Manejo de XML desde C#

Contenido

1 Manejo de DOM

1.1 Carga de datos mediante XML DOM

Los datos guardados en formato XML son datos almacenados en formato texto con estructura. Esto quiere decir que, a *priori*, no es posible saber, por ejemplo, cuantos campos contiene un determinado *registro*. Utilizando DOM, es posible recoger únicamente aquellos datos en los que se está interesado, haciendo que la aplicación, además, sea flexible y no dependa de una determinada sintaxis en cuanto a su fichero de datos. La única desventaja de este método es que precisa de suficiente memoria en el sistema como para albergar todos los datos, pues construye la estructura en memoria con toda la información disponible en el archivo.

Lo primero que se necesita es hacer un *using* del namespace X*ml*. El namespace *Text* también es importante, pues es en el que se encuentra la clase **Encoding**, para las diferentes codificaciones de texto.

```
using System.Xml;
using System.Text;
```

Lo anterior proporcionará el acceso, principalmente, a las clases *XmlDocument* , *XmlAtribute* y *XmlNode*, que serán las clases necesarias para poder realizar la carga del documento XML. Una vez dentro del procedimiento *cargar()* (el nombre es sólo un ejemplo), lo primero que es necesario hacer es cargar el documento entero

```
XmlDocument docXml = new XmlDocument();
docXml.Load( “datos_aplicacion.dat” );
```

Ahora, el contenido del documento está cargado en memoria, en forma de *XmlNodes*, que deberemos explorar para poder encontrar los datos que intentamos recuperar. Existe un nodo raíz, al que se accede mediante *docXml.DocumentElement*, y sobre el que es posible hacer un *foreach*, recorriendo todos sus nodos hijos (*ChildNodes*):

```
foreach(XmlNode nodo in docXml.DocumentElement.ChildNodes) {
     // … más cosas
}
```

De hecho, este esquema es recursivo: cada *XmlNode* tiene una propiedad *ChildNodes* que contiene, a su vez, *XmlNode*'s. Por ejemplo, dado un *XmlNode x* es posible recorrer recursivamente su contenido con una función como la que se muestra a continuación. Nótese que el nombre del nodo se guarda en la propiedad *Name*, mientras que la información guardada se puede acceder mediante la propiedad *InnerText*.

```
public static void recorrerNodos(XmlNode x) {
     foreach(XmlNode nodo in x.ChildNodes) {
          System.Console.Write( nodo.Name
                               + “: “ + nodo.InnerText
          );
```

```
            if ( nodo.ChildNodes.Count > 0 ) {
                recorrerNodos( nodo );
            }
        }
    }
```

Algunas veces, los nodos de un documento XML contienen también atributos. En ese caso, es necesario emplear la clase *XmlAttribute*, con un funcionamiento similar a *XmlNode*. El siguiente procedimiento mostraría todos los atributos de un nodo determinado.

```
    public static void recorrerAtributos(XmlNode x) {
        foreach( XmlAttribute atr in x.Attributes) {
            System.Console.Write( atr.Name
                                  + ": " + atr.InnerText
            );
        }
    }
```

1.2 Salvaguarda de datos en XML mediante DOM

La salvaguarda de datos mediante DOM es relativamente cómoda, aunque tiene una desventaja bastante grande, y es que, al crear la estructura de objetos en memoria, se está duplicando la información a guardar. Esto está más justificado al cargar, cuando la estructura del documento XML no tiene por qué estar claramente prefijada. Al margen de ésto, es una opción perfectamente válida.

Lo primero, como siempre, es crear el documento XML para DOM:

```
XmlDocument docXml = new XmlDocument();
```

A continuación, se crea la información de control de XML, esa cabecera con etiqueta *<!xml* al comienzo de un documento.

```
XmlNode nodo = docXml.CreateNode( XmlNodeType.XmlDeclaration,
                "xml", ""
);
((XmlDeclaration)nodo).Encoding = "utf-8";
```

Una vez creada la etiqueta, ésta se añade al documento, lo que será una constante al trabajar de esta forma.

```
docXml.AppendChild( nodo );
```

A continuación, el nodo principal, raíz, debe ser creado:

```
XmlNode nodoPpal = docXml.CreateNode(
                    XmlNodeType.Element, “Personas”, ""
);
docXml.AppendChild( nodoPpal );
```

A partir de este momento, dejaremos de añadir los nodos al documento XML, y los añadiremos al nodo principal.

Ahora, se pueden empezar a crear los nodos del documento:

```
XmlNode nodo = docXml.CreateNode(
          XmlNodeType.Element, “Persona”, ""
);

XmlAttribute atr = docXml.CreateAttribute( “Nombre” );
atr.InnerText = “Baltasar”;
nodo.Attributes.Append( atr );

nodo.InnerText = “Universidad de Vigo”;
nodo.AppendChild( nodoPpal );
```

Y, finalmente, el archivo se guarda, convirtiéndose a XML automáticamente.

```
docXml.Save( “personas.dat” );
```

2 Manejo del XML Parser de C#

El parser de C# es verdaderamente potente, y puede ser empleado para cargar y, especialmente, para guardar datos (la carga de datos puede ser muy onerosa por este método).

2.1 Salvaguarda de datos mediante el XML Parser de C#

En primer lugar, es necesario crear un documento que permita crear las estructuras XML que se necesitarán para poder guardarlas posteriormente (aunque, al contrario que con DOM, en este caso se van guardando las marcas directamente).

```
XmlTextWriter textWriter =
     new XmlTextWriter( "estudiantes.xml", Encoding.UTF8 )
;
```

Se comienza el documento. Esto escribe la información en la etiqueta *<!xml* del comienzo.

```
textWriter.WriteStartDocument();
```

A continuación, es necesario crear la etiqueta que va a ser el nodo principal del documento. Nótese que por cada etiqueta que se abra, será necesario recordar cerrarlas.

```
textWriter.WriteStartElement( "Estudiantes" );
```

Es el momento de crear un registro de esta pequeña base de datos. Se comienza, como siempre en XML, con un nodo.

```
textWriter.WriteStartElement( "Estudiantes" );
```

… al que se le añadirá un atributo.

```
textWriter.WriteStartAttribute( "Nombre" );
textWriter.WriteString( “Baltasar” );
textWriter.WriteEndAttribute();
```

Es el momento de darle un valor al nodo estudiante, y cerrarlo finalmente.

```
textWriter.WriteString( “Universidad de Vigo” );
textWriter.WriteEndElement();
```

Así como de cerrar el nodo principal del documento XML.

```
textWriter.WriteEndElement();
```

… y finalizar de escribir el documento.

```
textWriter.WriteEndDocument();
textWriter.Close();
```

2.2 Recuperación de datos mediante XML parser

El peso de la recuperación de datos la lleva la clase XmlTextReader, de la que tendremos que crear una instancia para poder empezar a trabajar sobre el archivo XML.

```
XmlTextReader textReader = new XmlTextReader( "uni.xml" );
```

Una vez hecho esto, en realidad ya no se puede hacer mucho más. Ahora se trata de recuperar cada uno de los elementos XML (sean atributos o nodos) del documento XML. Para cada elemento, y según el orden en el que se hayan leído (es probable que sea necesario programar un autómata finito de estados para hacerlo correctamente), se va recuperando la información.

```
while( textReader.Read() ) {

     XmlNodeType nType = textReader.NodeType;

     if ( nType == XmlNodeType.Attribute ) {

                    // … más cosas

     }

     if ( nType == XmlNodeType.Element ) {

                     // … más cosas

     }
}
```

3 Linq

XML Linq permite acceder y recuperar información desde colecciones de elementos, utilizando una sintaxis muy similar a SQL. Para poder acceder a Linq, es necesario, en el apartado de referencias del proyecto, incluir *System.Data.Linq*.

Se muestra un ejemplo a continuación:

```
int[] v = { 4, 2, 3, 1, 5, 6, 9 ,8 ,7, 10 };

IEnumerable<int> res =
                    from x in v
                    where x % 2 == 0
                    select x
                    orderby x;

foreach(int x in res) {
     System.Console.Write( "{0} ", x );
}
```

El código más arriba toma del vector v aquellos elementos que son pares, y los introduce en la colección *res*. Por supuesto, no es estrictamente necesario Linq para hacer esto, sino que sería posible crear una colección con **List**<>, e ir introduciendo aquellos elementos que cumplen la condición citada. Con Linq, todo ese trabajo se reduce muchísimo.

La siguiente versión del mismo código emplea funciones *lambda*, lo cual proporciona muchísimas posibilidades.

```
int[] v = { 4, 2, 3, 1, 5, 6, 9 ,8 ,7, 10 };

IEnumerable<int> res = v.Where( x => x % 2 == 0 )
                        .Select(x => x * 2)
                        .OrderBy(x => x);

foreach(int x in res) {
     System.Console.Write( "{0} ", x );
}
```

La única diferencia con el código anterior, es que el select toma el valor que cumple con la condición en *where*, y lo multiplica por dos.

4 XML Linq

Xml Linq es la opción que resulta, con diferencia, más sencilla de usar. Para poder acceder a XML Linq, es necesario, en el apartado de referencias del proyecto, incluir *System.Xml*, *System.Text* y *System.Xml.Linq*.

Es importante tener en cuenta que, aunque el código resulte mucho más sencillo, las operaciones que se disparan para procesar el archivo XML son de la misma complejidad.

4.1 Escribir XML

Escribir XML es tan sencillo como crear un elemento raiz, y hacer colgar de él los subnodos necesarios.

Con el siguiente código, se crea un archivo XML en el que se guardan los nombres e *e.mails* de dos personas.

```
var raiz = new XElement( "personas",
               new XElement( "persona",
                    new XElement( "nombre", "Baltasar" ),
                    new XElement( "email", "jbgarcia@uvigo.es" ) ),
               new XElement( "persona",
                    new XElement( "nombre", "Lourdes" ),
                    new XElement( "nombre", "lborrajo@uvigo.es" ) )
               );

raiz.Save( "personas.xml" );
```

4.2 Leer XML

En el siguiente código, se utiliza XML Linq para recuperar el e.mail del elemento cuyo nombre es Baltasar.

```
XElement raiz = XElement.Load( "personas.xml" );

IEnumerable<string> emails =
     from el in raiz.Elements( "persona" )
               where (string)el.Element( "nombre" ) == "Baltasar"
               select (string)el.Element( "email" );

foreach(string email in emails) {
     System.Console.WriteLine( email );
}
```

5 Bibliografía

- Los 101 ejemplos de Linq:
  - https://code.msdn.microsoft.com/101-LINQ-Samples-3fb9811b
- Utilizar funciones lambda con Linq
  - https://msdn.microsoft.com/es-es/library/bb397675.aspx
- Linq a XML vs. XML DOM
  - https://msdn.microsoft.com/es-es/library/bb387021.aspx