Twin City Perl Workshop 2008

Processing XML documents with

XML::LibXML

Petr Pajas
About me

• Dept. of Computational Linguistics, Charles University, Prague

• maintainer of XML::LibXML + LibXSLT
  – written by Matt Sergeant, Christian Glahn

• xsh (shell language for XML processing)
  – combines XPath, Perl, Shell
  – lots of usefull features
  – visit http://xsh.sourceforge.net
  or search CPAN for XML::XSH2
Introduction

• about 200 XML-* distributions on CPAN
  • XML::DOM, XML::Simple, XML::RSS, ... XML::LibXML
  • APIs, parsers, generators, filters
  • generic or vocabulary-specific
  • different processing strategies

• what to consider before you choose?
Processing strategy decision

- Data size:
  - small (kb) / Large (10s of MB) / Huge (100s of MB)
- Processing:
  - Read-only / Read->process->write / Stream
- Vocabulary:
  - custom (application specific) / defined by user, standard, customer, etc.
- Context requirement
- Required data model
  - nested structures / DOM tree / custom objects
- Data traversal (systematic / random cherry picking)
XML::LibXML
a swiss knife for all that

• interface to libxml2 library by Daniel Veillard
  see http://xmlsoft.org

• Standard compliant parser

• Validation: DTD, RelaxNG, XML Schema 1.0

• SAX, DOM, and Reader APIs

• XInclude, C14N, XPath, and XSLT 1.0 (with libxslt and XML::LibXSLT)
XML Parser

- libxml2 parser is very fast
- supports many character encodings
- parsing from files, strings, file handles (PerlIO), URLs (FTP and HTTP and possibly other via input callbacks)
- Parses gzipped XML documents
- Optional on-the-fly validation and XInclude processing
DOM Parser

- **building in-memory DOM Tree**
  ```perl
  my $parser = XML::LibXML->new;
  my $dom = $parser->parse_file($filename_or_URL);
  ```

- **easy navigation**
  - using DOM Level 2 API
  - XPath support

- **C data structures inside**
  - less memory usage than e.g. nested Hash/Arrays
  - accessors via object methods (XS)
    - slower than Perl hash/array accessors
    - if many, Perl->XS->Perl transitions get expensive
SAX Parsing

- Event-driven API for stream parsing
  - clean interface
  - requires little memory
  - user's class defines handler methods
    - `start_element`, `end_element`, `characters`, ...
- SAX layers can be chained:
  - parser -> filter -> filter -> .... -> handler/writer
- Slow for large documents
  - at least one (but usually much more) Perl method calls for each input node (multiply by millions)
- `XML->SAX, DOM->SAX, and SAX->DOM` interfaces
XML::LibXML::Reader

- Cursor-based API for stream XML parsing (pull-parsing)
  ```perl
  while ($reader->read) {
      my $type = $reader->nodeType;
      # start_element, end_element, text, comment,...
      my $el_name = $reader->name;
      my $depth = $reader->depth;
      my $val = $reader->getAttribute('foo');
      ....
  }
  ```

- Supports fast-forward skipping of uninteresting nodes
  - nextSibling
  - nextSiblingElement(name,nsURI)
  - nextElement(name,nsURI)
  - nextPatternMatch(XPathPattern)
Building DOM fragments

- $reader->copyCurrentNode( $deep )
  - extract a DOM node or subtree
- Creating sub-documents
  - $reader->preserveNode()
    - keep the current node in the in-memory tree
  - $reader->preservePattern(XPathPattern)
    - preserve matching elements along with all ancestors and descendants
  - $reader->document()
    - collect the DOM consisting of preserved nodes
Memory usage

Input: 15 MB of XML (dictionary)
Parser performance

- XML::* - Parsing + building data structures
  - Reader_2 = XML::LibXML::Reader+retrieve all names and values
  - Reader+AT = XML::LibXML::Reader + build nested Array tree
Accessor performance

- XML::LibXML : $el->name  250K/s
- XML::DOM      : $el->getTagName  500K/s
- Native hash   : $el->{name}    2200K/s
XPath querying

• Basic

$doc->findnodes(q{ /x/y[@foo="bar"] });
$node->findvalue(q{ count(preceding::y[not(@foo)]) });

• Advanced

my $xc = XML::LibXML::XPathContext->new();
$xc->registerNs('x', 'http://www.w3.org/1999/xhtml');
my @nodes = $xc->findnodes('//x:div',$doc);

• Pre-compiled (NEW)

my $exp = XML::LibXML::XPathExpression
    ->new($complex_xpath);
$node->findnodes($exp); # many times
XPath pattern matching

- Subset of XPath - path(s) of name tests
  
  ```perl
  $pat = XML::LibXML::Pattern
  ->new('..//foo|//p:bar/p:/*/*', { p=>'http://foobar/ns' });
  ```

- DOM node matching
  
  ```perl
  if ($pat->matchesNode($node)) { ... }
  ```

- Reader pointer matching
  
  ```perl
  $reader = XML::LibXML::Reader->new(location=>$url);
  ...
  if ($reader->matchesPattern($pat)) {...}
  ...
  while ($reader->nextPatternMatch($pat)) { ... }
  ```
Why it is fast, why is it slow

- Inside XS, most things are very fast
- Perl -> XS -> Perl transitions and Perl calls are expensive
  - affects SAX as well as DOM traversal or Reader that stops on every element
  - ref($node) eq 'XML::LibXML::Element' is 3x faster than
    $node->nodeType == XML_ELEMENT_NODE
  - DON'T: if ($node->name eq "foo") {... } elsif ($node->name eq "bar") ...
    (use variables)
- XPath over wide DOM or returning many nodes is slow
  - XPath requires nodes to be returned in document order
  - document-order sort is O(C(n) * n^1.2) (Shell's sort)
    where C(n) (complexity of comparing positon of A and B) is not constant!
  - for read-only documents use indexElements() to make C(n)=1
**XPath versus DOM traversal**

Approximating XPath 'child::y' or './y' in DOM

1. `$node->getChildrenByTagNameNS(undef,'y')`
2. `grep $_[0]->nodeName eq 'y', $node->childNodes`
**XPath/DOM Performance**

- a) /x/y

Higher is better
XPath/DOM Performance

- b) /x/y[@bar="baz4"]

Higher is better
Performance benefit of indexElements on XPath

Thicker is better

co=pre-compiled XPath   xp=XPath expression   i=indexed   u=unindexed
That's all

Thank you!