Difference between DOM and SAX.

By Yvan Ayaay, Technical Support Engineer, 4D Inc.
TN 06-14

Introduction

4th Dimension supports two kinds of parsing modes for XML documents: DOM (Document Object Model) and SAX (Simple API XML). DOM and SAX are different sets of XML commands for manipulating XML files. Both have different scopes and programming styles. Their characteristics offer advantages and disadvantages. In this technote, the differences between the two will be discussed. It will be shown how to create and parse XML documents in DOM and in SAX to emphasize their distinction. This technote will require a basic understanding of XML. You can check the technote "XML: An Introduction" for more information.

Overview

DOM (Document Object Model) and SAX (Simple API XML) are two different modes in manipulating XML documents that 4th Dimension offers. XML (Extensible Markup Language) is a standardized data format aimed to reach universally interchangeable data. The DOM mode and SAX mode are two sets of XML commands that differ in their flow of parsing, limitations, and programming approach. For instance, DOM parses and builds an XML tree in memory and it allows accessing nodes in random either forward or backward. While SAX does not store XML in memory but follows a top to bottom traversing. In this regard, though the SAX mode appears to be more memory efficient, the DOM mode can be more flexible as the SAX XML abides a linear approach. Nevertheless, deciding which mode to use is really a programming preference.

The following sections depict the dissimilarities between the DOM and SAX XML commands. The advantages and disadvantages of these commands are discussed. Then, examples in the creation and parsing of XML document in DOM and SAX modes will be illustrated.

Advantages and Disadvantages

The SAX and DOM XML commands are two very different modes in parsing XML documents. Both have advantages and disadvantages:

Memory Usage: DOM stores the entire structure tree in memory. Because of this, access to each element of the source is extremely fast. However, it can be a limitation when dealing with very large XML documents as it might not fit into the available memory. SAX, on the other hand, does not store XML in memory. It handles an XML document in an event streaming approach. Because of this, SAX is not restricted to the size of XML document and to the amount of memory available.

Parsing Traversal: SAX parses an XML document node by node in a linear fashion. It follows a top to bottom approach returning an event for every element it encounters such as begin tag, end tag, etc. It does not allow random access manipulation of an XML document. When parsing an existing XML document, the document needs to be opened in read-only. Because of this, you would not be able to modify elements, their values, and attributes with SAX. Conversely, DOM builds an XML tree in memory and can parse it in a random approach. This mode allows you to traverse the XML tree in any direction. You can search tags backward or forward in the XML tree and then evaluate data that they hold. DOM has many commands that allow you to modify elements, their values, and attributes.

Choosing one over the other: Deciding between DOM and SAX can be based on the factors discussed above. DOM will be nice for smaller XML files since it stores the entire XML document in memory. But for very large XML documents, SAX would be ideal. If a data needs to be extracted from an XML file once, SAX could be considered because of its serial approach. However, if the data need to be randomly selected from the XML file, DOM will be ideal. Because of DOM's random approach, it will be a better a choice in dealing with more complicated XML documents. The programming styles of these two modes are significantly different. So, it can be a programming preference as well—whichever you like better.

Creating an XML Document

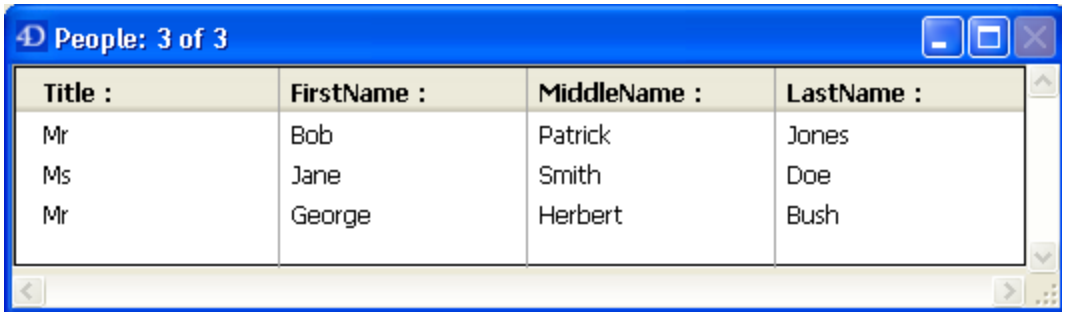
Both the DOM mode and SAX mode allow you to create an XML document. Their approach in creating the XML structure is much different. DOM builds the structure in memory, uses references to element nodes to properly build the hierarchy and then save to file. Whereas SAX creates an XML document first and uses the document reference to build the hierarchy in a linear fashion.

When creating an XML document using DOM, you first create an XML tree in memory--creating elements and putting values--and then save the XML tree

into a file. When creating elements, references to the elements are stored and taken into account to build a well-formed XML hierarchy. These references are used to create child and sibling elements.

In contrast, when creating an XML document using SAX, an XML document is first created and the elements and values are created. Just like creating and writing to a document in 4th Dimension, a document reference is used. Unlike DOM which can use many references to properly build the XML tree, only one reference (document reference) is used in SAX wherein the elements and values are created sequentially from top to bottom. Once all the elements are created, the document is closed.

To better illustrate how to create an XML document in these two modes, let us consider the people table with records as shown below:



Let us say, you want to create an XML document that shows people names as shown below:

```
<?xml version="1.0"  encoding="UTF-16"  standalone="no"  ?>
<people>
  <name title="Mr">
      <first>Bob</first>
     <middle>Patrick</middle>
     <last>Jones</last>
   </name>
  <name title="Ms">
      <first>Jane</first>
     <middle>Smith</middle>
     <last>Doe</last>
   </name>
  <name title="Mr">
      <first>George</first>
     <middle>Herbert</middle>
     <last>Bush</last>
   </name>
</people>
```

The title field is the attribute of the name tag and the rest of the fields--the first, middle, and last--are child elements of the name element.

In DOM mode

To create the above XML in DOM, you can run the code below:

```
`Method: XMLCreateDOM
C_STRING(16;vRootRef;vElemRef)
C_TEXT($rootElem;$vElem1;$vElem2;$vElem3)
C_TEXT($mtitle;$myfilepath)
$rootElem:="people"
$myfilepath:="C:\\XMLStuff\\TestPeople1.xml"  `  name of the XML file to create.
vRootRef:=DOM Create XML Ref($rootElem)
ALL RECORDS([People])
DOM SET XML OPTIONS(vRootRef;"UTF-16";False) `Set  encoding and stand alone values.
 `Go  through all records in People table.
For (i;1;Records in selection([People])
  $MainElem:="/people/name"
  $mtitle:=[People]Title
   `Create  name element and the title attribute
  vElemRef:=DOM Create XML element(vRootRef;$MainElem;"title";$mtitle)
  $vElem1:="first"
   `Create  first name element which is a child of name element.
  vElemRef1:=DOM Create XML element(vElemRef;$vElem1)
  $fname:=[People]FirstName
   `Set  first name value
  DOM SET XML ELEMENT VALUE(vElemRef1;$fname)
  $vElem2:="middle"
   `Create  middle name element which is also child of name element.
  vElemRef2:=DOM Create XML element(vElemRef;$vElem2)
  $middle:=[People]MiddleName
   `Set  middle name value
  DOM SET XML ELEMENT VALUE(vElemRef2;$middle)
  $vElem3:="last "
   `Create  last name element
  vElemRef3:=DOM Create XML element(vElemRef;$vElem3)
  $last:=[People]LastName
   `  set last name value
  DOM SET XML ELEMENT VALUE(vElemRef3;$last)
  NEXT RECORD([People])
End for
` save XML to file.
DOM EXPORT TO FILE(vRootRef;$myfilepath)
```

As you can see in the code, an empty XML tree is first created with "people" as the root element. Then a sub-element "name" is created with a title attribute using the reference of the "people" element. The "name" should contain the first, middle, and last name elements and values. So, to create these child elements, the reference of the "name" element is used to create the "first", "middle", and "last" elements. The respective references for these child elements are, then, used to create their respective values. To create a "name" node for every record, the reference of the root element "people" is used in a loop. Each time, a new "name" element with its child elements and values is appended to the XML structure. At the end, the XML tree is saved

into a file. As shown in this mode, the element references are used to build the desired XML structure.

In SAX mode

To create the same XML document above using SAX, you can perform the code below:

```
`Method: XMLCreateSAX
C_TIME($docref)
C_TEXT($rootElem;$vElem1;$vElem2;$vElem3)
C_TEXT($mtitle;$myfilepath)

ALL RECORDS([People])
$myfilepath:="C:\\XMLStuff\\TestPeople1.xml"
$DocRef:=Create document($myfilepath)
SAX SET XML OPTIONS($DocRef;"UTF-16";True)  ` set encoding and stand alone values
$rootElem:="people"
`Create root element "people". A begin tag of the same name is added.
SAX OPEN XML ELEMENT($DocRef;$rootElem)
  `Go through all the records in table
For ($i;1;Records in selection([People]))
  $MainElem:="name"
  $mtitle:=[People]Title
    `Create parent element "name" with the title attribute. A begin tag "name" is added.
  SAX OPEN XML ELEMENT($DocRef;$MainElem;"title";$mtitle)
  $vElem1:="first"
    `Create child element "first". Begin tag "first" is created.
  SAX OPEN XML ELEMENT($DocRef;$vElem1)
  $fname:=[People]FirstName
    `Set value for the first tag.
  SAX ADD XML ELEMENT VALUE($DocRef;$fname)
  `Close last element created. An end tag "first" is added.
  SAX CLOSE XML ELEMENT($DocRef)
  $vElem2:="middle"
  `Create a child element "middle". Begin tag "middle" is created.
  SAX OPEN XML ELEMENT($DocRef;$vElem2)
  $middle:=[People]MiddleName
    `Set value for middle tag.
  SAX ADD XML ELEMENT VALUE($DocRef;$middle)
    `Close the "middle" tag. An end tag "middle" is added.
  SAX CLOSE XML ELEMENT($DocRef)
  $vElem3:="last"
  `Create a child element "last". Begin tag "last" is created.
  SAX OPEN XML ELEMENT($DocRef;$vElem3)
  $last:=[People]LastName
    `Set value for last tag.
  SAX ADD XML ELEMENT VALUE($DocRef;$last)
    `Close the "last" tag. An end tag "last" is added.
  SAX CLOSE XML ELEMENT($DocRef)
```

```
    `Close the "name" tag. An end tag "name" is added.
   SAX CLOSE XML ELEMENT($DocRef)
   NEXT RECORD([People])
End for
  `Close the "people" tag.
SAX CLOSE XML ELEMENT($DocRef)
CLOSE DOCUMENT($DocRef)
```

As you can see in the code above, a document is first created. The document reference is then used to create the elements in a stream-like fashion from top to bottom. Every time an element is created using the SAX OPEN XML ELEMENT command, a begin tag for that element is created. You will have to add a matching close tag at the proper location. This is necessary to properly build the hierarchy. The SAX CLOSE XML ELEMENT command adds a closing tag to the last element created by the SAX OPEN XML ELEMENT command. To create a child element, you will have to create another XML element without closing the parent element. And to create a sibling element, you will have to close the current element and then create another element. The opened elements should be matched with close tags to build the desired XML structure. The CLOSE DOCUMENT command closes the XML document.

Opening and Traversing XML Documents

The DOM and SAX commands open an XML document and parse it quite differently. DOM opens the document and builds an XML tree in memory. You can find elements using the XPath notation (http://www.4d.com/docs/CMU/CMU10099.HTM) starting from a reference (element reference). You can retrieve and modify the names and values of the elements. SAX, on other hand, traverses the document in a linear fashion. SAX returns an event as it reads the document from top to bottom. The document to be parsed by SAX needs to be open in read-only mode. Thus, when parsing an existing XML document using SAX, you will not be able to modify values and element names.

Parsing in DOM mode

DOM can open a document using the DOM Parse XML Source command which returns a reference to the document. There are many DOM commands that let you read, parse and write the elements and attributes (http://www.4d.com/docs/V6U/V6U00067.HTM). You can find and count elements and get their values and modify element names, its value, and attributes. Also, you can traverse through the XML structure with commands that allows you to directly access the parent, next and previous sibling, and first and last child elements. Using the XPATH notation (designed to navigate XML structures), you can directly set elements within an XML structure without having to indicate the full access path.

Below is an example code that illustrates how to find all instances of an element and return its values. The method below takes three parameters. The first parameter is the name of the element to find, the second parameter holds the pointer to an array that will store the values of the elements found, and the third parameter is the path to the XML document to be parsed. You pass the name of the element and all the instances of this element are looked up and their values stored in an array.

```
  `Method: DOMGetElemVal
  `Description:  This method returns the values of the element you want to find.
  `              You pass the name of the element ($1), a pointer to the array ($2) that will
contain the values,
  `              and the path to the XML file ($3) you want to parse.
C_TEXT($1;$fElem;$pathfile;$3)
C_POINTER($2)
$pathfile:=$3
   ` $pathfile:="C:\\XMLStuff\\TestPeopleDOM2.xml"
ARRAY TEXT(ElemVals;0)
  `Parse XML document
$ref1:=DOM Parse XML source($pathfile)
  `Find first element "name" and return reference to this node
vName:=DOM Find XML element($ref1;"/people/name")
$fElem:=$1
  `loop until all element values are found
While ((OK=1) & (vName#""))
    `find reference to the element to find
   vFirst:=DOM Find XML element(vName;"/name/"+$fElem)
    `get value of this element
   DOM GET XML ELEMENT VALUE(vFirst;value)
    `save value to array
   APPEND TO ARRAY(ElemVals;value)
    `go to the next sibling "name" element.
   vName:=DOM Get Next sibling XML element(vName)
End while
  `Copy array with values to passed array.
COPY ARRAY(ElemVals;$2->)
```

As you can see from the code above, the XML document is first parsed using the DOM PARSE XML Source command. Then the element “name” which contains the target element is looked up. The DOM Find XML element command is used to find the reference to the “name” node and this reference is used to find the child element. Once the child element is found, the element value is retrieved using the DOM GET XML ELEMENT VALUE command and is stored in an array in the code above. To go to the next node of the element “name”, the DOM Get Next sibling XML element command is used. The reference to the first element “name” is used to go to the next sibling node. The process is repeated until all the element “name” are processed.

Parsing in SAX mode

SAX, in contrast, opens an XML document using the OPEN DOCUMENT command. The document should be opened in read-only mode to avoid any conflict between 4th Dimension and the Xerces library when you open "standard" and XML documents simultaneously. If you execute a SAX parsing command with a document open in read-write mode, an alert message is displayed and parsing is impossible.

Traversal of the XML structure needs to be carried in a linear fashion from the beginning to the end of the document when using SAX. To traverse and move through the document, the SAX Get XML node command is used. This command returns events for every SAX event encountered. Below is a list of events that could be returned:

Constant	Type	Value
XML Start Document	Longint	1
XML Comment	Longint	2
XML Processing Instruction	Longint	3
XML Start Element	Longint	4
XML End Element	Longint	5
XML DATA	Longint	6
XML CDATA	Longint	7
XML Entity	Longint	8
XML End Document	Longint	9

Based on the type of event returned, you can then check and retrieve information such as element name, value, attributes, entity, comments, CDATA etc. Every time the SAX Get XML node command is used, you will have to check the event returned to retrieve the desired information. For instance, to find a specific element, you will have to call the SAX GET XML node until an XML Start Element is encountered, then you can then use the SAX GET XML ELEMENT command to retrieve the element name. You can then verify if that is the element you are looking for.

Below is a sample code that illustrates how to find elements and retrieve values using SAX. This method is the same as the example code in the parsing in DOM mode only that is implemented in SAX.

```
  `Method: SAXGetElemVal
  `Description: This method finds an element and returns values for all instances found of this element in the XML document.
  ` Parameter: $1 - name of element to find , $2 - pointer to array which will hold values , $3 - path to XML document.

C_TEXT($1;$fElem;$pathfile;$3)
C_POINTER($2)
ARRAY TEXT(ElemVals;0)
$pathfile:=$3 ` path to the XML document to be opened.
  `"C:\\XMLStuff\\testpeopledom2.xml"

DocRef:=Open document($pathfile;"xml";Read Mode ) ` open XML document as read only.
$fElem:=$1 `element to find
```

```
If (OK=1)
  Repeat
    MyEvent:=SAX Get XML node(DocRef)
    If (MyEvent=XML Start Element) ` check if the type of event is an XML Start Element
      `get the name of the element
      SAX GET XML ELEMENT(DocRef;name;prefix;attrNames;attrValues)
      If (name=$fElem) `check if the element is the one you want
        MyEvent:=SAX Get XML node(DocRef) ` get the next node
        If (MyEvent=XML DATA) ` check if the event type is data
                `Get the data of the element
              SAX GET XML ELEMENT VALUE(DocRef;Value)
                `Append value to array
              APPEND TO ARRAY(ElemVals;Value)
        End if
      End if
    End if
  Until (MyEvent=XML End Document) ` repeat until the end of XML is reached
End if
CLOSE DOCUMENT(DocRef)
 `Copy array with values to the passed array
COPY ARRAY(ElemVals;$2->)
```

As you can see from the code above, the document is first opened in read only mode. Then, you keep on calling the SAX GET XML node command until the end of document is reached. The said command returns an event each time. The event returned is checked and if the event is an XML Start element, the SAX GET XML ELEMENT is used to get the element information. If the element is the one you are looking, the SAX Get XML node command is called again to move in the document and then retrieve the data by using SAX GET XML ELEMENT VALUE. This value is then stored in an array. As you can see the traversal is done using one document reference and is carried out in linear approach.

Conclusion

DOM (Document Object Model) and SAX (Simple API XML) are different modes in 4th Dimension for parsing XML documents. These two modes offer their own advantages and disadvantages. DOM reads the whole XML document and builds an XML tree in memory. This mode allows flexibility in XML traversal as you can traverse at any directions but can be limited with the available memory when dealing with very large documents. Modifications of elements and their values are possible in this mode when parsing existing XML documents. Meanwhile, SAX does not generate a representation of XML content in memory. This mode works in a streaming approach where events are passed when traversing the document from the beginning to the end. Though it is not limited with the amount of memory you have, it is not as flexible as DOM as far as reading, analyzing, and manipulating XML structure.