УДК 004.[822+823]

СЕМАНТИЧЕСКИЕ ФРЕЙМЫ: РАЗРАБОТКА ФОРМАТА СЕРИАЛИЗАЦИИ ДАННЫХ

А.И. ДУНЧЕНКО
(Представлено: Д.В. ПЯТКИН)

Представлено понятие о семантических фреймах. Сделан вывод о необходимости разработки формата сериализации для этой модели представления знаний, перечисляется перечень проблем, которые при этом требуют решения. Выполнен анализ наиболее очевидных способов достижения цели. Показано, как существующие наработки и стандарты могут быть адаптированы для решения поставленной задачи.

Семантические фреймы (СФ) – это универсальная система представления знаний об окружающем нас мире стредствами ЭВМ, которая позволяет интерпретировать и структурировать поступающую информацию в виде, приемлемом для понимания как человеком (как правило, экспертом в конкретной проблемной области), так и компьютерной системой, выполняющей обработку этой информации.

Семантические фреймы являются гибридной моделью, базирующейся на концепциях *семантических сетей* [1] и *фреймов* [2] (отсюда и соответствующее название) и дополняющей их элементами *логики первого порядка* [3] и *нечеткой логики* [4].

При проектировании любого программного обеспечения, независимо от его сложности и того, выполняет ли оно конвеерную обработку данных или глобально оперирует средой выполнения (имеет побочные эффекты), всегда возникает вопрос о необходимости введения фиксированных механизмов обмена данными – протоколов взаимодействия – между компонентами системы: функциями, объектами, модулями или даже отдельными приложениями [5]. Примерами такого взаимодействия являются: *FFI* [6], *CGI* [7], *FIFO* [8], *сокеты* [9], *каналы* [10], сообщения.

В связи с этим рассмотрению подлежат следующие проблемы:

1) *обеспечение аппаратной независимости*, которое столь же актуально и в нынешнее время, когда большинство настольных пользовательских систем работает на little-endian i386- и amd64-платформах, мобильные устройства – на dual-endian ARM-процессорах, а в сети Internet стандартным считается big-endian порядок байтов;

2) *обеспечение независимости от типа операционной и файловой систем*, так как даже при использовании виртуальных машин и интерпретируемых высокоуровневых языков программирования, попрежнему имеют место трудности в реализации, обусловленные кодировками, регистро-(не)зависимостью, механизмами работы с файлами, устройствами и т.п.;

3) *обеспечение независимости от применяемого языка программирования (ЯП) или версии компилятора* (наиболее ярко выражена эта проблема в open-source сообществе, где разработчики имеют негативную тенденцию к внесению изменений в ABI [11] или API [12] с выходом каждой новой мажорной версией своего ПО);

4) *in-house разработка или применение уже существующего стандарта* (последнее, как правило, является более предпочтительным решением).

Модель СФ представляет собой не только спецификацию, но и фактическую реализацию – библиотеку, написанную на ЯП Common Lisp, – которая предназначена для работы в составе более сложной интеллектуальной программной системы. Поэтому для нее также справедливы все перечисленные ранее проблемы, ведь конечное ПО в процессе своего функционирования предполагает:

1) сохранение и чтение файлов с жесткого диска;

2) обмен данными между компонентами посредством сокетов;

3) отображение информации пользователю.

Таким образом, необходимо разработать формат для сериализации данных, который позволит не только эффективно выполнять операции ввода/вывода, но и будет приемлемым для восприятия человеком (экспертом или программистом, выполняющим проверку/отладку системы). Следуя этим требованиям, которые, кстати, соответствуют Unix-философии, целесообразно применить текстовый формат. Причем саму проблему можно разделить на две подзадачи:

1) там, где предполагается оперирование данными только в рамках библиотеки – сохранение и чтение из файлов – использовать *символичекие выражения* [13], на которых, собственно, и базируется синтаксис Common Lisp;

2) в случае если необходимо передавать данные между программами посредством CGI или сокетов (например, между сервером базы знаний и веб-сервером), утилизировать один из существующих текстовых языков разметки [14], адаптировав его под конкретные нужды.

Среди всех существующих языков разметки и представления данных наиболее подходящими являются следующие кандидаты: JSON, YAML, XML.

Рассмотрим их более подробно.

JSON (JavaScript Object Notation [15]) является подмножеством синтаксиса языка JavaScript. Считается, что, несмотря на свое происхождение, JSON 1) не зависит от JavaScript и может использоваться с любым другим языком программирования и 2) за счет своей лаконичности по сравнению с XML, может быть более подходящим для сериализации сложных структур. Однако при детальном расмотрении оказывается, что оба эти утверждения далеки от истины.

Во-первых, система типов JSON полностью соответствует таковой именно языка JavaScript, поэтому, в сравнении с большинством других ЯП, зачастую либо оказывается крайне примитивной, либо не позволяет выполнить однозначную конверсию между значениями. Примеры: 1) в Common Lisp нет различий между константами *null* и *false*, оба значения представлены как *nil*, в Scheme вместо nil вообще используется пустой список; 2) символы и комплексные числа (C, Common Lisp, Scheme, Ruby, Python) приходится представлять в виде строки, в результате чего выполнить обратную интерпретацию в ряде случаев невозможно (что есть "4" – строка, символ или число?); 3) объекты JSON не являются полноценными hash-таблицами – в отличие от, к примеру, C++, Ruby или Python, ключами могут быть только строки.

Во-вторых, простой по сравнению с тем же XML синтаксис в то же время является и недостатком, так как провоцирует избыточные вложенные структуры там, где в XML можно обойтись лишь аттрибутами. Кроме того, JSON просто-напросто не умеет представлять рекурсивные структуры.

YAML (YAML Ain’t Markup Language [16]) ставит перед собой следующие задачи:

1) быть выразительным, расширяемым и доступным в понимании человеку;

2) поддерживать структуры данных, характерные для различых ЯП, обеспечивая легкую переносимость;

3) быть простым в реализации и использовании за счет обработки в один проход.

YAML имеет больше типов данных, чем JSON, поддерживает теги и ссылки на блоки. И тем не менее этот формат также не желателен в качестве решения для поставленной задачи. Создатели формата, как и автор языка Python, повторили ту же ошибку, сделав пробельные символы синтаксическим элементом. Это часто является причиной трудно обнаруживаемых ошибок и приводит к значительным трудностям как при визуальном промотре многостраничных текстов, так и при написании утилит, предназначенных для анализа (обработки) документов. Иначе говоря, YAML является приемлемым вариантом, если речь идет о работе с файлами конфигурации или небольшой базой данных, но для представления таких сложных структур, как фреймы, этот формат не подходит.

Что касается XML (eXtensible Markup Language [17]), то несмотря на предостаточное количество критики в адрес данного формата (например, что для обработки документов требуется больше оперативной памяти и вычислительных ресурсов, чем в случае с JSON/YAML), он является если не идеальным, то вполне обоснованным решением проблемы: гибкий синтаксис XML позволяет 1) адаптировать его под любые задачи, связанные с разметкой и 2) сериализовать сколь угодно сложные иерархические структуры данных. Таким образом, взамен потери производительности мы получаем необычайно мощное средство представления данных, уступающее по своим возможностям только S-выражениям.

Определившись с форматом можно приступить к этапу проектирования структуры документа, т. е. опираясь на иерархию классов, предназначенных для описания и работы с фреймами, определить перечень используемых тегов, их аттрибутов, а также правила формирования документа. Для этого можно воспользоваться формальной системой для определения синтаксиса EBNF (extended Backus-Naur form [18]):

document	=	prologue, pool node;
prologue	=	'<?xml version="1.0" encoding="UTF-8" ?>';
pool node	=	'<pool', [' name="', pool name, '"'], '>', {frame node}, '</pool>';
frame node	=	'<frame id="', frame id, '"', [' name="', frame name, '"'], '>', frame head node, frame body node, '</frame>';
frame head node	=	'<head>', [classifiers node], [clarifiers node], '</head>';
classifiers node	=	'<classifiers>', {classifier node}, '</classifiers>';
classifier node	=	'<c>', classifier, </c>;
clarifiers node	=	'<clarifiers>', {clarifier node}, '</clarifiers>';
clarifier node	=	' ', predicate, ;
frame body node	=	'<body>', {slot node | slot reference node}, '</body>';

slot node = '<slot id=" ', slot id, ' " ', [' name =" ', slot name, ' " '], '>', slot head node, slot data node, '</slot>';
slot head node = '<head>', [classifiers node], [clarifiers node], [membership node], '</head>';
membership node = '<membership>', membership, '</membership>';
membership = *real number between 0 and 1*;
slot data node = '<data type=" ', slot data type, ' ">', slot data, '</data>';
slot data type = 'boolean' | 'character' | 'number' | 'string' | 'JSON' | 'S-exp', 'regexp', 'base16' | 'base64' | 'slot-ref' | 'frame-ref' | custom data type;
custom data type = *string*;
slot reference node = '<slot-ref id=" ', slot id, ' " />'.
frame id = id;
slot id = id;
id = *string.*

Как видно из правил, если пул, фрейм или слот не имеет идентификатора, аттрибут *name* следует опускать. Для классификаторов и квалификаторов, которые с большой долей вероятности будут самыми часто встречающимися элементами, используются краткие теги **<*c*>** и **<*p*>** (сокращения от *classifier* и *predicate* соответственно). Сами предикаты вписываются в виде исходных *F-выражений* (F-expressions), а не в полученной после парсинга символической форме. Если слот принадлежит фрейму, то используется элемент **<*slot*>**, иначе (например, если фрейм ссылается на слот с классификатором *shared*) – **<*slot-ref*>**.

Теперь, когда общая структура документа четко определена, следует акцентировать внимание на промежуточном представлении СФ с помощью символических выражений, т. е. выбрать такой способ организации данных, который позволит с легкостью не только генерировать XML-разметку, но и выполнять обратную операцию – строить фреймы на основе данной формы.

Эта задача легко решается, если для работы с XML использовать пакет *cl-xmls*, который, кстати, и использует S-выражения в качестве входных/выходных данных. Построение каждого элемента починено весьма простому правилу, которое в нотации EBNF выглядит следующим образом:

node = '(', node name, node attributes, node children, ')';
node name = literal;
node attributes = '(', { attribute }, ')';
attribute = '(', attribute name, attribute value, ')';
attribute name = literal;
attribute value = value;
node children = value | { node };
literal = *string* | *symbol*;
value = *any Lisp object but pair*.

Такой незаурядный способ организации данных позволяет эффективно выполнять их обработку, применяя функциональный подход к программированию и встроенные в стандартную библиотеку языка Common Lisp макросы и функции. Более того, учитывая тот факт, что *cl-xmls* автоматически конвертирует любые значения в строки, можно существенно сэкономить память, если в качестве имен тегов использовать символы из пакета *keywords*.

Подводя итоги, сделаем вывод, что поставленная задача по разработке формата для сериализации СФ успешно решена: для внутренних нужд библиотеки, реализующей спецификацию СФ, будет использоваться промежуточное представление, основанное на символических выражениях, а для межпроцессного взаимодействия, где требуется обеспечить максимальную изолированность компонентов и независимость от программных решений, – язык разметки XML.

Единственной нерешенной проблемой остается сериализация триггеров. Причины, по которым это на данный момент представляется невозможным, следующие:

1) при сохранении замыканий нужно также сохранять информацию о среде выполнения, а в случае с Common Lisp, контекст может быть как статическим, так и динамическим;

2) триггеры могут быть написаны на разных ЯП, не обязательно Common Lisp;

3) стек может и вовсе содержать цепочку вызовов из разных ЯП, например:

C → Common Lisp → C→ Tcl → C → Common Lisp → C,

в результате чего выполнить полный захват и последующую корректную инициазицию невозможно в принципе.

ЛИТЕРАТУРА

1. Semantic network [Electronic resource] // Wikipedia – The Free Encyclopedia. – Mode of access: https://en.wikipedia.org/wiki/Semantic_network. – Date of access: 20.09.2017.
2. Frame (artificial intelligence) [Electronic resource] // Wikipedia – The Free Encyclopedia. – Mode of access: https://en.wikipedia.org/wiki/Frame_(artificial_intelligence). – Date of access: 20.09.2017.
3. First-order logic [Electronic resource] // Wikipedia – The Free Encyclopedia. – Mode of access: https://en.wikipedia.org/wiki/First-order_logic. – Date of access: 20.09.2017.
4. Fuzzy logic [Electronic resource] // Wikipedia – The Free Encyclopedia. – Mode of access: https://en.wikipedia.org/wiki/Fuzzy_logic. – Date of access: 20.09.2017.
5. Inter-process communication [Electronic resource] // Wikipedia – The Free Encyclopedia. – Mode of access: https://en.wikipedia.org/wiki/Inter-process_communication. – Date of access: 20.09.2017.
6. Foreign function interface [Electronic resource] // Wikipedia – The Free Encyclopedia. – Mode of access: https://en.wikipedia.org/wiki/Foreign_function_interface. – Date of access: 20.09.2017.
7. Common Gateway Interface [Electronic resource] // Wikipedia – The Free Encyclopedia. – Mode of access: https://en.wikipedia.org/wiki/Common_Gateway_Interface. – Date of access: 20.09.2017.
8. FIFO (computing and electronics) [Electronic resource] // Wikipedia – The Free Encyclopedia. – Mode of access: https://en.wikipedia.org/wiki/FIFO_(computing_and_electronics). – Date of access: 20.09.2017.
9. Unix domain socket [Electronic resource] // Wikipedia – The Free Encyclopedia. – Mode of access: https://en.wikipedia.org/wiki/Unix_domain_socket. – Date of access: 20.09.2017.
10. Pipeline (computing) [Electronic resource] // Wikipedia – The Free Encyclopedia. – Mode of access: https://en.wikipedia.org/wiki/Pipeline_(computing). – Date of access: 20.09.2017.
11. Application binary interface [Electronic resource] // Wikipedia – The Free Encyclopedia. – Mode of access: https://en.wikipedia.org/wiki/Application_binary_interface. – Date of access: 20.09.2017.
12. Application programming interface [Electronic resource] // Wikipedia – The Free Encyclopedia. – Mode of access: https://en.wikipedia.org/wiki/Application_programming_interface. – Date of access: 20.09.2017.
13. S-expression [Electronic resource] // Wikipedia – The Free Encyclopedia. – Mode of access: https://en.wikipedia.org/wiki/S-expression. – Date of access: 21.09.2017.
14. Markup language [Electronic resource] // Wikipedia – The Free Encyclopedia. – Mode of access: https://en.wikipedia.org/wiki/Markup_language. – Date of access: 21.09.2017.
15. JSON [Electronic resource] // Wikipedia – The Free Encyclopedia. – Mode of access: https: //en.wikipedia.org/wiki/JSON. – Date of access: 21.09.2017.
16. YAML [Electronic resource] // Wikipedia – The Free Encyclopedia. – Mode of access: https: //en.wikipedia.org/wiki/YAML. – Date of access: 21.09.2017.
17. XML [Electronic resource] // Wikipedia – The Free Encyclopedia. – Mode of access: https: //en.wikipedia.org/wiki/XML. – Date of access: 21.09.2017.
18. Extended Backus–Naur form [Electronic resource] // Wikipedia – The Free Encyclopedia. – Mode of access: https://en.wikipedia.org/wiki/Extended_Backus%E2%80%93Naur_form. – Date of access: 21.09.2017.