than the DOM interface.
- Unless you build a DOM style tree from your application's internal representation for the data, you can't as easily write the XML file back to disk. This is not a concern if your program only reads XML files, and does not write them.
- You may have to redesign or rethink your file loading routines.

It is not all bad however. There are benefits to the use of the SAX API:

- The DOM tree is not constructed, so there are potentially less memory allocations.
- If you convert the data in the DOM tree to another format, the SAX API may help remove the intermediate step.
- If you do not need all the XML data in memory, the SAX API allows you to process the data as it is parsed.

The SAX Interface

As you know, the DOM style interface (ie. what is returned by `xmlParseFile` or `xmlParseMemory`) produces a tree of xmlNode structures that can then be walked to extract the data.

The SAX interface works quite differently on the other hand. You instead pass a number of callback routines to the parser in a xmlSAXHandler structure. The parser will then parse the document and call the appropriate callback when certain conditions occur.

Some of the more useful callbacks are `startDocument`, `endDocument`, `startElement`, `endElement`, `getEntity` and `characters`. Most of these are quite self explanatory. The `characters` callback is called when characters outside a tag are parsed.

As you can see, the code for a SAX style parser seems almost inside out, when compared with the DOM style tree method. For this reason, a state machine aproach is useful.

It is interesting to note that the `xmlParseFile` function and friends are all implemented in terms of the SAX interface, so no power is lost by using this interface.

Some Examples

To give you some idea of where SAX may be useful, some examples may help.

An Array of Numbers

Consider an XML document structured like follows:

```
<?xml version="1.0"?>
<array>
  <number>0</number>
  <number>1</number>
  ...
  <number>42</number>
</array>
```

The file describes an array of numbers. In this case, the most appropriate data structure to represent the information as would be an array – not a tree. By using the SAX interface, we can write the numbers directly to an array rather than using the DOM tree as an intermediate format.

As I said earlier, state machines make using the SAX interface much easier. At any one time, you are in one state. The `startElement` and `endElement` callbacks cause the state to change. You will also want to perform some other actions during state changes – in this case, build the array.

For this example, we can use four states:

START
INSIDE_ARRAY
INSIDE_NUMBER
FINISH

In the `startDocument` callback, we would initialise any variables that are required during parsing. This would include the array to hold the numbers (a glib GArray would be useful), and a buffer to hold character data (a glib GString is a good choice here). It would also set the initial state to START. the `endDocument` callback, we can free these variables (we would leave the array though).

The interesting code is in the `startElement` and `endElement` callbacks. What they do will depend on the current state, and the element name that is being opened or closed.

For the array example, the `startElement` function would act as follows:

- If we are in the START state and the element name is array, then change to the INSIDE_ARRAY state. Any other names would be an error
- If we are in the INSIDE_ARRAY state, and the element name is number, change to the INSIDE_NUMBER state. We would also zero the character data buffer. Any other element would be an error in this state.
- If we are in the INSIDE_NUMBER or FINISH states, it is an error.

The `characters` function would append the character data to the buffer if it was in the INSIDE_NUMBER state. If it is in any other states, the data could be discarded.

The `endElement` function would act as follows:

- If we are in the INSIDE_NUMBER state, the character data buffer should hold the number. We convert the string to an integer and append it to the array. We then change to the INSIDE_ARRAY state.
- If we are in the INSIDE_ARRAY state, change to the FINNISH state.
- If we are in the START or FINNISH states, an error has occured

Note that we know what to do in the `endElement` function even without looking at the element name. By using a state machine model like this, we can narrow down the number of possible element names, which reduces string comparisons as well.

Large Repetitive XML Files

Sometimes we have XML files with many subtrees of the same format describing different things. An example of this is the `fullIndex.rdf.gz`. The file contains repeating sections like follows:

```
  ...
  <RDF:Description about="ftp://rpmfind.net/linux/redhat/redhat-6.0/i386/RedHat/RPMS/emacs-
X11-20.3-15.i386.rpm">
    <RPM:Name>emacs-X11</RPM:Name>
    <RPM:Summary>The Emacs text editor for the X Window System.</RPM:Summary>
  </RDF:Description>
  <RDF:Description about="ftp://rpmfind.net/linux/Mandrake/6.0/Mandrake/RPMS/emacs-
el-20.3-14mdk.i586.rpm">
    <RPM:Name>emacs-el</RPM:Name>
    <RPM:Summary>The sources for elisp programs included with Emacs.</RPM:Summary>
  </RDF:Description>
  ...
```

One operation that we might want to do is to search for a package by name. For this type of operation, at any one time, we are only concerned with one subtree in the document – the others need not be in memory. For this reason, a SAX based parser would be a good idea.

With the DOM tree aproach, the memory usage of the program will increase as the size of the index file increases. With the SAX based aproach, the memory usage should be fairly constant despite changes in the size of the index.

This benefit is particularly useful when parsing XML documents of sizes similar to the RDF dumps of the Open Directory Project (http://www.dmoz.org). The overhead of the DOM tree can become unacceptable when parsing 35 megabyte XML files.

For a working example of this sort of situation, see the XML parser in `/gnorpm/find/search.c` (http://cvs.gnome.org/lxr/source/gnorpm/find/search.c).

Simple Tree Structures

Sometimes it may make sense to use the SAX interface even if the data in the XML file is inherently tree based.

The DOM style interface of libxml is designed to be able to represent any well formed XML document. But the save format of most applications generally uses only a subset of XML. For instance, GLADE does not use attributes for any elements, so you could argue that the support for attributes in the DOM interface is bloat when used with GLADE.

Since the XML you are reading in has some known constraints, it is usually possible to store the information in more compact structures.

I am looking at implementing something like this in libglade. In this situation, the changeover looks like it will decrease the memory requirements of the internal structures by a factor of four, and has increased the speed of the parser slightly.

You can look at the source for the new libglade SAX based parser at `/libglade/test/saxp.c` (http://cvs.gnome.org/lxr/source/libglade/test/saxp.c). There is also a diagram that describes the transitions between the different states in the parser at `/libglade/test/states.dia` (http://cvs.gnome.org/lxr/source/libglade/test/states.dia).

In this particular type of situation, it is worth spending a bit more time deciding between DOM style and SAX interfaces. If you keep with the DOM interface, your program will probably use a bit more memory, but it has the advantage that you can modify the tree, and then write the XML file back to disk with a single function call. If you switch over to a different representation, you will need to either convert the information to the DOM style tree representation and then dump it to a file, or write your own output routines.

Another deciding factor is laziness. If you use the SAX interface, you will need to write some parsing code. On the other hand, the DOM interface does this for you.

In the libglade case, memory usage and speed were considered important, and the XML data did not have to be written back to a file, so choice of a SAX parser was relatively easy.

Implementing a SAX Based Parser

Now you should have a pretty good idea about what a SAX parser is, and how you might implement one using a state machine design. Now it is time to learn how to implement the parser using libxml's C translation of the SAX interface.

The Basics

As stated before, you use the SAX parser by passing a number of callback routines stored in a xmlSAXHandler structure to one of the SAX parser routines. Here is the prototype for the structure:

```
typedef struct xmlSAXHandler {
    internalSubsetSAXFunc internalSubset;
    isStandaloneSAXFunc isStandalone;
    hasInternalSubsetSAXFunc hasInternalSubset;
    hasExternalSubsetSAXFunc hasExternalSubset;
    resolveEntitySAXFunc resolveEntity;
    getEntitySAXFunc getEntity;
    entityDeclSAXFunc entityDecl;
    notationDeclSAXFunc notationDecl;
    attributeDeclSAXFunc attributeDecl;
    elementDeclSAXFunc elementDecl;
    unparsedEntityDeclSAXFunc unparsedEntityDecl;
    setDocumentLocatorSAXFunc setDocumentLocator;
    startDocumentSAXFunc startDocument;
    endDocumentSAXFunc endDocument;
    startElementSAXFunc startElement;
    endElementSAXFunc endElement;
    referenceSAXFunc reference;
    charactersSAXFunc characters;
    ignorableWhitespaceSAXFunc ignorableWhitespace;
    processingInstructionSAXFunc processingInstruction;
    commentSAXFunc comment;
    warningSAXFunc warning;
    errorSAXFunc error;
    fatalErrorSAXFunc fatalError;
} xmlSAXHandler;
typedef xmlSAXHandler *xmlSAXHandlerPtr;
```

To start off with, we can set all these functions to NULL. If we use a NULL SAX parser like this, then we will have a parser that only checks the well formedness of a document. By adding a few callbacks, we can get it to do just about anything.

If you do not care about reentrancy of your parser, you can save state between callbacks in global variables. If you choose this approach, you could use one of the following functions:

```
#include <parser.h>
xmlDocPtr xmlSAXParseFile(xmlSAXHandlerPtr sax, const char *filename, int
recovery);
xmlDocPtr xmlSAXParseMemory(xmlSAXHandlerPtr sax, char *buffer, int size,
int recovery);
```

In these functions, you will most likely ignore the return type (it will probably contain a NULL or garbage). However, if you want to write a reentrant parser, you will need to make some changes. A `user_data` parameter is passed to all SAX callbacks, which can be used to pass state information between callbacks. Unfortunately, with the current libxml API, there is not an easy way to set this parameter, so you will need to include a rewritten version of `xmlSAXParseFile` in your program. Daniel says that he will probably change the above functions in a future version to be more useful in this case. Until then, you will probably want to use a function like this:

```
#include <parser.h>
#include <parserInternals.h>

int myXmlSAXParseFile(xmlSAXHandlerPtr sax, void *user_data, const char *file-
name) {
    int ret = 0;
    xmlParserCtxtPtr ctxt;

    ctxt = xmlCreateFileParserCtxt(filename);
```

```
    if (ctxt == NULL) return -1;
    ctxt->sax = sax;
    ctxt->userData = user_data;

    xmlParseDocument(ctxt);

    if (ctxt->wellFormed)
        ret = 0;
    else
        ret = -1;
    if (sax != NULL)
        ctxt->sax = NULL;
    xmlFreeParserCtxt(ctxt);

    return ret;
}
```

This function could then be used like so:

```
static xmlSAXHandler my_handler {
    ...
};

struct ParserState {
    RetVal return_val;
    StatesEnum state;
    ...
};

RetVal parse_xml_file(const char *filename) {
    struct ParserState my_state;

    if (myXmlSAXParseFile(&my_handler, &my_state, filename) < 0) {
        free_ret_val(my_state.return_val);
        return NULL;
    } else
        return my_state.return_val;
}
```

In this example, we expect the `startDocument` SAX handler to initialise the ParserState structure passed to it, and the `endDocument` to free its members, but leaving *`return_val`* so that it can be used later.

The startDocument and endDocument Callbacks

These callbacks are generally used to perform some initialisation and deinitialisation for your parser callbacks. Their prototypes are as follows:

```
void startDocument(void *user_data);
void endDocument(void *user_data);
```

It should be fairly self explanatory how to write these functions.

The characters Callback

The `characters` callback is called when there are characters that are outside of tags get parsed. Its prototype is as follows:

```
void characters(void *user_data, const CHAR *ch, int len);
```

The CHAR type is an alias for char. It is used so that it will be easier to add unicode support to the parser at a later date. For all intents and purposes though, you can think of *ch* as an array of char's. Note that the character data is not necessarily nul terminated. This is so that libxml does not need to copy the character data out of its internal buffers before passing it to the callback.

In your callback, you will probably want to copy the characters to some other buffer so that it can be used from the endElement callback. To optimise this callback a bit, you might adjust the callback so that it only copies the characters if the parser is in a certain state. Note that the `characters` callback may be called more than once between calls to `startElement` and `endElement`.

The startElement and endElement Callbacks

These callbacks are where most of the state machine logic will go into these two callbacks. Their prototypes are:

```
void startElement(void *user_data, const CHAR *name, const CHAR **attrs);
void endElement(void *user_data, const CHAR *name);
```

In these callbacks, the *name* parameter is the name of the element. The *attrs* parameter contains the attributes for the start tag. The even indices in the array will be attribute names, the odd indices are the values, and the final index will contain a NULL.

In most parsers, as well as making state transitions in these callbacks, you will probably also collect the data in the XML file. In the `startElement` callback, you will often allocate structures to hold the data. In the `endElement` callback, you will usually interpret the character data collected by the `characters` callback and put the data in one of the structures allocated by `startElement`. The `endElement` callback may also free some of the intermediate structures if it is no longer needed.

The getEntity Callback

You may have been wondering how entities (eg <, etc) are handled by the SAX interface. This is done by the `getEntity` callback:

```
xmlEntityPtr getEntity(void *user_data, const CHAR *name);
```

The xmlEntity structure holds some information about the entity. This structure will not be freed by the parser, so it makes sense to create these structures once, and then pass a pointer to the appropriate one when this function is called. After calling `getEntity`, the expansion of the entity is passed to the `characters` callback. This way, you do not need to worry about decoding entities anywhere else in your callback routines.

If your XML document only contains the standard entities (<, >, ', " and &), then it is possible to write a very short implementation for this callback:

```
static xmlEntityPtr my_getEntity(void *user_data, const CHAR *name) {
    return xmlGetPredefinedEntity(name);
}
```

For most parsers, this will be sufficient.

Structural Errors

If there are structural errors in the XML file, the parser will call one of three error callbacks: `warning`, `error` or `fatalError`.

If you want to pass these errors to the standard glib logging functions, you might want to use an implementation something like this:

```
static void my_warning(void *user_data, const char *msg, ...) {
    va_list args;

    va_start(args, msg);
    g_logv("XML", G_LOG_LEVEL_WARNING, msg, args);
    va_end(args);
}

static void my_error(void *user_data, const char *msg, ...) {
    va_list args;

    va_start(args, msg);
    g_logv("XML", G_LOG_LEVEL_CRITICAL, msg, args);
    va_end(args);
}

static void my_fatalError(void *user_data, const char *msg, ...) {
    va_list args;

    va_start(args, msg);
    g_logv("XML", G_LOG_LEVEL_ERROR, msg, args);
    va_end(args);
}
```

Note that libxml is not a validating parser, so only structural errors will be picked up. So any validation of the format will have to be done by your parser routines.

How to Handle an Expanding File Format

With most applications, you will want to add to the XML file format as you add features to the application. For this reason, you will want to code your callbacks so that they don't barf on an unknown or unexpected tag.

With the DOM style interface, if you come to a node with an unexpected name, you will usually ignore it, and the subtree under it. It is probably a good idea to use a similar process for a SAX based parser.

To implement this sort of error recovery, we will need an extra state for the parser – UNKNOWN. We will also need to pass two extra variables in the *`user_data`* parameter to the callbacks – prev_state and unknown_depth.

When we hit an unknown element in the `startElement` callback, we can save the current state to prev_state, and then change the state to UNKNOWN, and set unknown_depth to 1. If `startElement` is called while in the UNKNOWN state, we increment the unknown_depth variable.

In the `endElement` callback, if we are in the UNKNOWN state, decrement unknown_depth. If unknown_depth is zero, change the state to prev_state. The `characters` callback should probably return immediately if in the UNKNOWN state as well.

Using this sort of logic, it should be possible to ignore unknown sections of the document quite easily. The UNKNOWN state is also useful when writing the parser. This way you can test out portions of the parser before it is complete.

Conclusion

This tutorial should have given you a good idea about how to think about writing SAX parsers, and how to implement them with libxml. If you want more information about the other callbacks, look at the libxml API documentation.

Another place for information on SAX is the Simple API for XML (http://www.megginson.com/SAX/index.html) page. It is concerned with the original java implementation, but many of the concepts should be useful with libxml.