Mark Overmeer

XML and Perl

Perl als Klebstoffsprache

Als das Internet noch jung war, wurde Perl zur ultimativen Sprache um das Betriebssystem, die Datenbanken und Webseiten zusammenzubringen. Es war nicht allzu schwierig, die Featureliste von awk, shell und sed zu übertreffen; Perl wurde bald zu einem gewichtigen Spieler. Aber man bleibt nicht automatisch an der Spitze: Eine Sprache muss sich mit der Welt um sie herum weiterentwickeln. Perl hat seine führende Position in vielen Gebieten verloren, zum Beispiel durch das totale Ignorieren von XML-basierten Standards.

Nicht ohne Grund hat XML unter Perl-Programmierern einen schlechten Namen. Perl-Leute mögen Programme, die mächtig sind, sich an DWIM ("Do what I mean") halten und effizient arbeiten. Die XML-Umgebung ist extrem geschwätzig, formell und wurde oft von Leuten mit wenig Programmiererfahrung designed.

Zum Beispiel wurden die XML Schemata ganz klar von Bibliothekaren konzipiert. Der Standardtyp eines Elements ist *anyType*, was in Programmierersprache bedeutet: "Es ist mir egal, es ist jetzt Dein Problem". Programmierer beginnen mit einem klar definierten einzelnen Bit, fügen diese zu Bytes zusammen, und so weiter.

Also hassen Perl-Leute XML von Natur aus. "Sie" haben sich in YAML verliebt, das sehr einfach scheint und daher sehr stark bevorzugt wird. Regel 1 in der Programmierung: Wenn Du denkst, dass etwas Neues viel besser und schneller ist als etwas Existierendes, hast Du wahrscheinlich nicht alle Komplikationen verstanden. So gibt es aktuell keine volle Unterstützung des neuen YAML Standards in Perl. Das neue Ideal ist JSON, YAML ist vorbei.

Einführung von XML::Compile

In der Zwischenzeit hat sich die "Professionelle Welt" auf XML standardisiert. Wenn Perl wieder Boden als generische Klebstoffsprache gut machen möchte, sollte es Implementierungen für die meisten XML-basierten Standard bieten. Es gibt nur ganz wenige dieser Implementierungen auf CPAN. Wahrscheinlich weil es keine Basis-Standard-Bibliothek gab, um diese umzusetzen.

Auf CPAN kannst Du einige Dutzend Module finden, die XML lesen und schreiben können. Aber diese können XML nur auf dem einfachsten Weg behandeln. Moderne XML-Protokolle sind Schema-basiert: Typen und Strukturen sind im Detail beschrieben. Die Schemata werden größer und größer. Es ist schrecklich, hunderte dieser Elemente in so einem Standard nach Perl zu übersetzen, Anweisung für Anweisung, Knoten für Knoten - so wie es die meisten CPAN-Module verlangen.

Seit 2006 habe ich an einer intelligenten Unterstützung für XML Schemata gearbeitet, später auch für SOAP und WSDL Standards. Schemata sind groß und Perl ist relativ langsam, also habe ich entschieden, die Last der Schema-Verarbeitung in die Initialisierungsphase des Programms zu legen um die Laufzeit schnell zu machen. In anderen Implementierungen (wie Java-Bibliotheken) werden Nachrichten zur Laufzeit gegen das Schema gematcht; sie laufen interpretiert. `XML::Compile` übersetzt die Schemata in wiederverwendbare Codereferenzen, die XML nach Perl, Perl nach XML übersetzen und sogar Beispiel-XML-Nachrichten und -Perl-Hashes generieren können.

XML lesen

Eigentlich ist das Schema-getriebene Lesen sehr einfach mit `XML::Compile`:

```
# compile once
use XML::Compile::Schema;
my $schema =
  XML::Compile::Schema->new($schemafn);
my $reader =
  $schema->compile(READER => $type);

# run often
my $hash   = $reader->($xml);

# Data::Dumper is your friend
print Dumper $hash;
```

Das XML (`$xml`), das verarbeitet werden soll, kann entweder ein String, ein Dateiname mit XML oder ein `XML::LibXML::Document` sein. `XML::Compile` basiert auf `XML::LibXML`, einem XS-Wrapper um Gnomes *libxml2*. Stelle sicher, dass Du eine aktuelle Version von beidem installiert hast, weil in den älteren Versionen viele Bugs enthalten sind.

Etwas schwieriger ist das `$type`. Du musst herausfinden welches Element im Schema das oberste Element in Deiner Nachricht ist. Vielleicht aus der Dokumentation. Und dann haben diese Element sowohl einen Namensraum als auch einen Namen in diesem Namensraum: ein Paar. Es ist nicht sehr praktisch, Paare herumzureichen. Deshalb erstellt Du `$type` so:

```
my $type = "{$namespace}$name";

# cleaner
use XML::Compile::Util 'pack_type';
my $type = pack_type $namespace, $name;
```

Nach ein paar Monaten habe ich festgestellt, dass es sehr unpraktisch ist, diese kompilierten Codereferenzen zwischen Funktionen und Modulen hin- und herzureichen. Es wäre viel schöner eine Art generischen Schemamanager zu haben. Also wurde ein neues Modul hinzugefügt, das diese kompilierten Strukturen einsammelt. Dieses Modul vereinfacht auch die Typspezifikation mit Präfixen. Das erste Beispiel umgeschrieben:

```
# compile once
use XML::Compile::Cache;
my $schema =
    XML::Compile::Cache->new(
       $schemafn, allow_undeclared => 1);
$schema->prefix(xyz => $namespace);

# run often
my $hash = $schema->reader(
                      "xyz:$name")->($xml);
```

Der *reader* wird bei der ersten Verwendung kompiliert. Das ist aber nicht das was Du in Daemons haben willst: In diesem Fall willst Du immer alle möglichen *reader* kompilieren bevor die Kindprozesse geforkt werden. In diesem Fall kannst Du das benutzen:

```
# compile once in the parent
use XML::Compile::Cache;
my $schema =
   XML::Compile::Cache->new($schemafn);
$schema->prefix(xyz => 'http://something');
$schema->declare(READER => "xyz:$name");
$schema->compileAll;

...fork...
# run often in any child
my $hash =
   $schema->reader("xyz:$name")->($xml);
```

Validierung

Alle Elemente in `$xml` werden während des Einlesens validiert. Zur selben Zeit werden sie zu benutzbaren Perl-Werten umgewandelt. Zum Beispiel müssen für einige Datentypen die optionalen Leerzeichen entfernt werden (whitespace 'collapse'). Die *dateTime*-Werte werden in passende Zeiteinheiten übersetzt, base64-kodierte Daten werden dekodiert, die boolschen Werte *false* und *true* werden zu '0' und '1', und so weiter.

Einige Parameter können zur Optimierung der Übersetzungen genutzt werden. Mit diesen Standardwerten:

```
validation      => true
check_values    => true
check_occurs    => true
ignore_facets   => false
sloppy_integers => false
sloppy_floats   => false
```

Die letzten beiden Parameter können oft verwendet werden. Die Schematypen *integer* und *float* erlauben Werte, die nicht in Perls Integer und Float passen. `XML::Compile` steckt diese

Werte in ineffiziente `Math::BigInt` und `Math::BigFloat` Objekte. In vielen Fällen hätte der Schema-Designer besser Typen wie 'unsignedShort' verwendet. Nachlässig ("sloppy") zu sein bedeutet: Zusichern, dass keine großen Werte vorkommen.

```
$schema->declare(
  READER => "xyz:$name", sloppy_float => 1);
```

Ein typisches Programm von mir enthält viele Überprüfungen der Eingaben, auf die Struktur der Daten und die Plausibilität der Werte. In meinem Code wird 3/4 der Logik für diese Aufgabe verwendet: Erstellen und Handling von Eingabeparametern von Skripten und Funktionen. Wie auch immer, wenn das Schema strikt ist, wird das XML (automatisch) validiert bevor es in Dein Programm kommt. Es werden weniger Tests benötigt. Wenn zum Beispiel das Schema das Folgende enthält:

```
<element name="gender">
  <simpleType>
    <restriction base="token">
      <enumeration value="male"    />
      <enumeration value="female"  />
      <enumeration value="unknown" />
    </restriction>
  </simpleType>
</element>
```

`XML::Compile` erzeugt automatisch eine Fehlermeldung beim Geschlecht `none`, also braucht Dein Programm das nicht mehr zu verifizieren. Die Wirklichkeit ist aber härter: Viele Schemata sind unspezifisch.

Verschachtelte Hashes

Das Ergebnis des Einlesens ist eine einzige (manchmal sehr tiefe) Struktur von verschachtelten HASHes und ARRAYs. Lass uns einen Blick auf die Übersetzung eines üblichen Konstrukts werfen. XML Schemata haben die folgenden Bausteine:

```
<record>
  <price>3.14</price>

  <name lang="nl">ijsje</name>

  <order number="PO1234">
    <paid>false</paid>
  </order>
</record>
```

Das erste Element innerhalb des Eintrags ist ein *simple* Type. Danach folgt ein *simple* mit Attributen (`complexType` mit `simpleContent`). Nur der mittlere Fall ist etwas schwieriger auf einen einzelnen HASH zu mappen: Der Wert hat keinen Namen. Der `reader` gibt das obige Beispiel als folgende Struktur zurück:

```
record =>
  { price => Math::BigFloat->new('3.14')
  , name  => {lang => 'nl', _ => 'ijsje'}
  , order => {number => 'PO1234', paid => 0}
  }
```

Wenn ein Element mehrfach vorkommen kann, wird es immer als ARRAY zurückgegeben. Zum Beispiel:

```
<element name="x" type="int"
         maxOccurs="unbounded" />
x => [ 1, 2, 3 ]
x => [ 4 ]
```

Die vielen Nicht-Schema-getriebenen XML verarbeitenden Module - wie `XML::Simple` - würden im letzten Fall kein Array zurückliefern. Sie können mit Elementwiederholungen nur umgehen, wenn sie sie in der Nachricht sehen. Das macht die Verwendung der Daten ein wenig komplex:

```
# when you use XML::Simple
my $r = $data->{...}{x} || [];
my @x = ref $r eq 'ARRAY' ? @$r : $r;

# when you use XML::Compile
my @x = @{$data->{...}{x} || []};
```

Warum nicht einfach XML::(LibXML::)Simple verwenden?

Ein Seiteneffekt der Schema-basierten Verarbeitung ist die "Reinigung" der Werte. Oft sind die Typen im Schema sehr unterschiedlich zu den Typen in Perl, aber nur wenn sie im Detail betrachtet werden. Der Schematyp "integer" zum Beispiel, kann mit Leerzeichen um den Wert benutzt werden, aber auch mit Leerzeichen zwischen den Ziffern! Und er muss mindestens Dezimalzahlen mit 19 Ziffern (64 bit) unterstützen. Ist Dein Perl bzw. das Perl des Kunden mit 64-bit Integern kompiliert?

Wenn Du `XML::Simple` zum Einlesen Deiner Dateien verwendest, musst Du entweder sehr viel Zeit auf die Validierung und Bereinigung verwenden oder darauf vertrauen,

dass die Gegenseite die selbe Teilmenge aus dem Wertebereich verwendet wie Du es aus den Beispielen heraus erwartest. Du meinst vielleicht zu wissen, dass alle XML-Integer in die Perl-Integer passen. Du sagst vielleicht voraus, dass ein boolscher Wert mit '0' oder '1' kodiert wird und nicht mit 'true' und 'false'. Es ist auf jeden Fall auf lange Sicht eine unsichere Situation.

Schlüssel umschreiben

Standardmäßig sind die Schlüssel im Perl HASH die einfachen Namen der Elemente im XML; der Namensraum wird ignoriert. Das ist oft kein Problem, weil die Schemata sehr selten so sehr miteinander verstrickt sind dass Namenskollisionen entstehen. Andererseits möchtest Du vielleicht den Namensraum aus Gründen der Klarheit, aus Spaß oder nur zur Dokumentation sehen:

```
$schema->addKeyRewrite('PREFIXES');  # all!
$schema->addKeyRewrite('PREFIXES(abc,xyz)');
```

Auch ist der bevorzugte Stil der XML-Designer, kleingeschriebene Namen mit "-" zwischen den Worten zu benutzen. Hashschlüssel mit "-" sind in Perl 5 unhandlich (Perl 6 liebt sie). Mit der nächsten Zeile werden die Bindestriche in Unterstriche umgewandelt:

```
$schema->addKeyRewrite('UNDERSCORES');
```

Du kannst sogar HASHes mit "Name-zu-Schlüssel"-Mappings übergeben. Du kannst das dazu verwenden, Elemente direkt zu Datenbankfeldern zu mappen. Diese Key-Mappings beeinflussen aber nur die Schlüssel in Perl, nicht das XML das gelesen oder geschrieben wird.

Wenn zum Beispeil die `PREFIXES`- und `UNDERSCORE`- Regeln aktiviert sind, wird das Element mit dem Namen `{http://something}my-name` nicht in einem Schlüssel "my-name" sondern in dem Wort "xyz_my_name" enden.

Erzeugen von Beispielen

Diese Datenstrukturen können groß werden und die Schemata können sehr komplex sein, über mehrere Dateien mit komplexen Vererbungsverbindungen verteilt. Um Dir zu helfen, die Datenstrukturen zu verstehen, kann `XML::Compile` kommentierte Beispiele erzeugen:

```
print $schema->template(PERL => $type);
print $schema->template(XML  => $type);
```

Die Ausgabe kann nicht direkt verwendet werden (weil es alle Optionen einer Auswahl ausgibt), aber sie wird sich beim Verstehen der Datenstruktur als hilfreich erweisen. Wenn Du Regeln zum Umschreiben der Schlüssel definiert hast, werden Sie auch auf die Perl-Beispiele angewendet.

Schreiben

Schließlich kannst Du auch perfekte *writer* für XML Elemente schreiben. Du muss nicht auf die Reihenfolge der Elemente, Namensräume, Rundungsfehler, Typekonvertierung und so weiter achten: Es macht einfach was man möchte! Das Ergebnis wird validiert. Ich werde hier nur die Version mit `::Cache` zeigen:

```
use XML::Compile::Cache;
my $schema = XML::Compile::Cache->new($xsdfn
  , allow_undeclared => 1);
$schema->prefix(p => $namespace);

my $doc  = XML::LibXML::Document->new('1.0'
  , 'UTF-8');
my $node = $schema->writer("p:$name")
                       ->($doc, $hash);
$doc->setDocumentElement($node);
print $doc->toString(1);
```

Wenn Dein Schema hässliche Konstrukte wie "any" verwendet, musst Du mehr als einen *writer* aufrufen um den gesamten DOM-Baum aufzubauen. All diese Subbäume des Ergebnisdokuments müssen mit dem gleichen `$doc` gebaut werden. Das ist der Grund für die drei zusätzlichen Zeilen rund um den eigentlichen *writer*.

Sei gewarnt, dass es in `XML::LibXML` ein paar Unterschiede zwischen dem Aufruf von `toString()` auf einem Knoten und dem Dokument gibt. Nur der letzte Fall stellt sicher, dass das utf-8 encoding korrekt ist.

Ein Protokoll implementieren

Ein Protokoll, das auf XML basiert, ist sehr einfach zu unterstützen, so lange es ein Schema dazu gibt. Wir haben *reader* und *writer* in den vorherigen Beispiel gesehen: Nur ein Dutzend Zeilen Code. Wir müssen das in ein *package* verpacken und etwas Abstraktion bieten. Du kannst einige Beispiele auf CPAN finden, für verschiedene Verwendungen von XML wie die Nutzung von Dateien und SOAP. Schau Dir an, welches am besten passt. Das Basismodul liefert auch ein paar Beispiele mit.

Als erstes musst Du überlegen, wohin Du *xsd* (XML Schema) und *wsdl* (Beschreibung von Webservices) Dateien installierst. Üblicherweise werden auf UNIX-Systemen diese Dateien unter */usr/share/* oder */usr/local/share/* gespeichert. Das ist im Allgemeinen nicht sehr nützlich: Wenn Dein Programm startet, wo sind dann diese Dateien? Es ist auch nicht Plattformunabhängig. Wusstest Du, dass **alles** im lib-Verzeichnis einer Distribution installiert wird? In `Geo::KML` benutze ich zum Beispiel folgendes:

```
lib/Geo/KML.pm
lib/Geo/KML/xsd/kml-2.1/kml21.xsd
lib/Geo/KML/xsd/kml-2.2.0/kml22gx.xsd
lib/Geo/KML/xsd/kml-2.2.0/ogckml22.xsd
```

Schemata für verschiedene Versionen des Schemas werden angeboten - relativ zum Hauptmodul. Dieses Modul sammelt die Dateien mit den folgenden Zeilen ein:

```
package Geo::KML;
sub new(%)
{   my ($class, %args) = @_;
    my $version = $args{version} || '2.2.0';
    (my $dir    = __FILE__)
        =~ s!\.pm$!/xsd/kml-$version!;
    my @xsd     = glob "$dir/*.xsd";
    my $schema  =
        XML::Compile::Cache->new(\@xsd);
    bless {schema => $schema}, $class;
}

sub writeKML($$)
{   my ($self, $data, $filename) = @_;
    my $doc = XML::LibXML::Document->
                new('1.0', 'UTF-8');
    my $xml = $self->{schema}->
                writer('kml')->($doc, $data);
    ...
}
```

Es ist sehr einfach den Speicherort der konstanten Dateien auf Basis des Speicherorts des Pakets mit `__FILE__` zu ermitteln. Die Konstante gibt den absoluten Speicherort der Datei an. Achtung: *glob* behandelt Pfade mit Leerzeichen nicht korrekt.

Natürlich musst Du *declare* und *prefix* Deklarationen zum `$schema`-Objekt hinzufügen. Das ist oftmals auch versionsabhängig und die Anzahl der Versionen kann wachsen. Daher erstelle ich für gewöhnlich einen HASH mit diesen Einstellungen.

Das obige Setup bietet eine schöne Abstraktion, bei der das Schema außerhalb des Zugriffsbereichs des Endbenutzers liegt. Das ist auf der einen Seite sauber, auf der anderen Seite ist es unpraktisch zum Debuggen und zur Template-Generierung. Die andere Lösung wäre, Vererbung zu verwenden:

```
package Geo::KML;
use base 'XML::Compile::Cache';
sub init($)
{   my ($self, $args) = @_;
    my $version = $args{version} || '2.2.0';
    (my $dir    = __FILE__)
         =~ s!\.pm$!/xsd/kml-$version!;
    $self->importDefinitions(
              [glob "$dir/*.xsd"] );
    $self;
}

sub writeKML($$$)
{   my ($self, $data, $filename) = @_;
    my $doc    = XML::LibXML::Document->
                    new('1.0', 'UTF-8');
    my $xml    = $self->writer('kml')->
                    ($doc, $data);
    ...
}
```

Du kannst weitere Schemata mit `importDefinitions()` laden - zu jeder Zeit. So viele Du willst, oder unterschiedliche Versionen des Schemas wenn Du willst. Es kann auch dazu benutzt werden, fehlerhafte Schema-Komponenten zu überschreiben: Die zuletzt geladene Definition überschreibt die vorherigen Definitionen mit dem gleichen Namen.

Der *reader* ist ein wenig schwieriger: Wenn eine Datei zum Einlesen bereitgestellt wird, wissen wir nicht, welches Top-Element wir erwarten sollen. Wir können auch die Initialisierung mit einer bestimmten Version beginnen bis wir das Wurzelelement kennen.

```
my $data = Geo::KML->readKML($source);

package Geo::KML;
sub readKML($)
{   my ($class, $source) = @_;
    my ($root, %details) =
          XML::Compile->dataToXML($source);
    my $rootns  = $root->namespaceURI;
    my $version = $self->
          namespace2version($rootns);
    my $self = $class->
          new(version => $version);
    $self->reader($kml)->($root);
}
```

Normalerweise wird der *reader* Dateinamen und String automatisch nach XML parsen, mit der Methode `dataToXML()`. In diesem Fall müssen wir die Quelle selbst parsen um herauszufinden welcher Namensraum im Wurzelelement des Dokuments benutzt wird. Einige Magie ist nötig, um den Namensraum in die verwendete Version des Protokolls zu übersetzen.

In diesem Moment wurde das niedrigste Level an Support erreicht. Es wird erwartet, dass Benutzer die Datenstruktur verstehen und erzeugen können, die deren Information in XML repräsentieren. Werden wir nur so ein niedriges Level unterstützen? Es wäre viel schöner, eine Abstraktionsschicht über diesem Modul hinzuzufügen um die Struktur der Nachricht und Unterschiede der Versionen aus Benutzersicht zu verbergen. Manchmal ist das einfach, oft ist es eine große Aufgabe. Für KML ist es zu viel Arbeit.

Fazit

`XML::Compile` ist die Basis für eine wachsende Anzahl von Modulen auf CPAN: `XML::Compile::SOAP`, `::SOAP::Daemon`, `Geo::KML` und viele mehr. Wenn Dir die Geschichte gefällt, werde ich einige davon in der nächsten Ausgabe von $foo erläutern.