Journal of Computational Information Systems 7: 16 (2011) 5883-5890
Available at http://www.Jofcis.com

Using JSON for Data Exchanging in Web Service Applications

Dunlu PENG[†], Lidong CAO, Wenjie XU

School of Optical-Electrical and Computer Engineering, University of Shanghai for Science and Technology, Shanghai 200093, China

Abstract

XML is the de facto standard format for data exchanging in the context of web service applications. While as has been semi-structured, XML data has to be parsed before being processed at both the client and the server sides. Applications show that the parsing process of XML is quilt time and memory consuming, such as DOM needs to load the whole XML document into memory before processing the data. Compared with XML, JSON is a light-weight key-value style data exchanging format. As we know, the efficiency of mapping data between different data models is the key point to improve the performance of web service applications. In this work, a data model mapping approach, named as Dynamic Advanced Binding (DAB), is proposed for processing JSON data. This approach efficiently maps the JSON data onto the host language (such as Java) by dynamically building the Data Mapping Template (DMT) during the runtime. The DMTs are specified by extending context-free grammar and implemented with a pushdown automaton with output. The experiments are conducted to verify the performance of the proposed approach.

Keywords: JSON; Web Services; Data Model Mapping; Service-oriented Architecture; XML

1. Introduction

As the most common technique applied to realize Service Oriented Architecture (SOA), web service has gained lots of attention in nowadays. XML, which is a platform independent language for representing data and can realize the interoperation between heterogeneous platforms, has become the de facto standard for data exchanging in the context of web service applications. However, an XML parser is needed to read the XML data for web service at the client side. This brings two critical issues: 1) for the security of the application, a page should read XML data from the same domain, which improves the cost of deployment and creation of DMTML components; 2) due to the low efficiency of reading and parsing XML data during the execution of services, the performance of web service has been decreased.

In the age of human-oriented Web 2.0, Asynchronous JavaScript and XML (AJAX) becomes one of the highlights for developing web services. AJAX applies JavaScript at the client side, which can vastly decrease the workload of the server. To address the aforementioned issues of XML-based services, in this work, we investigate the performance of using JSON to take place of XML to represent the data exchanged in the application of web services.

JSON is a key-value style lightweight data exchanging format. For its simplicity, JSON makes it easy for human to read and write, and for computers to generate and parse. Though JSON has some characteristics similar to the C family of programing languages, such as C, C++, Java, JavaScript and Perl, it is indepen-

[†] Corresponding author.
Email addresses: dunlu.peng@gmail.com (Dunlu PENG).

dent of any programming language. JSON is a native data form for JavaScript, which means no special APIs or jars are needed to process JSON data. These features make JSON a promising format for data exchanging in web service applications.

In this paper, we address how to process JSON-based data efficiently in web services and put our focus on designing efficient approaches for the serialization and deserialization of JSON data. The paper is organized as follows. In Section 2 we present some related work about JSON data and its application in web services. Section 3 analyzes the performance of JSON-based web services and the key factors having effect on it. We introduce two common data binding techniques and propose the dynamic advanced binding for JSON data processing in the same section. Section 4 presents the concept model and implementation model based on data mapping template for JSON-based data. We verify the performance of the proposed approach with experiments in Section 5. Finally, we draw the conclusions in Section 6.

2. Related Work

In recent years, a lot of research work has been done for processing JSON data. Most of the work compared JSON-based data with XML-based data to investigate their impact on the performance of web services. In [1], two factors that made XML-based web service running with low efficiency were pointed out. The factors were (1) parsing and formatting XML data and (2) network communication protocol of web services. It was certified in [2] that optimizing the encoding and decoding SOAP algorithms could improve the performance of web services. A dynamic template driven approach was presented in [3] to process SOAP with high performance. The paradigm efficiently maps the data between XML and Java.

Compared with XML, JSON can be parsed efficiently. Researches have already shown that JSON has been able to take the place of XML as data exchanging format in web services. This was verified with simulating test on the performance of JSON and XML [4]. A JSON-based object serialization algorithm was proposed for presenting the navigator for any Java object and producing a collection of JSON expressions according to the navigator [5]. Li. etc. used Java reflection to realize mapping data between JSON and Java. However, the performance of web services will be decreased with exploiting the technique to process complicated application data [6].

3. Analysis

In this section, we analyze how to process JSON data at the server side in the context of web service applications, and try to find the key factors to the performance of processing JSON data. Then, we introduce some related techniques and definitions applied to JSON data.

3.1. Procedure of invoking a JSON-based Web Service

Nowadays, data exchanging in web service applications between client and server is through HTTP. HTTP itself might be a bottleneck of the web service performance. However, in our work, we do not address this issue. We focus on how to process JSON data efficiently.

Fig.1 shows the entire procedure of invoking a JSON-based request-response web service. As we can see, it consists of five steps. Step 1 is to parse JSON data, which means the server will parse the data as soon as it receives the request data from the clients. In Step 2, the parsed JSON data is deserialized into application data. Step 3 is service invocation. The server invokes specific web service and catches the results. The time it takes is relevant to the computational complexity of the web service. Step 4 is serializing the results returned from Java data into JSON data. Step 5 is the final step which writes the JSON data into an output stream of HTTP, and then sends it to the client side.

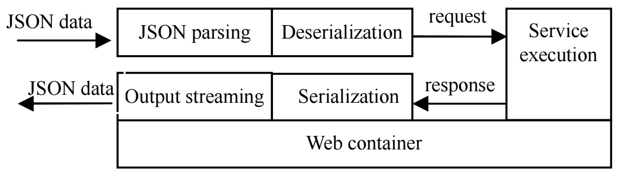


Fig. 1 Steps for Invoking JSON-based Web Service

3.2. Dynamic Advanced Binding

During the invocation of web services, mapping data between JSON and the application is the core operation to exchange data with JSON. Here we design a data template-based mapping model which is composed of three elements: JSON data type, Java data type and the mapping rules. The mapping procedure includes three phrases: parsing data, deserializing data and serializing data.

Before introducing our model, we should distinguish two groups of concepts: *deferred binding* and *advanced binding*, *dynamic binding* and *static binding.*

(1) *Deferred binding* and *advanced binding*

The difference between these two mechanisms lies in the moment when to get the binding information and execute the binding operations. In this phase, the parsed JSON data will be deserialized to application data. Binding here refers to the data model for mapping, and binding information refers to the binding rules used in the mapping. Advanced binding means the binding operation is executed after all the binding information has been acquired, while deferred binding means executing binding operation and acquiring the binding information are done simultaneously.

(2) *Dynamic binding* and *static binding*

Dynamic binding means the binding rules of JSON-Java mapping type are appended during the runtime, while for static binding the rules are added during the compiling time. Based on the aforementioned concepts, we classified the common techniques used in Java data mapping into two categories: *static advanced binding* and *dynamic deferred binding.*

In the *static advanced binding*, the template which records the binding information is generated with the code generator before binding execution. After being generation, the template is used to map data. In this way, the performance of the data mapping model has been improved because of avoiding the reflection of Java. However, the flexibility will be weakened because new template cannot be added in runtime.

The template is generated with the Java reflection operation during the binding execution in the *dynamic deferred binding*. In addition, the dynamic binding technique can map data in the runtime and add new data model mapping pair dynamically, which increases the flexibility of binding operation. However, facts show the performance will be dropped when the Java reflection is used too frequently.

In this work, we propose the dynamic advanced binding (DAB) for processing JSON dat. Fig. 2 illustrates its mechanism. The rules for mapping data between JSON and Java are generated in the compiling time. The Data Mapping Template (DMT) is produced with the dynamic code generator according to the JSON data type, Java data type and the mapping rules. Finally, the DMT is used to realize the mapping.

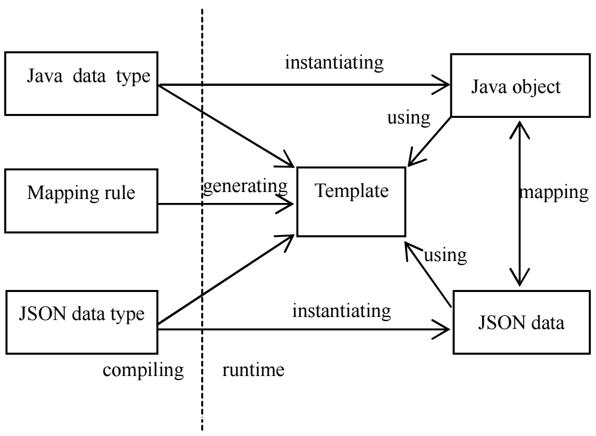


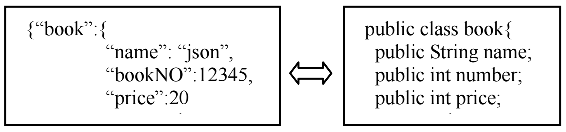
Fig. 2 Mechanism of Dynamic Advanced Binding

4. Data Mapping Model

In the previous section, we analyzed the features of DAB and discussed its importance for processing JSON data. This section presents the definitions, data model and implementation model of DMT.

4.1. Definitions for Data Mapping Template

We use the extended context-free grammars to describe JSON data model and Java data model, and generate the mapping rules with corresponding mapping scheme. Fig. 3 compares the data types of JSON with those of Java.



```
{"book":{
        "name": "json",
        "bookNO":12345,
        "price":20
```

```
public class book{
  public String name;
  public int number;
  public int price;
```

Fig. 3 Comparison of Data Types between JSON and XML

Definition 1. For any data type T in JSON and Java, its Data Mapping Template is described as a tuple DMT= (G^S, G^J), where G^S and G^J represent the context-free grammars of JSON data type and Java data type, respectively.

Definition 2. The ***context-free grammars*** for DMT is defined as $G=(V_T, V_N, S, \mathscr{P}, M)$: V_T is a non-null finite set of non-terminals, $\forall A \in V_T$, where A is a non-terminal. For G^S, V_T is the JSON data type set of T, including scalars, objects and arrays. For G^J, V_T is the Java data type set of T, containing simple types, arrays and JavaBeans; V_N is a non-null finite set of terminals, $\forall \alpha \in V_N$, where α is a terminal. For JSON, V_N is the set of names in JSON data. For Java, V_N is the name set of the fields in Java data; S is a start symbol and a non-terminal as well. $\mathscr{P}$ is a non-null finite set of production. The form of each $\mathscr{P}$ is $A \rightarrow \alpha$, $A \in V_T$, $|A| \leq |\alpha|$; M is a set of mapping scheme, $\forall m \in M$, where m is a scheme consisting a group of atomic operations which are corresponding to some productions and defining a series of actions in the production reduction.

Every production in G corresponds to one or more atomic operations. JSON data can be organized as a tree. A non-leaf node is a JSON object and a leaf is composed of a scalar pair of name and value. The value of an object is the sub-tree rooted at the object. The values can be parsed by a JSON processor. The atomic operations of JSON data model contain the creation of JSON objects and getting their values, adding sub-objects, and setting or getting values for key-value pair.The atomic operations of Java data model contain building and accessing objects, getting/setting values for the fields. The atomic operations of JSON and Java data model are listed in Table 1.

Table 1 Comparison of Atomic Operations between JSON and Java

JSON	Java
createObject(objectName)	*createJavaType(typeName)*
addChildObject(object)	*setSimpleValue( )*
getNextChildObject()	*setFieldValue(name, value)*
getNextPair()	*getFieldName( )*
setPairValue (name, value)	*getFieldValue( )*
returnObject()	*getIndexValue(index)*
	setIndexValue(index, value)
	returnType( )

Table 2 Grammar G^{S} and Mapping Scheme M^{S} of JSON Data Model

G^{S} and M^{S}	
$S \rightarrow$ name{T^{S}.obj=S.obj}T^{S}{S.value=T^{S}.value}	$T^{S} \rightarrow$ { T^{SS}.value=createJavatype()， T^{SS}.obj= T^{S}.obj} T^{SS}{ T^{S}.value= T^{SS}.value }
$T^{S} \rightarrow${T^{SC}.value=createJavaType( )， T^{SC}.obj=T^{S}.obj}T^{SC}{T^{S}.value= T^{SC}.value }	$T^{SS} \rightarrow$name{T^{SP}.obj =TSS. getNextPair()， T^{SS}.value=TSP.value }T^{SP} T^{SS}
$T^{SC} \rightarrow$name{T^{S} .obj= T^{SC}.getNextChildObject()}T^{S} {T^{S}.setFieldValue(T^{S}.value)} T^{SC}	$T^{SA} \rightarrow \varepsilon$
$T^{SC} \rightarrow \varepsilon$	$T^{SS} \rightarrow \varepsilon$
$T^{S} \rightarrow${ T^{SA} .value=createJavaType()， T^{SA}.obj= T^{S}.obj }T^{SA} { T^{S}.value= T^{SA}.value }	$T^{SP} \rightarrow \varepsilon$ {T^{SP} .setSimpleValue(T^{SP}.getPairValue())}

Table 2 shows the grammar G^{S} and mapping scheme M^{S} developed for JSON data model. In the grammar production P, S is a start symbol; T^{S} represents the JSON data model which is divided into three groups: scalar type SS, array type SA and complicated type SC. And SS consists of SP which stands for key-value pairs. G^{S} defines JSON data model T^{S} as tree structure. T^{S} has two attributes: T^{S}.obj and T^{S}.value, where T^{S}.obj represents JSON data while T^{S}.value refers the Java data mapped from JSON data.

As we mentioned, G^{S} and G^{J} are used to describe the definition of JSON and Java data model separately, and the mapping rules are represented with the mapping scheme M^{S} and M^{J} (Table 3). The braces in the above tables stand for the implementation of mapping rules. DMT defines the mapping scheme of a specific data type T. We take JSON data as an example and propose an algorithm to generate the DMTs.

Fig. 4 depicts the algorithm whose input is JSON data T^{S} and output is a DMT instance containing T^{SS}, T^{SC} and their corresponding scheme T^{SA}. It makes use of a depth-first traversal on the JSON data. For simplicity, in this work, we omit recursive analysis of JSON object and JSON array. We generate a DMT instance with algorithm and showed it in Table 4.

Table 3 Grammar G^J and the Mapping Scheme M^J of Java Data Model

G^J and M^J	
$S \rightarrow$ {*S.createObject* ( ) , T^J.*value=S.value*, T^J·*obj=S.obj*}T^J	$T^{JA} \rightarrow$ { T^J.*createObject* ( ) , T^J. *Value=*T^{JA}.*getIndexValue* ( ) } T^J { T^{JA}.*addChildObject* (T^J.*obj*) }T^{JA}
$T^J \rightarrow$ {T^J.*obj=*T^{JC}.*obj*, T^{JC}.*value=*T^J.*value*}T^{JC}	$T^{JA} \rightarrow \varepsilon$
$T^{JC} \rightarrow$ *field*{ T^J.*value=*T^{JC}.*getFiledValue* (*field*), T^J.*createObject* ( )}T^J {T^{JC}.*addChildObject* (T^J.*obj*) }T^{JC}	$T^J \rightarrow$ { T^J.*obj=*T^{JS}.*obj*, T^{JS}.*value=* T^J.*value*}T^{JS}
$T^{JC} \rightarrow \varepsilon$	$T^{JS} \rightarrow \varepsilon$ { T^{JS} .*setPairValue* (T^{JS}. *Value*) }
$T^J \rightarrow$ { T^J .*obj=* T^{JA}.*obj*, T^{JA}.*value=* T^J.*value* }T^{JA}	

```
Algorithm 1:    DeserializingJSON
Input: $T^S$-- JSON-style data
Output: ($T^{SS}$, $T^{SC}$, $T^{SA}$ )—DMT Java Data objects
1： Procedure:
2:     $S \rightarrow T^S$
3:   For each type t in S do
4:         If t is an array do
5:             $T^S \rightarrow T^{SA}$
6:             For each element in $T^{SA}$ do
7:             $T^{SA} \rightarrow T^S T^{SA}$
8:           End For
9:             $T^{SA} \rightarrow \varepsilon$
10:        ElseIf t is a simple type do
11:            $T^S \rightarrow T^{SS}$
11:            $T^{SS} \rightarrow T^{SP} T^{SS}$
12:            $T^{SP} \rightarrow \varepsilon$
13:        Else //complex data type
14:            $T^S \rightarrow T^{SC}$
15:            For each element in $T^{SA}$ do
16:              $T^{SC} \rightarrow T^S T^{SC}$
17:            End For
18:            $T^{SC} \rightarrow \varepsilon$
19:        End if
20:    End for
21:  Return $T^{SS}$, $T^{SC}$, $T^{SA}$
```

Fig. 4 Algorithm for Generating DMT from JSON

Table 4 JSON→Java DMT Instance

JSON→Java DMT instance	
$S \rightarrow$ *book*{T^S.*obj=S.obj*}T^S{*S.value=* T^S.*value*}	T_1^{SS}.*value=B.value*}*B* T_2^{SS}
$T^S \rightarrow$ {T_0^{SS}·*value=new Book()*， T_0^{SS}.*obj=* T^S.*obj*} T_0^{SS} {T^S.*value=* T_0^{SS}.*value*}	$B \rightarrow \varepsilon${*B.value=Integer.parseInt(N.getPairValue)*}
$T_0^{SS} \rightarrow$ *name*{*N.obj =* T_0^{SS}.*getNextPair()*，	$T_2^{SS} \rightarrow$ *bookNo*{*P.obj =* T_2^{SS} . *getNextPair()*，
T_0^{SS}.*value=N.value*}*N*T_1^{SS}*N*	T_2^{SS}.*value=P.value*}*P* T_3^{SS}
$N \rightarrow \varepsilon${*N.value=N.getPairValue*}	$P \rightarrow \varepsilon${*P.value=Integer.parseInt(P.getPairValue)*}
$T_1^{SS} \rightarrow$ *bookNo*{*B.obj =*T_1^{SS}.*getNextPair()*，	$T_3^{SS} \rightarrow \varepsilon$

4.2. Data Mapping Automaton

We just discussed the context-free grammar-based conception model for mapping JSON data to Java data, in this section we propose its implementation model. In the model, for any data type T, its implementation model is regarded as a pair of pushdown automaton which is used to recognize the data type defined by grammar G^S and G^J. The *Data Mapping Automaton (DMA)* is defined as follows.

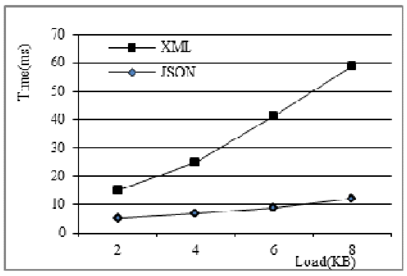
Definition 3. A ***Data Mapping Automaton*** is denoted as $DMA=(Q, \Sigma, \Gamma, \delta, q_0, Z_0, F, O)$. Where, Q is a finite set of state; $\sum$ is a finite alphabet, each element of which becomes a input characters; Γ is an alphabet for the stack; $q_0 \in Q$ is the initial state of M; $Z_0 \in \Gamma$ is a special stack character, called initial character of the stack , the only character in the stack when M initials. F is a set of signaled state; O is a set of output actions; δ is a function of state transitions. $\delta(q_0, a, Z_0)=\{(p, \gamma, O), (p, \gamma, O), \ldots, (p, \gamma, O)\}$, where M is at the state of q_0, the top element in the stack is Z_0, and a is an input character, for each i=1, 2, 3, …, m, the state of M is selectively converted to p, the top element Z_0 is popped out, and characters in γ are pushed down into the stack with the output execution at the same time. $DMA=(Q, \Sigma, \Gamma, \delta, q_0, Z_0, F, O)$can be produced by the context-free grammars $G=(V_T, V_N, S, \mathscr{P}, M)$.

The whole process of mapping data is as follows. When a JSON data object enters the DMA, the JSON Processor parses the data as a tree structure as the DMA runs. The virtual tree is input into the memory temporarily and then the state controller gets the JSON tree nodes. By doing so, the controller checks the chart and executes state transition operation according to the characters in the stack. Simultaneously, a java object is built and assigned. As the DMA ends, it returns a Java object which is mapped from the JSON object. When the input object is a Java data object, use Java Type Reader to substitute for the Jackson JSON Processor. Then, the controller executes the same operations as for JSON data object. Eventually it returns a JSON data object.

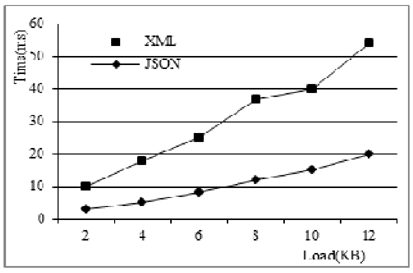
5. Experimental Evaluation

In this section, we employ the experiments to verify the performance of the DMT driven JSON processing approach. It was compare with that of the XML-based web services. We ran the experimental on a PC with 1.73 Intel Core Duo CPU, 2GB Memory installed with Windows XP Profession SP3. The JVM we used is JDK1.6.0 and Apache Tomcat 5.5 was exploited as the Web container. XML and JSON data was parsed with Simple 2.5.2 and Jackson1.7.6, respectively.

We firstly conducted experiments to investigate the scalability of serializing/deserializing java data objects to/from JSON and XML data objects. The time was measured for serializing/deserializing as the size of data objects changed from 2kb to 12kb.



(a) Comparison of Serialization



(b) Comparison of Deserilization

Fig. 5 Comparison of Scalability between JSON and XML

Fig.5 presents the result of the experiments. Fig. 5(a) describes the measurement of time needed for the serialization of Java data objects to JSON/XML. From the figure, we can see that, the time needed for serializing java data objects both increases to JSON and XML data objects. However, for JSON data, it increases much slower than that for XML data, which means the scalability for using JSON as the data format for exchanging has better performance than using XML. Fig. 5(b) illustrates the deserializing JSON/XML data objects to java data objects. Similar to serialization, the performance of deserializing JSON data objects to java data objects has surpassed that of deserializing XML data objects to java data objects.

To verify the parsing performance, we conduct the second group of experiments by comparing the time needed for parsing different sizes of data with Jackson parser for JSON and DOM/ SAX for XML data. We present the results in Fig.6. From the figure, we know that, parsing JSON data spent less time than parsing XML data with DOM or SAX at the same size. That is because JSON is a native language of javascript, it need not parse JSON data at the client side, while we shall use the specific APIs to parse XML data by constructing a DOM tree or generating events streams with SAX.

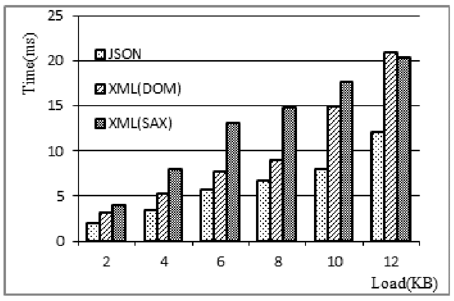


Fig.6 Comparison of Serilization

6. Conclusion

Data exchange is a very important issue for making service based computing more efficiently. In this work, we investigate how to employ JSON as the data exchange format for web service applications. We present an efficient approach to serialize and deserialize JSON data in web service applications. Compared with XML, using JSON-style data for exchanging can improve the performance of web service applications. Our approach is based on the data mapping template generated by dynamic advanced binding technique through which the JSON data can be processed efficiently. Experimental results show that JSON performs better than XML in being parsed, being serialized and being deserialized. In our future work, we will try to study how to integrate JSON data with databases.

Acknowledgement

This work is supported by Municipal Nature Science Foundation of Shanghai under Grant No.10ZR1421100 and Foundation of Graduate Core Course Construction of USST under Grand No. 16000206.

Reference

[1] D. Davis and M. Parashar. Latency performance of SOAP implementations. In *Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid- 2002),* pages 407–412, 2002.

[2] C. Kohlhoff and R. Steele. Evaluating SOAP for high perfomance business applications: Real-time trading systems. In *Proceedings of the 12th International World Wide Web (WWW2003),* pages 255-264, 2003.

[3] L. Hua, J.Wei, C. Niu and H. Zheng. High Performance SOAP Processing Based on Dynamic Template-Driven Mechanism. *Chinese Journal of Computer* 7(29), pages 1145-1156, 2006.

[4] C.Cui and H.Ni. Optimized simulation on XML with JSON. *Communication Technology* (*Chinese*) 42(8) pages 108-111. 2009 08, 212

[5] T. Zhang, Q. Huang, Y. Mao and X. Gao. Algorithm of object serialization based on JSON. *Computer Engineering and Applications(Chinese)* 43(15), pages 98-100, 2007.

[6] Z. Li Transition from Java to JSON data with Java reflection. A*cademic journal of Chengde Petroleum College* (Chinese) 3(12), pages 36-39 , 2010.