Geração dinâmica de APIs Perl para criação de XML

José João Almeida and Alberto Manuel Simões

Departamento de Informática
Universidade do Minho
{jj|ambs}@di.uminho.pt

Resumo É consensual que o XML como linguagem para a estruturação de documentos tem vindo a tomar um lugar relevante. É também evidente a vantagem obtida no uso de XML como linguagem de intercâmbio. No entanto, a sua sintaxe é demasiado descritiva pelo que a geração de documentos de forma manual é dolorosa sendo útil dispor de módulos que simplifiquem essa tarefa.
Neste artigo propomos um módulo Perl (XML::Writer::Simple) configurável via DTD que simplifica a tarefa de gerar XML.

1 Introdução

Antes de iniciar a apresentação do módulo Perl XML::Writer::Simple, vamos apresentar as vantagens e possibilidades de usar uma API para a geração de documentos XML, analisando um módulo que gera documentos XHTML.

1.1 Vantagens do uso de uma API de geração

Desde que o Perl tem vindo a ser usado como uma das principais ferramentas para a geração de páginas Web, bem como de formulários e *sites* dinâmicos, que existe um módulo Perl denominado CGI[4]. Este módulo, para além de conter uma grande panóplia de funções para a manutenção de formulários e parâmetros de cgis, contém um conjunto de funções úteis para a geração de XHTML[5].

Basicamente, para cada elemento válido em XHTML passa a existir uma função no ambiente de programação Perl, pelo que a simples invocação da função p com uma string como argumento, gera o XHTML do respectivo parágrafo.

A tabela 1.1 mostra alguns exemplos de como as funções do módulo CGI são usadas, e qual o resultado prático em XHTML.

Desta tabela salienta-se a facilidade com que se criam elementos mistos (segunda e terceira linha da tabela), bem como o suporte natural para atributos (quarta linha). Por fim, é demonstrada a possibilidade de uso de mapeamentos de elementos a listas.

As vantagens do uso de funções para a construção de documentos XHTML são várias:

Código CGI	Código XHTML
`p("bar")`	`<p>bar</p>`
`p("foo", "bar")`	`<p>foo bar</p>`
`p("foo", b("zbr"), "bar")`	`<p>foo <b>zbr</b> bar</p>`
`a({href=>"http://www.sapo.pt"}, "sapo")`	`<a href="http://www.sapo.pt">sapo</a>`
`ul(li(["a","b","c"]))`	`<ul><li>a</li><li>b</li><li>c</li></ul>`

Tabela 1. Utilização básica do CGI

– **escrita compacta**

Como sabemos o XHTML por vezes é demasiado descritivo, o que torna o processo de escrita demorado e moroso. Uma das principais vantagens da sintaxe do módulo CGI é o facto de não nos termos de preocupar com o nome do elemento que estamos a fechar (e portanto, garantimos[1] que o XML é bem formado)
Considere-se o seguinte exemplo:

```
1    <ul>
2      <li><b>Bold</b></li>
3      <li><i>Italic</i></li>
4      <li><strong>Strong</strong></li>
5      <li><code>Code</code></li>
6    </ul>
```

que seria gerado facilmente com:

```
1    ul(li([
2        b("Bold"),
3        i("Italic"),
4        strong("Strong"),
5        code("Code")
6          ]))
```

É fácil notar que a versão funcional é mais compacta e legível.

– **possibilidade de definição de macros pelo utilizador**

Dada a fácil integração do módulo com a linguagem de programação, torna-se simples para o utilizador a definição de novas funções, ou *macros* para a geração de código.
Consideremos um exemplo típico, de geração de hiper-ligações cujo texto ligado é o próprio valor da ligação. A escrita habitual em CGI deveria ser:

```
1    a({href=>"http://www.sapo.pt/"}, "http://www.sapo.pt/");
```

No caso de este tipo de ligações ser comum no documento ou aplicação que está a ser desenvolvida, o utilizador pode definir uma macro como:

[1] Na verdade o utilizador pode sempre forçar um XML mal formado mas se as funções forem utilizadas da forma correcta a boa formação do documento é garantida.

```
1  sub alink { a({href=>$_[0]}, $_[0]) }

2  alink("http://www.sapo.pt/");
```

Da mesma forma, se ao desenvolver um determinado *site* se está a usar uma tabela para mostrar o conteúdo de uma notícia, torna-se simples definir uma macro que abrevie todo o trabalho:

```
1  sub new_entry {
2    table(Tr(th(class=>"new",$_[0])), Tr(td($_[1]])))
3  }

4  new_entry("500 mortos nas estradas", "No último ano...");
```

Repare-se também que se a determinado ponto o webmaster decidir mudar o XHTML gerado de forma a tornar mais acessível a página web (deixando de usar tabelas para design) bastará para isso alterar o conteúdo desta função.

– **ligação a uma Linguagem de Programação**

Pelo facto de dispormos de uma linguagem de programação, passa a ser possível calcular dinamicamente partes constituintes do documento XHTML com base em funcionalidade, bases de dados, bases de conhecimento, etc. a que a linguagem de programação possa aceder.
Como o módulo está desenhado para uma linguagem de programação específica, e as suas estruturas de dados, torna-se simples o uso das mesmas para a geração rápida de XHTML:

```
1  @amigos=("Rui", "Ana", "Eva");

2  print ul(li([@amigos]))
```

Por outro lado, temos toda a liberdade de uso da linguagem de programação em causa. Supondo que a base de dados `dicDeLinks` contém urls ligados a uma determinada palavra-chave (por exemplo ligada através de uma `DB_File`) considere-se o seguinte extracto:

```
1  sub keyword {
2    my $k=shift;
3    a({href=> $dicDeLinks{$k}}, b($k)) }

4  print p(keyword("xata"));
```

Notas:

linha 3: a base de dados `dicDeLinks` contém urls ligados a uma determinada keyword.

linha 4: imprime `<p><a href="http://www.xata.org"><b>xata</b></a></p>`

2 XML::Writer::Simple

O módulo que aqui apresentamos — XML::Writer::Simple — pretende facilitar a geração de XML ligado a um ambiente específico. Existem outros módulos com este objectivo como o XML::Writer[3] mas que acabam por não facilitar nem se adaptar ao DTD em uso.

O módulo CGI baseia-se num conjunto fechado de etiquetas pelo que a sua definição poderia ter sido feita criando uma função para cada uma das etiquetas existentes. No entanto, ao estarmos a manusear XML é imprescindível que estas funções sejam geradas em tempo real.

Assim, o utilizador do módulo XML::Writer::Simple deve indicar um DTD[2] sobre o qual vai trabalhar. O módulo, ao ser carregado, interpreta o DTD[1] e extrai o conjunto de etiquetas válidas assim como outras propriedades.

2.1 XML::Writer::Simple a partir dum DTD

Vamos supor que um determinado DTD contém as seguintes definições:

```
1 <!ELEMENT     a   (b | c )                    >
2 <!ELEMENT     b   (c | #PCDATA)               >
```

A análise do referido DTD por parte do XML::Writer::Simple cria uma conjunto de funções (a,b,c) que, quando usadas de acordo com o exemplo seguinte:

```
1 use XML::Writer::Simple dtd => "ex.dtd";
2 print a(b("p1",c("p2"),"p3"));
```

geram o seguinte XML

```
1 <a>
2    <b>p1<c>p2</c>p3</b>
3 </a>
```

Além da definição destas funções, o módulo também irá criar um conjunto de PowerTags, de que iremos falar na secção 2.3.

2.2 XML::Writer::Simple a partir dum exemplo XML

A utilização de um exemplo XML para a definição das funções de geração de XML passa pela inferência do respectivo DTD[1].

Deste modo, o módulo pode ser usado com:

```
1 use XML::Writter::Simple xml => "ex.xml";
2
2 print a(b("p1",c("p2"),"p3"));
```

[2] Na verdade, como veremos mais à frente, o utilizador poderá usar um XML exemplo em vez de um DTD, ou ainda indicar a lista de elementos válidos.

2.3 PowerTags

As PowerTags são funções que representam mais do que uma etiqueta. É habitual em XHTML e, genericamente, em XML, existir etiquetas que só podem ter um tipo de filhos. Em XHTML são exemplos deste tipo de etiquetas as `ul` e `ol` que só podem conter `li`. Este tipo de propriedades podem ser inferidas a partir do DTD de forma a criar funções que automatizem a criação deste tipo de estruturas.

A tabela 2.3 mostra como estas funções podem abreviar a escrita. No exemplo em causa usamos as etiquetas `ul` e `li` do XHTML apenas como exemplo.

Código XML::Writer::Simple	**Código XML**
`ul_li("a","b","c")`	`<ul><li>a</li><li>b</li><li>c</li></ul>`
`ul_li({attr=>"val"},"a","b")`	`<ul attr="val"><li>a</li><li>b</li></ul>`

Tabela 2. Utilização de PowerTags

Em certos casos estas PowerTags não podem ser inferidas a partir do DTD porque, na realidade, não são correctas. Por exemplo, os `tr` do XHTML permitem mais do que um tipo de filhos (`td`, `th`). Dada a utilidade real de uma PowerTag para compor `tr` com `td` (`tr_td`), existe um método do módulo para forçar a sua criação:

```
1  powertag("tr","td");

2  tr_td("a","b","c");
```

Este tipo de raciocínio pode ser generalizado definindo PowerTags com mais do que um nível. Por exemplo, para a definição directa de tabelas usando dois níveis de listas aninhadas podemos usar:

```
1  use XML::Writer::Simple;

2  powertag("table","tr","td");

3  table_tr_td( ["a","b","c"],["d","e","f"] );
```

Com o resultado de

a	b	c
d	e	f

```
1  <table>
2    <tr> <td>a</td><td>b</td><td>c</td> </tr>
3    <tr> <td>d</td><td>e</td><td>f</td> </tr>
4  </table>
```

O mesmo resultado poderia ser obtido usando o comando `powertags` directamente na importação do módulo, como se segue:

```
1  use XML::Writer::Simple powertags=>[qw/table_tr_td/];

2  table_tr_td( ["a","b","c"],["d","e","f"] );
```

2.4 Exemplo: árvore de directorias em XML

O seguinte exemplo académico demonstra o uso do módulo XML::Writer::Simple para a construção de uma árvore de directorias em XML, usando a etiqueta `file` para anotar ficheiros, e a etiqueta `dir` para delimitar os ficheiros contidos nessa directoria (usando o atributo `name` para armazenar o nome da directoria em causa):

```
1  #!/usr/bin/perl -w
2  use XML::Writter::Simple tags => [qw/dir file/];
3  my $dir = shift;
4  print process($dir);
5  sub process {
6    my $dir = shift;
7    my $xml;
8    chdir $dir;
9    for my $file (<*>) {
10     if (-d $file) { $xml .= process($file) }
11     else          { $xml .= file($file)    }
12   }
13   chdir "..";
14   return dir(name=>$dir,$xml)
15 }
```

Notas:

linha 2: carregar o módulo indicando-lhe as etiquetas válidas;
linha 4: processar a directoria base;
linha 8: mudar para a directoria a processar;
linha 9: para cada ficheiro existente...
linha 10: no caso de ser directoria, processar recursivamente;
linha 11: no caso de ser ficheiro, criar etiqueta respectiva;
linha 14: criar a etiqueta referente à directoria;

O output gerado é semelhante a:

```
1  <dir name="etc">
2     <file>passwd</file>
3     <file>group</file>
4     <dir name="httpd">
5        <dir name="conf">
6           <file>httpd.conf</file>
7        </dir>
8     </dir>
9  </dir>
```

3 Implementação

O uso de uma linguagem como o Perl para a implementação deste módulo facilita em grande parte a tarefa. Dada que a linguagem é reflexiva, torna-se possível definir funções em tempo de execução.

Por outro lado, dada a forma como internamente as invocações de funções de um módulo Perl são realizadas, é possível a escrita de uma função de ordem superior[2] que trate de as interceptar (AUTOLOAD). Assim, deixa de ser necessária a definição de um método por etiqueta existente, mas apenas validar a etiqueta, criar a função correspondente e executar em conformidade.

Em termos de eficiência, é possível que a função de AUTOLOAD sempre que é chamado, em vez de executar o código em causa para gerar o XML respectivo, gere em tempo real a função que gera esse código, de forma a que da próxima vez que essa mesma etiqueta seja usada, a função AUTOLOAD[3] já não seja usada.

```
1  sub AUTOLOAD {
2     my $attrs = {};
3     my $tag = our $AUTOLOAD;
4     $attrs = shift if ref($_[0]) eq "HASH";

5     if (exists($PTAGS{$tag})) {
6       my @tags = @{$PTAGS{$tag}};
7       my $toptag = shift @tags;
8       return _xml_from($toptag, $attrs,
9                        _go_down(\@tags, @_));
10    } else {
11      return _xml_from($tag,$attrs,@_);
12    }
13 }
```

Notas:

linha 3: Nome do método invocado;
linha 4: Extrair atributos da invocação;
linha 5: Validar se estamos perante uma PowerTag;
linha 6,7 e 8: Processar recursivamente a PowerTag;
linha 11: Gerar função respectiva a etiqueta simples;

No caso das PowerTags, há necessidade de análise do DTD do documento. Para não obrigar a que o DTD seja analisado sempre que determinado método é chamado e não reconhecido como etiqueta, as PowerTags são definidas na altura em que o módulo é invocado.

[3] O código aqui apresentado não é o código Perl real. Apenas pseudo-código para ilustrar a ideia.

4 Conclusões

A possibilidade de tornar a geração de XML embutida numa linguagem de programação, torna mais eficaz este processo. Não só se poupa espaço, como o código se torna mais legível e garante maior fiabilidade no XML gerado.

Se houver os cuidados necessários na definição das funções de geração de XML pode-se obter XML correctamente (ou quase) identado, e mais legível do que o habitual XML gerado por aplicações.

Com as PowerTags mostramos ainda que com uma análise pequena ao DTD se torna possível a definição de macros automáticas.

Referências

1. José João Almeida and Alberto Manuel Simões. Inferência de tipos em documentos xml. pages 144–154, February 2005.
2. Mark Jason Dominus. *Higher Order Perl*. Morgan Kaufman, 2005.
3. David Megginson. *XML::Writer — Perl extension for writing XML documents*, Perl man pages edition.
4. Lincoln D. Stein. *CGI — Simple Common Gateway Interface Class*, Perl man pages edition.
5. W3C HTML Working Group. XHTML 1.0 The Extensible HyperText Markup Language (Second Edition). Technical report, W3C - World Wide Web Consortium, 2002. `http://www.w3.org/TR/xhtml1/`.