Acceso a datos. Apéndice JSON

Por Francisco García

Es frecuente que los programas necesiten almacenar los datos que manipulan, de modo que estos puedan ser consultados posteriormente. Una de las formas actuales más usadas sin duda es JSON (JavaScript Object Notation). Aunque en su nombre aparezca JavaScript es debido a que es muy común en el intercambio de ficheros web, pero evidentemente Java tiene sus librerías para trabajar JSON.

1.1. Formato de un fichero JSON

JSON puede representar dos tipos estructurados: *objetos* y *matrices*. Un objeto es una colección no ordenada de cero o más pares de nombres/valores. Una matriz es una secuencia ordenada de cero o más valores. Los valores pueden ser cadenas, números, booleanos, nulos y estos dos tipos estructurados.

El siguiente ejemplo muestra la representación JSON de un objeto que describe a un profesor. El objeto tiene valores de cadena para nombre y apellido, un valor numérico para la edad, un valor de objeto que representa el domicilio de la persona y un valor de matriz de objetos de números telefónicos.

```
{
     "nombre": "Fran",
     "apellido": "García",
     "edad": 25,
     "domicilio": {
        "direccion": "Lillo Juan, 128",
        "ciudad": "San Vicente del Raspeig",
        "comunidad": "Valenciana",
        "codigoPostal": 03690
     },
     "numerosTelefonos": [
        {
            "tipo": "casa",
            "numero": "666 666 666"
        },
        {
            "tipo": "movil",
            "numero": "777 777 777"
```

```
        }
    ]
}
```

1.2. API de modelos

La API de modelos de objetos crea una estructura de árbol, de acceso aleatorio, que representa los datos JSON almacenados en la memoria. Es posible recorrer el árbol y formular consultas. Este modelo de programación es el más flexible y posibilita el procesamiento en casos en que se requiera acceso aleatorio a la totalidad del contenido de la memoria. Sin embargo, a menudo no es tan eficiente como el modelo de streaming y requiere más memoria.

La API de modelos de objetos es similar a la API de modelos de objetos de documento (DOM) para XML. Es una API de alto nivel que proporciona modelos de objetos inmutables para estructuras de objetos y matrices JSON. Estas estructuras JSON se representan como modelos de objetos usando los tipos de
Java JsonObject y JsonArray. En la tabla siguiente se incluyen las clases e interfaces principales de la API de modelos de objetos.

Clase o interfaz	Descripción
Json	Contiene métodos estáticos para crear lectores, escritores, constructores de JSON y sus objetos de fábrica.
JsonGenerator	Escribe datos JSON en forma de stream, con un valor por vez.
JsonReader	Lee datos JSON de un stream y crea un modelo de objeto en la memoria.
JsonObjectBuilder JsonArrayBuilder	Crean un modelo de objeto o un modelo de matriz en la memoria agregando valores del código de aplicación.
JsonWriter	Escribe un modelo de objeto de la memoria en un stream.
JsonValue JsonObject JsonArray JsonString JsonNumber	Representan tipos de datos para valores en datos JSON.

JsonObject suministra una vista Map para obtener acceso a la colección no ordenada de cero o más pares de nombres/valores del modelo. De modo similar, JsonArray ofrece una vista List para obtener acceso a la secuencia ordenada de cero o más valores del modelo. JsonObject, JsonArray, JsonString y JsonNumber son subtipos de JsonValue. Hay constantes definidas en la API para valores JSON nulos, verdaderos y falsos.

La API de modelos de objetos usa patrones generadores para crear estos modelos de objetos desde cero. El código de aplicación puede usar la interfaz JsonObjectBuilder para crear modelos que representen objetos JSON. El modelo que se obtiene es del tipo JsonObject. El código de aplicación puede usar la interfaz JsonArrayBuilder para crear modelos que representen matrices JSON. El modelo que se obtiene es del tipo JsonArray.

Estos modelos de objetos también pueden crearse a partir de un origen de entrada (por ejemplo InputStream o Reader) usando la interfaz JsonReader. De modo similar, los modelos de objetos pueden escribirse en un origen de salida (por ejemplo OutputStream o Writer) usando la clase JsonWriter.

En el siguiente ejemplo trabajaremos con el archivo “profesor.json” y veremos diferentes formas de acceder a las partes del fichero. Para ello necesitaremos incorporar al proyecto la librería json-simple y los imports mostrados:

```
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.util.Iterator;
import java.util.Map;

import org.json.simple.JSONArray;
import org.json.simple.JSONObject;
import org.json.simple.parser.*;

.....

public static void apiModelos() {

		Object obj;
		try {
			// parseado el fichero "profesor.json"
			obj = new JSONParser().parse(new FileReader("profesor.json"));

			// casteando obj a JSONObject
			JSONObject jo = (JSONObject) obj;

			// cogiendo el nombre y el apellido
			String nombre = (String) jo.get("nombre");
			String apellido = (String) jo.get("apellido");

			System.out.println(nombre);
			System.out.println(apellido);

			// cogiendo la edad como long
			long edad = (long) jo.get("edad");
			System.out.println(edad);

			// cogiendo direccion
			Map domicilio = ((Map) jo.get("domicilio"));

			// iterando direccion Map
			Iterator<Map.Entry> itr1 = domicilio.entrySet().iterator();
			while (itr1.hasNext()) {
				Map.Entry pareja = itr1.next();
				System.out.println(pareja.getKey() + " : " + pareja.getValue());
			}
```

```
                        // cogiendo números de teléfonos
                        JSONArray ja = (JSONArray) jo.get("numerosTelefonos");

                        // iterando números de teléfonos
                        Iterator itr2 = ja.iterator();

                        while (itr2.hasNext()) {
                                itr1 = ((Map) itr2.next()).entrySet().iterator();
                                while (itr1.hasNext()) {
                                        Map.Entry pareja = itr1.next();
                                        System.out.println(pareja.getKey() + " : " + pareja.getValue());
                                }
                        }
                } catch (FileNotFoundException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                } catch (IOException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                } catch (ParseException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                }
        }
```

También podemos usar la API de modelos de objetos para obtener datos de páginas web. En el siguiente ejemplo podemos ver cómo obtener en Facebook los nombres y sus posts públicos relacionados con el término *java*. En el siguiente código, las líneas 1 a 3 crean JsonReader; la línea 5 crea JsonObject para los resultados; la línea 7 itera respecto de cada resultado; y las líneas 8 a 11 obtienen el nombre de la persona que publicó el post y el post en sí, y los imprimen. Nótese que JsonReader y otros objetos de esta API pueden usarse en la instrucción try-with-resources.

```
URL url = new  URL("https://graph.facebook.com/search?q=java&type=post");

   try (InputStream is = url.openStream();

        JsonReader rdr = Json.createReader(is)) {

        JsonObject obj = rdr.readObject();

        JsonArray results = obj.getJsonArray("data");

        for (JsonObject result : results.getValuesAs(JsonObject.class)) {

            System.out.print(result.getJsonObject("from").getString("name"));

            System.out.print(": ");

            System.out.println(result.getString("message", ""));

            System.out.println("-----------");

        }

   }
```

1.3. API de streaming

La API de streaming ofrece un modo de analizar y generar JSON en streams. Le otorga al programador el control sobre el análisis y la generación. La API de streaming ofrece un analizador basado en eventos y brinda al desarrollador de aplicaciones la posibilidad de "pedir" el evento siguiente en lugar de tener que ocuparse del evento en una devolución de llamada. De este modo, el desarrollador cuenta con mayor control procedimental del procesamiento JSON. El código de aplicación puede procesar o descartar el evento del analizador y pedir el siguiente evento (extraer el evento). El modelo de streaming es adecuado para el procesamiento local cuando no se requiere acceso aleatorio a otras porciones de la información. De manera similar, la API de streaming permite generar JSON bien formado en stream escribiendo un evento por vez.
La API de streaming es similar a la API de streaming para XML (StAX) y está constituida por las interfaces JsonParser y JsonGenerator. JsonParser contiene métodos para analizar datos JSON usando el modelo de streaming. JsonGenerator contiene métodos para escribir datos JSON en un origen de salida. En la siguiente tabla se incluyen las clases e interfaces principales de la API de streaming.

Clase o interfaz	Descripción
Json	Contiene métodos estáticos para crear analizadores y generadores JSON, y sus objetos de fábrica.
JsonParser	Representa un analizador basado en eventos que puede leer datos JSON en un stream.
JsonGenerator	Escribe datos JSON en forma de stream, con un valor por vez.

En el siguiente ejemplo muestro como trabajar con todas las opciones de esta API utilizando el fichero “profesor.json”. Para ello necesitaremos incorporar a nuestro proyecto el archivo javax.json y las librerías indicacas:

```
import javax.json.Json;
import javax.json.stream.JsonParser;

.....

        public static void apiStreaming() {
                JsonParser parser;
                try {
                        parser = Json.createParser(new FileReader("profesor.json"));

                        while (parser.hasNext()) {
                                JsonParser.Event event = parser.next();
                                switch (event) {
                                case START_ARRAY:
                                case END_ARRAY:
                                case START_OBJECT:
                                case END_OBJECT:
                                case VALUE_FALSE:
                                case VALUE_NULL:
                                case VALUE_TRUE:
                                        System.out.println(event.toString());
```

```
                                        break;
                                case KEY_NAME:
                                        System.out.print(event.toString() + " " + parser.getString() + " - ");
                                        break;
                                case VALUE_STRING:
                                        System.out.println(event.toString() + " " + parser.getString());
                                        break;
                                case VALUE_NUMBER:
                                        System.out.println(event.toString() + " " + parser.getLong());
                                        break;
                                }
                        }
                } catch (FileNotFoundException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                }
        }
```

Este ejemplo consta de tres pasos.
Obtenga una instancia de analizador llamando al método estático Json.createParser.
Itere los eventos del analizador con los métodos JsonParser.hasNext y JsonParser.next.
Realice el procesamiento local para cada elemento.

El ejemplo muestra los diez tipos de eventos posibles del analizador. El siguiente método del analizador lo avanza al siguiente evento. Para los tipos de evento KEY_NAME, VALUE_STRING y VALUE_NUMBER, puede obtener el contenido del elemento llamando al método JsonParser.getString. Para VALUE_NUMBER eventos, también puede usar los siguientes métodos:
JsonParser.isIntegralNumber
JsonParser.getInt
JsonParser.getLong
JsonParser.getBigDecimal

1.4 Crear JSON

Si lo queremos hacer con la API de modelos, este sería un ejemplo:

```
// Java program for write JSON to a file

import java.io.FileNotFoundException;

import java.io.PrintWriter;

import java.util.LinkedHashMap;

import java.util.Map;

import org.json.simple.JSONArray;

import org.json.simple.JSONObject;


public class JSONWriteExample

{

	public static void main(String[] args) throws FileNotFoundException

	{

		// creando JSONObject

		JSONObject jo = new JSONObject();


		// poniendo los primeros datos en JSONObject

		jo.put("nombre", "Fran");

		jo.put("apellido", "Garcia");

		jo.put("edad", 25);


		//  para la dirección primero hay que crear un LinkedHashMap

		Map m = new LinkedHashMap(4);

		m.put("direccion", "Lillo Juan, 128");

		m.put("ciudad", "San Vicente del Raspeig");

		m.put("comunidad", "Valenciana");

		m.put("codigoPostal", “03690”);


		// domicilio a JSONObject

		jo.put("domicilio", m);
```

```
                    // para los números de teléfono primero crear el JSONArray
                    JSONArray ja = new JSONArray();

                    m = new LinkedHashMap(2);
                    m.put("tipo", "casa");
                    m.put("numero", "666 666 666");
                    // añadiendo a la lista
                    ja.add(m);
                    m = new LinkedHashMap(2);
                    m.put("tipo", "movil");
                    m.put("numero", "777 777 777");
                    // añadiendo a la lista
                    ja.add(m);

                    // añadiendo los números de teléfono al JSONObject
                    jo.put("numerosTelefonos", ja);

                    // Escribiendo el:"profesor.json" in cwd
                    PrintWriter pw = new PrintWriter("JSONExample.json");
                    pw.write(jo.toJSONString());
                    pw.flush();
                    pw.close();
            }
}
```

Para escribir un ejemplo usando la API de streaming podría valernos el siguiente ejemplo:

```
FileWriter writer = new FileWriter("test.txt");

JsonGenerator gen = Json.createGenerator(writer);

gen.writeStartObject()

  .write("firstName", "Duke")
```

```
  .write("lastName", "Java")
  .write("age", 18)
  .write("streetAddress", "100 Internet Dr")
  .write("city", "JavaTown")
  .write("state", "JA")
  .write("postalCode", "12345")
  .writeStartArray("phoneNumbers")
    .writeStartObject()
      .write("type", "mobile")
      .write("number", "111-111-1111")
    .writeEnd()
    .writeStartObject()
      .write("type", "home")
      .write("number", "222-222-2222")
    .writeEnd()
  .writeEnd()
.writeEnd();
gen.close();
```

Este ejemplo obtiene un generador JSON llamando al método estático Json.createGenerator, que toma un writer o una secuencia de salida como parámetro. El ejemplo escribe datos JSON en el archivo test.txt anidando llamadas a los métodos write, writeStartArray, writeStartObject y writeEnd. El método JsonGenerator.close cierra el escritor subyacente o la secuencia de salida.

1.5 Para profundizar…

Puedes leer algunos de los documentos que han servido como bibliografía para la creación de este apéndice:

https://www.oracle.com/technetwork/es/articles/java/api-java-para-json-2251318-esa.html

https://www.tutorialspoint.com/json/json_java_example.htm

https://www.geeksforgeeks.org/parse-json-java/

https://javaee.github.io/tutorial/jsonp004.html