XML Parsing, SAX/DOM

Chengkai Li
Department of Computer Science and Engineering
University of Texas at Arlington
Arlington, TX 76019
cli@uta.edu

SYNONYMS

XML Programming Interface

DEFINITION

XML parsing is the process of reading an XML document and providing an interface to the user application for accessing the document. An XML parser is a software apparatus that accomplishes such tasks. In addition, most XML parsers check the *well-formedness* of the XML document and many can also validate the document with respect to a DTD (Document Type Definition) or XML schema. Through the parsing interface, the user application can focus on the application logic itself, without dwelling on the tedious details of XML.

There are mainly two categories of XML programming interfaces, DOM (Document Object Model) and SAX (Simple API for XML). DOM is a tree-based interface that models an XML document as a tree of nodes, upon which the application can search for nodes, read their information, and update the contents of the nodes. SAX is an event-driven interface. The application registers with the parser various event handlers. As the parser reads an XML document, it generates events for the encountered nodes and triggers the corresponding event handlers. Recently there have been newly proposed XML programming interfaces such as *pull-based parsing*, *e.g.*, StAX (Streaming API for XML), and *data binding*, *e.g.*, JAXB (Java Architecture for XML Binding).

HISTORICAL BACKGROUND

DOM (Document Object Model) was initially used for modeling HTML (HyperText Markup Language) by various Web browsers. As inconsistencies existed among the individual DOM models adopted by different browsers, interoperability problems arose in developing browser-neutral HTML codes. W3C (World Wide Web Consortium) standardized and released DOM Level 1 specification on October 1, 1998, with support for both HTML and XML. DOM Level 2 was released in November 2000 and added namespace support. The latest specification DOM Level 3 was released in April 2004.

SAX (Simple API for XML) was developed in late 1997 through the collaboration of several implementers of early XML parsers and many other members of the XML-DEV mailing list. The goal was to create a parser-independent interface so that XML applications can be developed without being tied to the proprietary API (Application Programming Interface) of any specific parser. SAX1 was finalized and released on May 11, 1998. The latest release SAX2, finalized in May 2000, includes namespace support.

SCIENTIFIC FUNDAMENTALS

XML parsing is the process of reading an XML document and providing an interface to the user application for accessing the document. An XML parser is a software apparatus that accomplishes such tasks. In addition, most XML parsers check the *well-formedness* of the XML document and many can also validate the document with respect to a DTD (Document Type Definition) [1] or XSD (XML Schema (W3C)) [2]. Through the parsing interface, the user application can focus on the application logic itself, without dwelling on the tedious details of XML, such as Unicode support, namespaces, character references, well-formedness, and so on.

XML Programming Interfaces

Most XML parsers can be classified into two broad categories, based on the types of API that they provide to the user applications for processing XML documents.

Document Object Model (DOM): DOM is a tree-based interface that models an XML document as a tree of various nodes such as elements, attributes, texts, comments, entities, and so on. A DOM parser maps an XML document into such a tree rooted at a `Document` node, upon which the application can search for nodes, read their information, and update the contents of the nodes.

Simple API for XML (SAX): SAX is an event-driven interface. The application receives document information from the parser through a `ContentHandler` object. It implements various event handlers in the interface methods in `ContentHandler`, and registers the `ContentHandler` object with the SAX parser. The parser reads an XML document from the beginning to the end. When it encounters a node in the document, it generates an event that triggers the corresponding event handler for that node. The handler thus applies the application logic to process the node specifically.

The SAX and DOM interfaces are quite different and have their respective advantages and disadvantages. In general, DOM is convenient for random access to arbitrary places in an XML document, can not only read but also modify the document, although it may take significant amount of memory space. To the contrary, SAX is appropriate for accessing local information, is much more memory efficient, but can only read XML.

- DOM is not memory efficient since it has to read the whole document and keep the entire document tree in memory. The DOM tree can easily take as much as ten times the size of the original XML document [3]. Therefore it is impossible to use DOM to process very large XML documents such as the ones that are bigger than the memory. In contrast, SAX is memory efficient since the application can discard the useless portions of the document and only keep the small portion that is of interests to the application. A SAX parser can achieve constant memory usage thus easily handle very large documents.

- SAX is appropriate for streaming applications since the application can start processing from the beginning, while with DOM interface the application has to wait till the entire document tree is built before it can do anything.

- DOM is convenient for complex and random accesses that require global information of the XML document, whereas SAX is more suited for processing local information coming from nodes that are close to each other. The document tree provided by DOM contains the entire information of the document, therefore it allows the application to perform operations involving any part of the document. In comparison, SAX provides the document information to the application as a series of events. Therefore it is difficult for the application to handle global operations across the document. For such complex operations, the application would have to build its own data structure to store the document information. The data structure may become as complex as the DOM tree.

- Since DOM maintains information of the entire document, its API allows the application to modify the document or create a new document, while SAX can only read a document.

DOM and SAX are the two standard APIs for processing XML documents. Most major XML parsers support them. There are also alternative tree-based XML models that were designed to improve upon DOM, including JDOM and DOM4J. In addition to DOM and SAX, other types of APIs for processing XML documents have emerged recently and are supported by various parsers. Two examples are *pull-based parsing* and *Java data binding*.

Pull-Based Parsing: Evolving from XMLPULL, StAX (Streaming API for XML) also works in a streaming fashion, similar to SAX. Different from SAX where the parser pushes document information to the application, StAX enables the application to pull information from the parser. This API is more natural and convenient to the programmer since the application takes full control in processing the XML document.

Java Data Binding: A data-binding API provides the mapping between an XML document and Java classes. It can construct Java objects from the document (*marshalling*) or build a document from the objects (*unmarshalling*). Accessing and manipulating XML documents thus become natural and intuitive, since such operations are performed through the methods of the objects. The application can thus focus on the semantics of the data themselves instead of the details of XML. JAXB (Java Architecture for XML Binding) is a Java specification based on this idea.

Validating Parsers

In addition to accessing XML documents, another critical functionality of XML parsers is to validate the correctness of the documents. Given that XML is a popular data model for data representation and exchange over the Internet, the correctness of XML documents is important for applications to work properly. It is difficult to let the application itself handle incorrect documents that it does not expect. Fortunately, most major XML parsers have the ability to validate the correctness of XML documents.

The correctness of an XML document can be defined at several levels. At the bottom, the document should follow the syntax rules of XML. Such a document is called a *well-formed* document. For example, every non-empty element in a well-formed document should have a pair of starting tag and ending tag. Furthermore, the document should conform to certain semantic rules defined for the application domain, such as a "`state`" node containing one and only one "`capital`" node. Such semantic rules can be defined by XML schema specifications such as DTD or XSD (XML Schema (W3C)). An XML document that complies with a schema is called a *valid* document. Finally, the application may enforce its own specific semantic rules.

In principle all the parsers are required to perform mandated checks of well-formedness, although there are parser implementations that do not. A parser that checks for the validity of XML documents with respect to XML schema in addition to their well-formedness is a *validating parser*. Schema validation is support by both DOM and SAX API. Most major XML parsers are validating parsers, although some may turn off schema validation by default.

XML Parsing Performance

There is relatively few study in the literature on performance of XML parsing. However, as the first step in every application that takes XML documents to process, parsing can easily become the bottleneck of the application performance.

DOM is memory intensive since it has to hold the entire document tree in memory, making it incapable in handling very large documents. Therefore efforts have been made to improve DOM parser performance by exploiting *lazy* XML parsing [4]. The key idea is to avoid loading unnecessary portion of the XML document into the DOM tree. It consists of two stages. The pre-parsing stage builds a virtual DOM tree and the progressive parsing stage expands the virtual tree with concrete contents when they are needed by the application. Farfán *et al.* [5] further extends the idea to reduce the cost of pre-parsing stage by partitioning an XML document, thus only reading a partition into the DOM tree when it is needed.

Nicola *et al.* [6] investigated several real-world XML applications where the performance of SAX parsers is a key obstacle to the success of the projects. They further verified that schema validation can add significant overhead, which sometimes can be even several times more expensive than parsing itself.

Validation often incurs significant processing cost. One reason for such low efficiency is the division of parsing and validation steps. In conventional parsers these two steps are separate because validation often requires the entire document to be in memory thus has to wait till the parsing is finished. Therefore even for a SAX parser, the advantage of memory efficiency is lost. To cope with this challenge, there have been studies on integrating parsing and validation into a *schema-specific parser* [7, 8, 9]. For example, [7] constructs a push-down automaton to combine parsing and validation. Van Engelen *et al.* [10] uses deterministic finite state automata (DFA) to integrate them and the DFA is built upon the schema according to mapping rules. Kostoulas *et al.* [11] further applies compilation techniques to optimize such integrated parsers.

Takase *et al.* [12] explores a different way to improve parser performance. It memorizes parsed XML documents as byte sequences and reuses previous parsing results when the byte sequence of a new XML document partially matches the memorized sequences.

KEY APPLICATIONS*

Every XML application has to parse an XML document before it can access the information in the document and perform further processing. Therefore XML parsing is a critical component in XML applications.

URL TO CODE*

List of XML parsing interfaces:
DOM, http://www.w3.org/DOM/
JDOM, http://jdom.org/
DOM4J, http://dom4j.org/
SAX, http://www.saxproject.org/
StAX, http://jcp.org/en/jsr/detail?id=173
XMLPULL, http://www.xmlpull.org/
JAXB, http://www.jcp.org/en/jsr/detail?id=222

List of XML parsers:
Ælfred, http://saxon.sourceforge.net/aelfred.html
Crimson, http://xml.apache.org/crimson/
Expat, http://expat.sourceforge.net/
JAXP, https://jaxp.dev.java.net/
Libxml2, http://xmlsoft.org/index.html
MSXML, http://msdn.microsoft.com/en-us/library/ms763742.aspx
StAX Reference Implementation (RI), http://stax.codehaus.org/
Sun's Stax implementation, https://sjsxp.dev.java.net/
XDOM, http://www.philo.de/xml/
Xerces, http://xerces.apache.org/

CROSS REFERENCE*

XML, XML document, XML schema, XML element, XML attributes, XML Programming

RECOMMENDED READING

[1] Document Type Declaration, http://www.w3.org/TR/REC-xml/#dt-doctype
[2] XML Schema (W3C), http://www.w3.org/XML/Schema
[3] Harold, Elliotte Rusty. (2002): Processing XML with Java(TM): A Guide to SAX, DOM, JDOM, JAXP, and TrAX. Addison-Wesley 2002.
[4] Noga, M., Schott, S., Löwe, W. (2002): Lazy XML Processing. In ACM DocEng, ACM Press, New York, 2002.
[5] Farfán F., Hristidis V., and Rangaswami R. (2007): Beyond Lazy XML Parsing. DEXA 2007: 75-86.
[6] Nicola M. and John J. (2003): XML parsing: a threat to database performance. CIKM 2003: 175-178.
[7] Chiu, K., Govindaraju M., and Bramley, R. (2002): Investigating the limits of SOAP performance for scientific computing. In Proceedings of 11th IEEE International Symposium on High Performance Distributed Computing, pages 246-254, Edinburgh, Scotland, July 23-26, 2002.
[8] Thompson, H. and Tobin, R. (2003): Using finite state automata to implement W3C XML schema content model validation and restriction checking. In Proceedings of XML Europe, 2003.
[9] Zhang, W. and Van Engelen, R. (2006): A Table-Driven Streaming XML Parsing Methodology for High-Performance Web Services. 2006 IEEE International Conference on Web Services (ICWS): 197-204.
[10] Van Engelen, R. (2004): Constructing finite state automata for high performance XML web services. In Proceedings of the International Symposium on Web Services (ISWS), 2004.

[11] Kostoulas M., Matsa M., Mendelsohn N., Perkins E., Heifets A., and Mercaldi M. (2006): XML screamer: an integrated approach to high performance XML parsing, validation and deserialization. WWW 2006: 93-102.

[12] Takase T., Miyashita H., Suzumura T., and Tatsubori M. (2005): An adaptive, fast, and safe XML parser based on byte sequences memorization. WWW 2005: 692-701.