Scales Xml

A High Performance Scala Xml Library

Chris Twiner

Table of contents

1. Scales XML

1.1 0.6.0-M7

verage

Statement	77.11 %	Branch	73.21 %

Scales XML is an alternative XML library for Scala, its design started with the question - *what if the structure of XML was separated from its contents?*.

The answer for XML tends naturally to trees and zippers, enabling a combined model for both XML Tree handling and XML Event handling. This allows opportunities for saving memory usage and increasing performance.

The design aims of Scales Xml also target correctness first, an Iteratee based processing for Pull, an XPath like syntax for querying and manipulation and deep support for JAXP. See the release notes for Iteratee upgrade information.

🆕 Scalaz Iteratee library usage along with a necessary upgrade guide in the release notes

The main focus areas are

- Correctness
- Simplified and consistent model - shared between push and pull
- Allowing full re-use of the data model - QNames, Elems etc all reusable
- Fast Internal XPath Api
- Full XPath1.0 String evaluation support via Jaxen
- Iteratee based xml Pull handling - you choose what, when and how to process
- Non-blocking IO xml Pull processing via Aalto
- High Level of JAXP Integration - Validation, TrAX and Serialisation (when needed)
- Performance - fast with low memory usage

Last update: May 10, 2024 12:48:10
Created: May 10, 2024 12:48:10

2. Getting Started

2.1 Scales Xml Overview

Scales Xml, in addition to immutability, aims to focus on these separate concerns:

Correctness

- XmlItems such as Text, Comments, PIs and CData are not containing nodes, they cannot have children and they cannot contain invalid characters
- QNames are a fundamental piece of XML and are treated as first class citizens
- Elements are a QName, the attributes they contain, the namespaces they declare and also do not directly have children
- Attributes are combinations of either a PrefixedQName or NoNamespaceQName and a value

Structure and manipulation

- Structure of XML (Tree) is separated from the contents of the Xml, allowing maximal re-use
- Operations on XML (Path - a Zipper over Tree) is separated from both the contents and the structure
- Changing any update to individual XML Element or Items will not cause cascading changes in the rest of the structure (function of Path)
- Transformations on XML Trees (via Path) should be composable and simple to follow.
- The data model is the same for both Push and Pull parsing (with the addition of EndElem for XmlPull)

XML Comprehensions - XPath

- Navigation through XML is either through Path directly or the inbuilt XPath syntax
- Syntax closely resembles normal XPath leveraging, for example, Paths ability to navigate parents
- Syntax should separate the navigation of elements from available predicates as far as possible (niceties such as *QName remain)
- Syntax should not rely on string interpretation but on types (QName objects used directly)
- String based XPath 1.0 querying based on Jaxen is however available
- XPaths are used as a basis for Tree transformation
- The full XPath axe are available (with the exception of the namespace axis)

Pull API

- Based on javax.xml.stream, allowing optimisations from jaxp/stax upgrades
- Uses the same types as the SAX based parser
- Leverages Scalaz Iteratees to provide processing of the event stream
- You can choose how to process an xml stream, should your code control the flow, the enumerator or a mixture?
- Immutable transformations on streams
- Provides an Iteratee to process XPath like locations via a ${cscala}List[QName]${cend}
- Provides a combinator (onDone) to iterate over ResumableIters allowing simplified XML parsing
- Non blocking IO based parsing via Aalto
- Scales adds the concept of resumable iteration:

```
// think resumable folds without cps plugin
type ResumableIter[E,A] = IterV[E, (A, IterV[E,_])]
```

Comprehensive Serialisation Support

- SerializerFactory / Serializer Typeclasses allow plugable serialisation schemes
- SerializeableXml Typeclass allows serializing of all XML through a single consistent interface
- Lazily create XML using EphemeralStreams and Iterator[PullType]
- Developers can customise the serialisation process while maintaining correctness
- Use XHTML style tag ends for empty elements
- Serialise your attributes in an order of your choosing
- Or choose to abandon certain correctness checks in favour of speed when you know that your XML inputs and outputs are always correct

JAXP Support

- XSLT transformation support
- Conversion to other XML DOMs
- Use of JAXP serialisation - e.g. pretty printing
- javax.xml.validation support
- Aims to hide all JAXP implementation inconsistencies

Xml Equality

- Compare xml items against each other
- Full control over QName handling
- Full control over QName values (attributes or text)
- Provides information about what is different and where
- Provides a simple conversion to scalaz.Equal

Performance

- Up to 20% faster than scala.xml
- Lower memory usage than both scala.xml and JAXP/DOM
- Parsing memory usage can be optimised for given usage

Last update: May 10, 2024 12:48:10
Created: May 10, 2024 12:48:10

2.2 Setup

Maven users should use scales-xml_2.12 as the dependency (org.scalesxml as the group).

Scales is cross compiled for 2.11 and 2.12.

2.2.1 Sbt 0.11.x +

```
libraryDependencies ++= Seq(
  // just for the core library
  "org.scalesxml" %% "scales-xml" % "0.5.0-M1"
  // and additionally use these for String based XPaths
  "org.scalesxml" %% "scales-jaxen" % "0.5.0-M1" intransitive(),
  "jaxen" % "jaxen" % "1.1.3" intransitive()
  // to use Aalto based parsing
  "org.scalesxml" %% "scales-aalto" % "0.5.0-M1"
)
```

Last update: May 10, 2024 12:48:10

Created: May 10, 2024 12:48:10

2.3 How To Use

To use ScalesXml you must use the following imports (where the objects ScalesUtils and ScalesXml import implicits).

```
import scales.utils._
import ScalesUtils._
import scales.xml._
import ScalesXml._

val prefixedNamespace = Namespace("uri:test").prefixed("pre")
val prefixedQName = prefixedNamespace("prefixed")

val elem = Elem(prefixedQName)

println("local name is "+localName(elem))
```

2.3.1 Parsing and XPaths

```
// Contains the document
val doc = loadXml(new FileReader("document.xml"))

// gets Path from the documents root
val path = top(doc)

// query for all nodes that match the XPath
path.\*("NoNamespace").\*(prefixedQName)
```

Last update: May 10, 2024 12:48:10

Created: May 10, 2024 12:48:10

2.4 Scales Xml Optimisation

2.4.1 Disclaimer

Measuring and comparing XML models is fraught with difficulty and will often vary based on individual XML documents and application behaviour.

2.4.2 Introduction

A basic premise of Scales Xml is to separate the structure of XML from its contents, allowing optimisation on both axes. Due to the Elem also being the child node container scala.xml can only leverage identical subtrees, not individual elements.

The speed of parsing is also directly related to how much garbage is produced during the parse. For example pre 0.1 versions of Scales used Path directly to build the resulting tree, creating many intermediate Path objects - 99.9% of which were immediately ready for garbage collection. Swapping this into a mutable stack of mutable trees improved parsing performance 20-30%.

Scales takes a flexible approach to memory management allowing the user to tune how much work is performed to reduce allocations during the parse and total resulting memory usage.

2.4.3 Options for memory optimisation

Scales Xml's separation of structure and content in particular allows the following areas of memory optimisation:

- QNames
- Attributes
- Elems
- Subtrees
- Structure

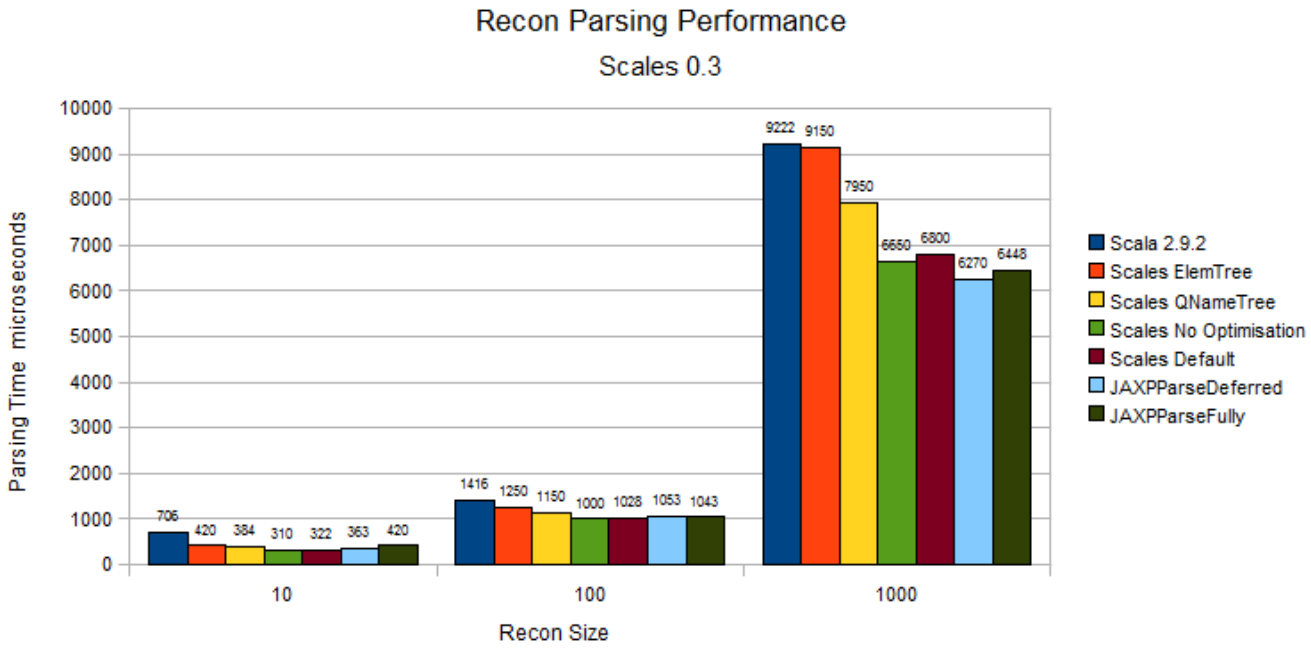
During several optimisation rounds it was clear that the largest wins were due to QName and simple Elem (no attributes or namespaces) usage, and as such forms basis for the default parsing behaviour.

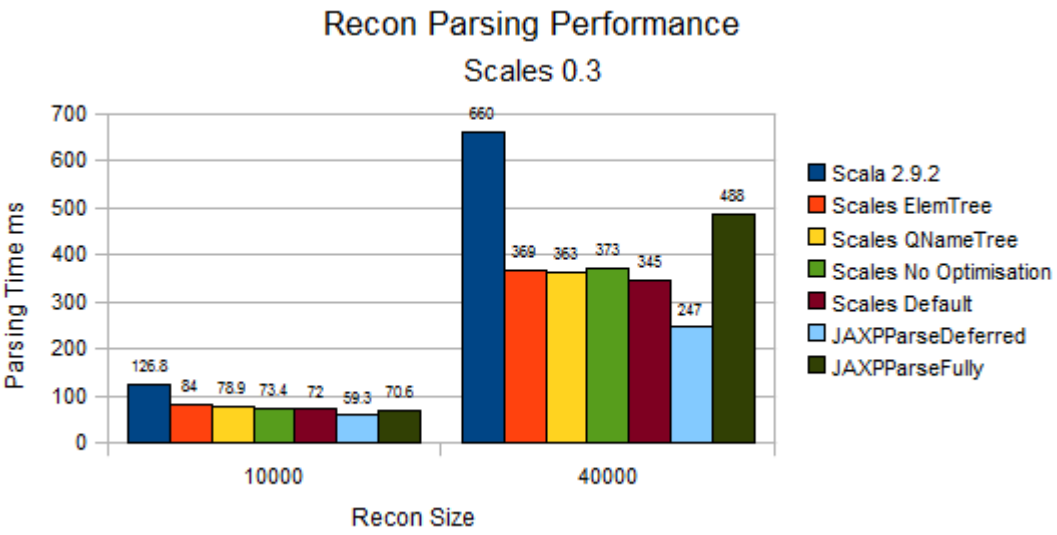
QNames often repeat during individual XML documents and, of course, across them in a given domain. QNames can also be shared between Attributes and Elems. There are two direct benefits of optimising at the QName/ level:

1. Reduced memory consumption
2. Reduced garbage during parsing

However, the time taken to cache QNames is significant for smaller documents (e.g. 150-200 elements) where typically any garbage effects are less likely to impact performance.

The below diagrams illustrate the relative parsing performance of Scales Xml against Scala XML (2.9.1) and the two common Xerces implementations. For all sizes of documents Scales is notably faster than Scala XML, and for larger documents it is considerable faster.

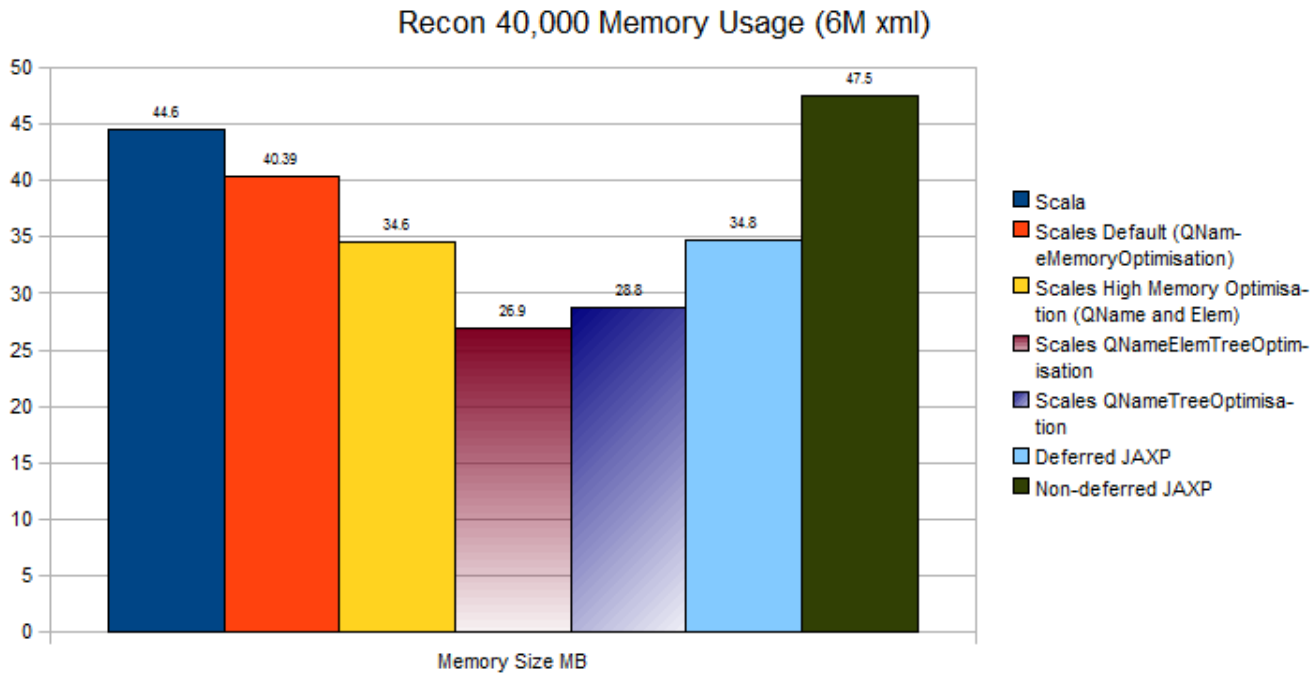




The most important point is that the developer has a choice, when needed, in how to optimise and that Scales allows the developer to easily enhance the default optimisations. The defaults use a thread safe singleton to cache, but developers can also choose to implement a ThreadLocal or a per parse strategy if it better fits. If in doubt profile.

2.4.4 Resulting Sizes

The Recon test (See ParsingPerformance - PerfData.reconDoc) approximates a few real world problems tackled by Scales has very few attributes and a flat structure. The 40,000 Recon size is chosen to demonstrate the resulting memory sizes (obtained by the excellent Yourkit profiler):



The default optimisation (which is also 10-20% faster on larger documents than Scala) strikes a very good balance between speed and memory usage. The Scales parsing optimisations also result in QName and Elem cache sizes of 2.24KB and 2.18KB respectively (the Elem caches are used by HighMemory and QNameElemTree Optimisations), showing more than a clear space saving.

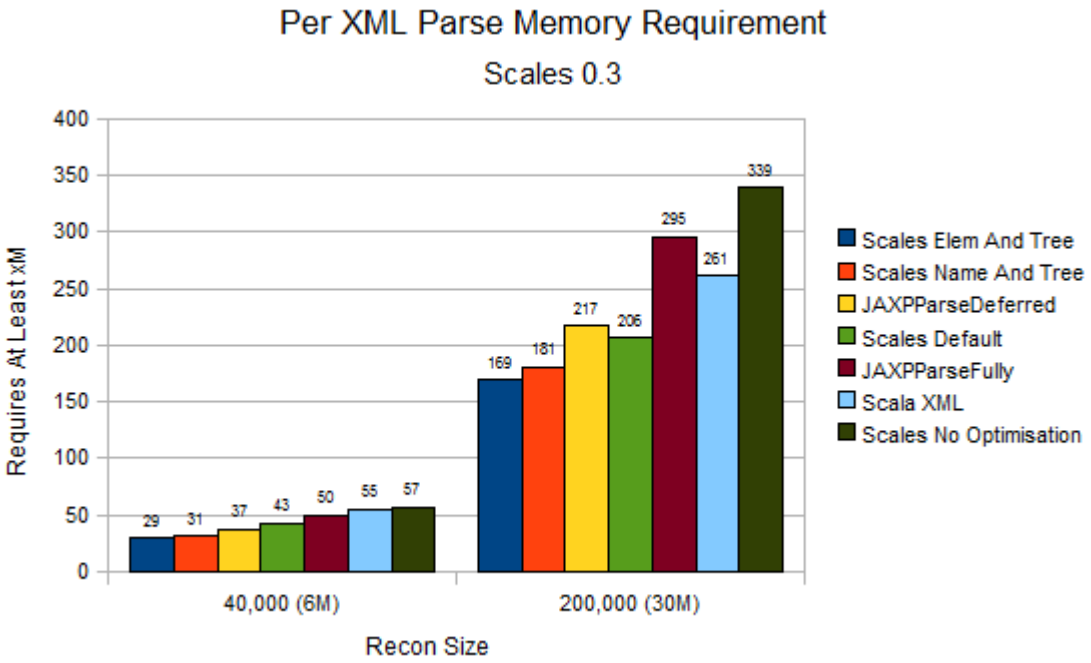
The slightly slower to parse (but still faster than Scala XML) Tree based optimisations reduce immediate memory usage by over 15Mb less than Scala XML for the same document, in fact the two Tree optimisations also consume 5Mb less than the Xerces deferred DOM impl.

Full Elem caching alone (not including Elem + Tree) reduces the overall performance by up to 15%, putting it around (and sometimes below) the parsing performance of Scala XML. The potential memory savings are also limited by how often the attribute or namespace values are identical, and it is often better to simply cache attribute values if there is a restricted range available.

See QName and Elem Memory Usage for more details on default memory saving.

2.4.5 Memory Consumption During Parsing

The memory consumed by Scales in its default configuration is close to or better than Xerces / JAXP default without the nasty expansion upon using it.



However as can be seen above the memory usage can be substantially reduced, in a similar fashion to Xerces Deferred processing (albeit immutably so).

2.4.6 Overall Parsing Performance

The overall parsing performance of Scales Xml, with default Optimisation, is around 20-25% faster than Scala XML for small to mid size documents to 40-45% faster for larger documents.

The larger the amount of data and the level of repeated structure the greater the difference in Scales Xml parsing performance.

Scales is less than 10% slower than JAXP Deferred for smaller documents and for larger documents up to 30% slower. For fully parsed documents however its less than 5% slower for smaller documents and up to 30% faster for larger docs.

2.4.7 Special Case - Pull Parsing via onQNames

A key feature of Scales is the XML Pull (via StAX) based parsing that leverages Scalaz Iteratees. A shining example of this being onQNames, the recon file being a driving force behind its design.

In the recon example what is actually interesting is three simple hash maps (Int -> Int), as such the memory usage of a onQNames -> map was examined. Both the overall resulting savings (over a full tree) and the runtime memory requirements to parse are examined using the following code:

```
object Recon {
  import PerfData.ns

  val Part = List(ns("Recon"), ns("Parts"), ns("Part"))
  val Bom = List(ns("Recon"), ns("BOMs"), ns("BOM"))
  val Rec = List(ns("Recon"), ns("Records"), ns("Record"))

  val id = ns("id")
  val version = ns("version")

}

case class Recon(val parts : Map[Int, Int],
             val boms : Map[Int, Int],
             val recs : Map[Int, Int])


import Recon._
import Functions._

val xml = pullXml(new java.io.StringReader(s)).it
foldOnDone( xml )( Recon(),
```

```
                    onDone( List(onQNames(Part),
                                 onQNames(Bom),
                                 onQNames(Rec)) )) {
      (recon, qNamesMatch) =>
        if (qNamesMatch.size == 0)
          recon
        else {
          // we expect only one to match in this pattern
          val matched = qNamesMatch.head
          val qnames = matched._1  // to get an onDone it must be defined
          val x = matched._2.get
          // only one child
          val pair = (text(x.\*(id)).toInt, text(x.\*(version)).toInt)

          qnames match {
            case Part => recon.copy( parts = recon.parts + pair )
            case Bom => recon.copy( boms = recon.boms + pair )
            case Rec => recon.copy( recs = recon.recs + pair )
          }
        }

    }
```

In short the memory requirements for the maps are 13.7MB, whereas the test itself can run in under 45MB, albeit with a high GC overhead. As such thats roughly 25MB required to actually parse the entire document in such a high level api.

Also worth noting is that it takes 42MB to generate the document

See the [PullParsing.html Pull Parsing], [RepeatedSections.html Pulling Repeated Sections] and the PullTests themselves for examples on how this approach can be best leveraged in your code.

Last update: May 10, 2024 12:48:10

Created: May 10, 2024 12:48:10

2.5 XML Model

2.5.1 QNames

[http://www.w3.org/TR/2009/REC-xml-names-20091208/#NT-NCName The namespaces spec] introduced namespaces into XML, giving the three following possible types of qualified names:

1. Fully Qualified (local name, prefix and namespace)
2. No namespace (local name only xmlns="")
3. Namespaced (local name and namespace - xmlns="a new default scoped namespace")

This can be seen modelled in the JAXP QName class, or as the QName ADT in Scales. The namespaces spec introduced the letter ":" as a separator of prefix and local name, as such its use is forbidden in QName prefixes or localNames (runtime check).

Creating QNames

DIRECTLY

```
val ns = Namespace("uri:namespace")
val pre = ns.prefixed("pre")
val prefixed = PrefixedQName( "local", pre )

// same as above
val prefixed2 = pre("local")

// without a prefix - only for Elems
val namespaced = ns("local")

val noNamespace = NoNamespaceQName("local")

val prefixedDirect = ns.prefixed("pre2","local")
```

IMPLICITS

```
// implicitly make it, allowing for \*("localNameOnly") XPath 1.0 syntax
val localOnly : NoNamespaceQName = "local"

// "directly" declaring it local with .l
val noNamespace = "local"l

val unprefixed = "test:namespace"::"local"
```

Namespaces & Scope

Namespaces are scoped according to the nesting of XML elements. If the root declares a default namespace it applies for all of the XML child elements unless overridden by introducing a new scoped default namespace.

This scoping does not apply to attributes, they are either fully qualified or have namespace xmlns="". Namespaces are in XML 1.0 URIs and in XML 1.1 IRIs.

Namespace prefixes also have a different binding relationship in XML 1.1 documents, where xmlns:pre="" is actually legal (albeit very confusing) and unbinds the "pre" prefix. This makes it invalid to use the "pre" prefix again (taken from the 1.1 namespaces spec):

```
<?xml version="1.1"?>
<x xmlns:n1="http://www.w3.org">
    <n1:a/>              <!-- legal; the prefix n1 is bound to http://www.w3.org -->
    <x xmlns:n1="">
        <n1:a/>          <!-- illegal; the prefix n1 is not bound here -->
        <x xmlns:n1="http://www.w3.org">
            <n1:a/>      <!-- legal; the prefix n1 is bound again -->
        </x>
    </x>
</x>
```

Scales attempts to provide validation of the names via the version, but this is more than a little edge case filled. Other libraries (XOM for example) have chosen to not support XML 1.1s use at all.

See XML Version Support for more details.

Namespaces in Scales

Namespaces are directly modelled as a type in Scales and are used to create QNames. Namespaces are created simply by

```
// a Namespace
val ns = Namespace("test:uri")
// a PrefixedNamespace
val p = ns.prefixed("p")

// this QName is fully qualified
val qname = p("localname")
```

PrefixedNamespaces are required to create a PrefixedQName but can also lead to simpler looking code. In addition, they can be used to declare namespace mappings on a particular level of a tree (attached to Elem - which can reduce re-use), the XML 1.1 example above requires this approach of declaring the mappings.

QNames in Scales - Let the compiler help us

Scales enforces the use of QNames throughout its api. Attributes can '''only''' be created with either a NoNamespaceQName or a PrefixedQName. Elements also can use UnprefixedQNames:

```
  val ns = Namespace("uri:namespace")
  val pre = ns.prefixed("pre")

  val unprefixedQName = ns("localName")
  val prefixedQName = pre("localName")
  // l is a NoNamespaceQName wrapper
  val nonamespaceQName = "localName"l

  val uelem = Elem(unprefixedQName)

  // implicitly converted, nonamespaceQName is valid for an attribute
  val nonamespaceAQN : AttributeQName =
    nonamespaceQName

  // won't compile as unprefixed but namespaced QNames are not valid for an attribute
  //val unprefixedAQN  : AttributeQName =
  //  unprefixedQName

  val root =
    <(uelem) /@(nonamespaceAQN -> "nv",
                prefixedQName -> "pv") /(
      prefixedQName, // these implicitly create an XmlTree with this QName
      nonamespaceQName
    )
```

Runtime Validation

Whilst the compiler can help us with correctness on types, we sacrifice a usable api if we force other content checks into the type system. For example we could model QName.apply functions to return an Either (or Validation) and force combination of all Elem related data, but this would distract from the api (especially considering that incorrect XML is only likely to come from the developer - the parser won't let it through otherwise).

Scales QNames and Namespaces check for correct local name and prefix content at runtime based on the compile time scoped XmlVersion. This still implies no object will ever get created with incorrect data, but forces these checks into throwing exceptions (or make the user suffer with the api that follows).

Valid characters for a given xml version are checked by Xerces XML Char utilities with the addition of a simple ":" check from Scales. Developers may not create a Namespace with "" unless using Xml11. Declaring PrefixedNamespaces with prefix "xmlns" or "xml" must match their predefined uris.

NB: I've not given up on attempting this completely

Equality

Namespaces are equal only when their uri is equal, but PrefixedNamespaces also take the prefix into account.

QNames use both namespace + local name to test for equality:

```
// using the above definitions
noNamespace == unprefixed // false

val unprefixed2 = "fred:uri" :: "local"

unprefixed2 == unprefixed // false

// =:= is a seperate method that acts as == but is typed on QNames
prefixed =:= prefixed2 // true

val prefixedl2 = pre("local2")

prefixed =:= prefixedl2 // false

prefixedDirect == prefixed // true
```

For PrefixedQNames its also possible to take the prefix itself into consideration (although not recommended):

```
prefixedDirect =:= prefixed // true

prefixedDirect === prefixed // false
```

SCALAZ EQUAL AND SCALES EQUIV

A Scalaz Equal typeclass is defined allowing (but shadowed by QName for its default):

```
import scalaz._
import Scalaz._

implicitly[Equal[QName]].equal(prefixedDirect, prefixed) // true
```

Equiv is a helper class from Derek Williams that allows defining conversions to types that have a provided Scalaz Equal type class instance. This is used by ListSet to allow removal of Attributes by their QName (via =:=).

```
import scalaz._
import Scalaz._

// the below is provided by the scales.utils.equivalent
implicitly[Equiv[QName]].apply(prefixedDirect, prefixed) // true

equivalent( prefixedDirect, prefixed ) // true

// PrefixedQName and NoNamespaceQName can be "converted" to QName
equivalent( prefixedDirect, localOnly) // false
```

Testing For QNames

Equality and Extractors can cover most usage scenarios but sometimes you want to test and extract in one go - just like Scala's regex support.

As part of the Xml DSL Scales provides both QName and Namespace Matchers. Starting with the following definitions we'll look at how to use these Matchers:

```
val ns = Namespace("uri:namespace")
val name = ns.prefixed("pre","localName")

val elem = Elem(name)
val attrib = Attribute(name, "val")
```

QNameMatcher allows you to test for whether an Attribute or an Elem is defined by a given QName (via =:=, namespace and local name only are compared):

```
val NamedMatcher = name.matcher // .m is a short cut

elem match {
  case NamedMatcher(e) => println("Matched element " + asString(e))
  case _ => error("oops")
}

attrib match {
  case NamedMatcher(a) => println("Matched attribute " + a)
  case _ => error("oops")
}
```

NamespaceMatcher simply checks for the Attribute or an Elem being part of a Namespace:

```
val NamespaceMatcher = ns.m // short for matcher

elem match {
  case NamespaceMatcher(e) => println("Matched element " + asString(e))
  case _ => error("oops")
}

attrib match {
  case NamespaceMatcher(a) => println("Matched attribute " + a)
  case _ => error("oops")
}
```

Serializing QNames

QNames for elements and attributes are considered markup, and depending on both the encoding and Xml Version will throw an InvalidCharacterInMarkup exception with the relevant QName part that caused the issue.

Last update: May 10, 2024 12:48:10

Created: May 10, 2024 12:48:10

2.5.2 Elem

XML Elements

Elements in XML are markup that scopes other parts of the document, typically this scoping is seen as a containment relationship in XML object models. Classical DOM follows this approach as does Scala XML.

Declaring

```
val ns = Namespace("uri:namespace")
val pre = ns.prefixed("pre")
val name = pre("localName")

val attr = pre("attribute")

val elem = Elem(name,
  Attribs("a" -> "non namespaced",
    attr -> "prefixed attribute"))
```

See attributes for an explanation of how -> works.

QName And Namespace Correctness

Elements also contain both declarations of namespaces, introducing new prefixes or default namespaces, and Attributes. Elements are declared with a given QName, which can also be a non-prefixed but namespaced.

```
<!-- this element name has no prefix but a namespace -->
<elem xmlns="uri:test">
  <!-- this element name also has no prefix but a namespace -->
  <child>
    <!-- this element name has no namespace and declares
         that all child elements, by default, have no namespace -->
    <grandchild xmlns="">
    .....
  <!-- this element contains an attribute (without a namespace)
       but has its name with a namespace -->
  <child attribute="value">
  ....
  <!-- this element declares a new
       namespace and prefix mapping. -->
  <child xmlns:pre="uri:anotherNamespace">
    <!-- this elements name is prefixed and within a namespace
         but the element declares a default namespace. -->
    <pre:grandchild xmlns="">
```

As can be seen there are a number of complex combinations with namespaces and attributes that are possible and that the attribute mechanism itself is used to declare namespaces.

When supporting XML 1.0 only it can also be seen that storing the namespaces declared on an element is not necessary if all the QNames are typed. XML 1.1, however, allows the removal of a namespace prefix definition and requires that this information is stored. If nothing else its quite helpful to be able to define where a namespace prefix declaration should appear.

Scales, however, does not conflate the ideas of namespace declaration, attributes or the qualified names of elements.

Elems Are Reusable

Scales Elem contains the QName, Attributes and optionally prefix declarations. They do not have any notion of containment and as such are re-usable not only within a document but across documents. Servers processing high volumes of related data can of course benefit from the reduced allocation costs, but the code can also benefit.

Declaring an Elem once and simply including it within a tree definition is not only made possible but encouraged:

```
val unprefixedQName = "uri:namespace" :: "localName"
val elem = Elem(unprefixedQName)

val root = <(elem) /(
  elem, elem, elem,
  elem /(
    elem
  )
)
```

```
asString(root)
```

gives (''formatting added to match root's above definition''):

```
<?xml version="1.0" encoding="UTF-8"?>
<localName xmlns="uri:namespace">
  <localName/><localName/><localName/>
  <localName>
    <localName/>
  </localName>
</localName>
```

Runtime Validation Checks

As Elem is created with a QName it is subject to all of QNames correctness and runtime checks.

In addition, Scales enforces that you cannot create an Elem with a prefix of xmlns or xml.

Last update: May 10, 2024 12:48:10
Created: May 10, 2024 12:48:10

2.5.3 Attributes

XML Elements can contain attributes, these are made of a pair: QName and a string value. Unlike Elements the QName of an Attribute must be either a fully qualified (PrefixedQName) or unqualified name (NoNamespaceQName).

Defining an Attribute

EXPLICITLY

```
val ns = Namespace("uri:namespace")
val pre = ns.prefixed("pre")

val prefixedQName = pre("localName")
val nonamespaceQName = "localName"l
val unprefixedQName = ns("localName")

val nonamespaceAttr = Attribute(nonamespaceQName, "value")
val prefixedAttr = Attribute(prefixedQName, "value")

// won't compile as Attributes can't have a namespace without a prefix
//val unprefixedAttr = Attribute(unprefixedQName, "value")
```

IMPLICITLY

```
// This can be used when defining elements or within the dsl
val attr : Attribute = nonamespaceQName -> "value"

// won't compile
// val noAttr : Attribute = unprefixedQName -> "value"
```

Equality

The XML specifications require that no two attributes that share namespace and localName may be in the same element. Attributes however must be testable for equality outside of this constraint:

```
val attr2 : Attribute = nonamespaceQName -> "another value"

attr.name == attr2.name // true

attr == attr2 // false, values are different

val attr3 : Attribute = nonamespaceQName -> "another value"

attr3 == attr2 // true
```

WITHIN AN ELEM

Scalaz Equal comes to the rescue again, we can separate the notion of attribute equality for simple comparisons and those of an Elem's requirements:

```
attr3 === attr // true - we don't take the value into account

attr3 == attr // false - values are compared
```

ATTRIBUTES LISTSET

The Scales Utils class ListSet compares using the notion of Equiv equivalence (see Equiv and Equal for more details). To make things easier for developers not using the DSL, Scales adds the Attribs() constructor (which ensures the appropriate Equal and Equiv type classes are always present):

```
val attributes = Attribs(attr, prefixedQName -> "yet another value")
```

Testing Against QNames or Namespaces

QNameMatcher and Namespace matcher provide simple matching logic that can simplify certain types of pattern matches, just like Scala Regex - see Testing For QNames for examples.

Last update: May 10, 2024 12:48:10
Created: May 10, 2024 12:48:10

2.5.4 XmlItem

Scales represents non Elem data as XmlItem, this includes Text, CData, PI (processing instructions) and Comment. They share the base trait XmlItem which provides one member:

```
val value : String
```

With PI also adding:

```
val target : String
```

Declaring

```
val text = Text("A text value")
val cdata = CData("Some cdata")
val comment = Comment("A comment")
val pi = PI("Target id","instruction value")
```

XmlItems Are Reusable

XmlItems, as with the rest of Scales - having separated data from structure, have no notion of ownership.

This implies they can be re-used both within and across documents. This follows with Elems and Trees themselves allowing whole sections of XML to be re-used.

Runtime Correctness Checks

There are a number of simple rules for XmlItems, driven by the spec:

1. Comments cannot contain the text "–"
2. CData cannot contain "]]>"
3. PIs cannot contain "?>" in the value or target
4. PIs target when lower cased cannot start with "xml"

Serializing XmlItems

Text, Comments and PI may contain character references and can be correctly serialized regardless of the encoding or Xml Version used.

SERIALIZING CDATA

CData is problematic to serialise due to:

- JAXP differences (Xerces/Xalan, Saxon and Sun jdk) all behave differently with regards to CData serialisation
- CData itself can be encoded but not < and &, doing so changes the CData.
- CData, similar to QNames, cannot be encoded via character references

The problems here should warn the user not use CData at all, as per ERH it does not add anything. It is only provided in Scales to allow documents to be passed through as is (e.g. content based routing).

Scales does its best to work around such issues including, forbidding CData splits, custom serializing (jre will write no end part to cdata) and verifying that the data can be written at all in the documents encoding.

Last update: May 10, 2024 12:48:10
Created: May 10, 2024 12:48:10

2.5.5 Xml DSL and Trees

XML DOMs traditionally create trees by each element "owning" its children. Scales eschews this model, instead using trees to model the containment, allowing re-use for all of the Scales model.

To simplify both creating and basic manipulation of trees Scales provides a DSL that closely resembles the structure of XML. The point of focus in the tree (or the depth) is controlled by nesting of the arguments. The implementing class is [./doc/scales/xml/dsl/DslBuilder.html DslBuilder]

The following example is a quick introduction into how to create trees (also used within the XPath guide):

```
val ns = Namespace("test:uri")
val nsa = Namespace("test:uri:attribs")
val nsp = nsa.prefixed("pre")

val builder =
  ns("Elem") /@ (nsa("pre", "attr1") -> "val1",
                 "attr2" -> "val2",
                 nsp("attr3") -> "val3") /(
    ns("Child"),
    "Mixed Content",
    ns("Child2") /( ns("Subchild") ~> "text" )
  )
```

Tour of the DSL

The tour will use the following definitions:

```
val ns = Namespace("uri:test")

val elem = ns("Elem")
val child = ns("Child")
val childl = "Child"l
val root = ns("Root")
val child1 = ns("Child1")
val child2 = ns("Child2")
val child3 = ns("Child3")
val fred = ns("fred")
```

Creating a Tree

To start a tree simply use a qname or elem followed by a DSL operator:

```
val dsl = elem / child

// <Elem xmlns="uri:test"><Child/></Elem>
asString(dsl)
```

or for visual distinction use the <( ) function

```
val dsl2 = <(elem) / child
```

Adding To The Tree

To add a subelement use:

```
// <Elem xmlns="uri:test"><Child/><Child2/><Child3/></Elem>
val dsl3 = dsl2 /( child2, child3)
```

The tree can be freely nested and, instead of a sequence of subtrees, a by-name version with taking an Iterable allows you to call other functions to provide the sub-trees.

Adding an Attribute

```
// <Elem xmlns="uri:test" attr="fred"><Child/><Child2/><Child3/></Elem>
val dsl4 = dsl3 /@( "attr" -> "fred" )
```

Setting Text

As we often set a single string for a given subtree the DSL provides a helpful feature to replace all child nodes with a single text node.

```
// <Elem xmlns="uri:test" attr="fred">a string</Elem>
val dsl5 = dsl4 ~> "a string"
```

This can, of course, be nested:

```
// res14: String = <Elem xmlns="uri:test" attr="fred"><Child/>
// <Child2/><Child3/><fred>fred's text</fred></Elem>
val dsl6 = dsl4 /( fred ~> "fred's text" )
```

Removing Children

Any child whose QName matches (namespace and localname =:=) will be removed via:

```
// <Elem xmlns="uri:test" attr="fred"><Child2/><Child3/><fred>fred's text</fred><Child xmlns=""/></Elem>
val dsl7 = (dsl6 -/ child) / childl
```

Note that due to infix limitations of Scala that the following won't compile:

```
// won't compile as its an even number of terms
val dsl7 = dsl6 -/ child / childl
```

If in doubt use brackets or dot accessors to be specific.

Removing Attributes

Similar to removing Elems QName and a - do the job:

```
// <Elem xmlns="uri:test"><Child2/><Child3/><fred>fred's text</fred><Child xmlns=""/></Elem>
val dsl8 = dsl7 -/@ "attr"
```

Optional XML

When using the DSL to template XML its often necessary to model optional content. This can be expressed via direct use of Option for an attribute, child or text directly:

```
def template(optionalText : Option[String) =
  ns("Elem") ~> optionalText

val hasTextChild = template(Some("text"))
val hasNoChildren = template(None)
```

As hasNoChildren demonstrates, when using None, there will be no child added. Using "" instead, would add an empty Child, whilst semantically identical upon serialization it changes the meaning of the in-memory DOM itself.

The Attribute and child variants function in the same way, but for some usages a fully cascading solution may be required. If the elem itself should also not be added when there are no children present then the [OptionalDsl.html Optional DSL] should be used.

Folding Within The DSL

The XPath fold facility is also present directly from the DSL, letting you stay within the tree building and manipulate at the same time with the full power of the DSL.

The DSL provides three fold functions, fold - returning an Either (as per the normal fold but wrapped with DslBuilder), fold_! (throws upon an error) and fold_?.

The fold_? function is perhaps the most commonly useful, and returns "this" if no folds took place (NoPaths), but throws if an error was found.

In each case the parameters are XmlPath => XPath (to allow selection) and the folder (what should we do with the selection) and is used thusly:

```
// remove all ChildX elements regardless of namespace yielding:
// <Elem xmlns="uri:test"><fred>fred's text</fred></Elem>
val dsl9 = dsl8.fold_?( _.\*( x => localName(x).startsWith("Child"))) {
  p => Remove()
}
```

See XPath Folds for more information and how to use it for transformations.

Last update: May 10, 2024 12:48:10

Created: May 10, 2024 12:48:10

2.5.6 Optional Xml DSL

The [XmlDsl.html Xml DSL] provides a simple way to create XML trees via a flexible builder approach. The Optional Xml DSL provides a similar approach to fully optionl trees, where no empty nodes should be present.

Given the following xml:

```
<root xmlns="uri:test">
  <optional>some possible text</optional>
</root>
```

we may like to remove optional if the "some possible text" was not defined (for example against a minoccurs 0 element).

The Optional DSL fully integrates within the Xml DSL itself, allowing easily defined optional subtrees, but uses a different notation to let you know at a glance you are dealing with the Optional DSL:

```
val ns = Namespace("uri:test")

val root = ns("root")
val optional = ns("optional")

def someOptionalText: Option[String] = ???

// optional is converted into an instance of OptionalDslBuilder allowing ?~> to be called
val optionalxml = root /( optional ?~> someOptionalText )
```

If someOptionalText returns Some("a value") optionalxml will serialize to

```
<?xml version="1.0" encoding="UTF-8"?><root xmlns="uri:test"><optional>a value</optional></root>
```

however returning None will collapse the optional element as well as the text (this is in contrast to ~> Option[String] in the normal Xml DSL which leaves the optional element present):

```
<?xml version="1.0" encoding="UTF-8"?><root xmlns="uri:test"/>
```

The api for the Optional DSL can be found here.

Cascading Optionals

The Optional DSL cascades so the following xml:

```
val deepNones =
  ?<("Alocal"l).?/(
    ?<("another"l) ?/(
      ("lowerstill"l) ?~> None ),
    ?<("yetan"l) ?~> None
  ).addNonEmpty(("ShouldNotGetAdded"l))

val result = deepNones.toOptionalTree
```

will have result.isEmpty == true.

Serializing OptionalDsls doesn't make much sense without at least one outer wrapping node, as such no SerializeableXml type class instance is provided.

The addNonEmpty method above allows adding child nodes directly, but filters out any empty trees (no child nodes or attributes). It does not, however, perform deep cascading, so prefer OptionalDslBuilder instances where possible.

The "?<" constructor is the Optional DSL counterpart to the Xml DSLs "<" and acts a visual marker.

Last update: May 10, 2024 12:48:10

Created: May 10, 2024 12:48:10

2.5.7 Scales XML Version Support

There are two versions of XML, 1.0 and 1.1, with 1.0 being most commonly used.

1.1 parsers can parse 1.0 xml, but not the other way around, which adds for another source of potential differences in user experience.

Differences Between 1.0 and 1.1

There are four main differences:

1. Unicode markup - elements can have names in Kanji
2. Certain whitespace is (dis)allowed
3. Namespaces can be "unbound" from their prefixes
4. Namespaces are IRIs not URIs

These differences make 1.1 support difficult to pleasantly achieve and, in order to keep a uniform interface, forces some correctness checks into the runtime.

How Does Scales Allow Both Versions ?

Scales takes a pragmatic approach to supporting both versions via using implicits, with the defaultVersion being 1.0. Override the implicit to provide scoped XML 1.1 support:

```
// scope here is Xml10
{
  implicit val defaultVersion = Xml11 // scope here is Xml11
  val ns = Namespace("http://www.w3.org")
  val pre = ns.prefixed("n1")
  val disabled = Elem("x"l, Namespace("").prefixed("n1"))
  val a = Elem(pre("a"))
  val x = Elem("x"l, pre)
}
```

All of the above example namespaces, local names and prefixes are then validated for correctness at runtime. The above example with Xml10 would throw as "" is not a valid namespace for XML 1.0 (and is only used in XML 1.1 for unbinding namespace prefixes).

IN PARSER WE TRUST - USERS WE PROTECT

Runtime checks for correctness are redundant if the model is created via a parser. The parser already checks the validity (users may choose to override the settings at the factory level).

However, for general usage it's helpful for the library to safeguard us against mistakes. Scales emphasis on correctness continues here as well.

The implicit FromParser is used by Scales to provide this runtime check, and the QNameCharUtils functions provide the helpers for a given version (via Xerces XMLChar and XML11Char.

Runtime XmlVersion QName Related Correctness

The implicitly scoped version dictates what runtime checks are applied. In particular the namespace, local name and prefix are all validated at runtime for their content. This is true for both Attribute and Elem QNames.

Serializing is also, of course, affected by the XML version. The document serialisation itself then requires checking the compatibility of QNames. The rule here is simply if it's an Xml11 QName only (all parts are correct) and the document version is Xml10 its an error.

Last update: May 10, 2024 12:48:10
Created: May 10, 2024 12:48:10

3. Parsing XML

3.1 Full XML Doc Parsing

Parsing a full XML document in Scales can be very straightforward:

```
val doc = loadXml(new FileReader("document.xml"))
```

The input to loadXml is an InputSource, a PathOptimisationStrategy and a Loaner[SAXParser]. Defaults are provided for the strategy and Loaner, but can be overridden.

Conversions exist (pulled in via ScalesXml._) from InputStream, Readers and URL to ease the use of the api.

PathOptimisationStrategys allow the developer to tweak both the memory consumption and generation (and therefore the performance). The default optimisation caches QNames across an application but does not attempt to cache elements or attributes. Caching elements and attributes can lead to significant memory savings at the cost of parsing performance.

As the names suggests PathOptimisationStrategies could also choose to optimise whole sub-trees, tests have not shown a general case where this is beneficial however (the cost of matching the tree typically outweighing potential memory savings).

Loaner is a simple interface to obtain SAXParser instances, other non default instances can be provided, such as JTagSoup or simply to allow customisations of SAX properties. The default SAX parser pool also takes care of common threading issues.

3.1.1 Direct SAX XMLReader Usage

As of Scales 0.3 there is direct support for SAX parsers via the loadXmlReader function. This follows the same Loaner approach as the normal JAXP factories, for example:

```
import org.xml.sax.XMLReader

object NuValidatorFactoryPool extends scales.utils.SimpleUnboundedPool[XMLReader] with DefaultSaxSupport {

  def create = {
    import nu.validator.htmlparser.{sax,common}
    import sax.HtmlParser
    import common.XmlViolationPolicy

    val reader = new HtmlParser
    reader.setXmlPolicy(XmlViolationPolicy.ALLOW)
    reader.setXmlnsPolicy(XmlViolationPolicy.ALLOW)
    reader
  }

}

val xmlFile = resource(this, "/data/html.xml")
val nuxml = loadXmlReader(xmlFile, parsers = NuValidatorFactoryPool)
```

Developers can also call readXml directly with a given XMLReader instance instead of using the pooled version.

If a given XMLReader cannot work with the DefaultSaxSupport it can override the appropriate functions. For example TagSoup doesn't support XmlVersion information, as such a TagSoupFactoryPool would look like:

```
object TagSoupFactoryPool extends scales.utils.SimpleUnboundedPool[XMLReader] with DefaultSaxSupport {

  // doesn't support xml version retrieval
  override def getXmlVersion( reader : XMLReader ) : AnyRef =
    null

  def create = {
    import org.ccil.cowan.tagsoup.Parser
    val reader = new Parser
    // disable namespaces
    reader.setFeature(Parser.namespacesFeature, false)
    reader
  }

}
```

Last update: May 10, 2024 12:48:10
Created: May 10, 2024 12:48:10

3.2 Pull Parsing

Scales Pull Parsing leverages StAX, the JAXP streaming api, and Scalaz Iteratees, to allow flexible parsing of large documents or many documents in memory constrained environment (e.g. a high performance server).

Scales, as with full tree parsing, allows configurable optimisation strategies and the pull parser used. The optimisation strategy type, unlike full tree parsing is a MemoryOptimisationStrategy and does not allow for tree path optimisations.

The input to all pull xml functions is a sax.InputSource, which allows the same conversions as for a full tree parse.

A curio exists with pullXmlCompletely, which uses the pull parser to load xml Docs. This may be of use in an environment where the StAX parser performs better than SAX, but tests have shown lower memory consumption and higher performance when using SAX to parse full trees.

Some of the advanced features of Pull Parsing may require importing Scalaz as well:

```
import scalaz._
import Scalaz._
import iteratee._ // may not always be required
```

3.2.1 Pull Model

The Scales Pull Model adds only EndElem to create PullType:

```
type PullType = Either[XmlEvent, EndElem]
```

Where XmlEvent is exactly the same Elem, XmlItem model as with full Trees. This is possible because Scales separates the structure of data and the data itself.

The developer only has to learn one single difference to be able to use pull parsing. For example the code to process a stream is simply:

```
val pull = pullXml(source)

while( pull.hasNext ){
  pull.next match {
    case Left( i : XmlItem ) =>
      // do something with an XmlItem
    case Left( e : Elem ) =>
      // do something with start of a new element
    case Right(endElem) =>
      // do something with the end of an element
  }
}
```

3.2.2 Resource Management

The above example has a serious potential flaw, if anything in the while loop throws the resource cannot be closed. To allow greater control of the resource Scales provides the following interface (full api details present [./doc/scales/utils/resources/CloseOnNeed.html here]):

```
trait CloseOnNeed {
  def ++ (close2: CloseOnNeed): CloseOnNeed
  def closeResource : Unit
}
```

Importantly closeResource only closes a resource once. This resource is directly available when calling pullXmlResource.

pullXml itself is also a Closable and provides the same guarantee, close only attempts to close the resource once.

Both results provide the isClosed function (via the IsClosed interface) allowing code to trust that it has been closed. (NB - a future version may choose to expose this in the type system, but integrating with the ARM library makes more sense).

What CloseOnNeed also adds is the ++ function, which combines one CloseOnNeed with another to create a new CloseOnNeed that closes the other two resources. This allows chaining of xml files (via pull iterators).

3.2.3 Simple Reading Of Repeated Sections

Given an xml document with the following format:

```
  <root>
    <nested>
      <ofInterest> <!-- Collect all of these -->
        <lotsOfInterestingSubTree>
        </lotsOfInterestingSubTree>
      </ofInterest>
..
    </nested>
....
  </root>
```

where the interesting parts are always repeating in the same location, we can model the interesting parts a simple List of QNames (very simplified XPath):

```
  val pull = pullXml(new java.io.InputStream(""))

  val qnames = List("root"l, "nested"l, "ofInterest"l)

  // only returns /root/nested/ofInterest paths
  val itr : Iterator[XmlPath] = iterate(qnames, pull)
```

The resulting Iterator contains paths with single child parents up to the root and all of the subtree of interest.

For more complex repeated sections see here for examples.

3.2.4 Buffering And Identifying Xml Messages

When parsing xml messages it is often necessary to identify the type of the message before further processing, for example what kind of soap request is being sent, or what is the root element?

To help with this issue Scales pull parsing offers the ability to "peek" into an event stream and replay the events again to fully process them.

A simple example is processing soap messages based on the first body element, you may want to choose different code paths based on this, but require elements in the header to do so. The usage is simple via the capture function and the skip/skipv functions:

```
  val xmlpull = // stream capture

  val captured = capture(xmlpull)

  // either the path or None if its EOF or no longer possible
  val identified = skip(List(2, 1))(captured) run

  val processor = identified.map(........

  // restart the stream from scratch
  processor.process(captured.restart)
```

The result from skip is simply Option[XmlPath], if the stream runs out or its no longer possible to get that position it is None. Only as much of the stream is read as needed, it will stop on the Left(Elem) event.

NB to only identify the first element, simply use skip(Nil) instead (or skipv()).

Last update: May 10, 2024 12:48:10

Created: May 10, 2024 12:48:10

3.3 Pulling Repeated Sections

Scales leverages and extends Scalaz Iteratees to allow resuming an Iteratee. This resuming action is simply returning the current value and the next continuation when done (ResumableIter). The iterate function, as shown here, uses this approach to provide a single path repeating section.

Many documents however have a more complex structure, of many repeated or alternating structures, the following shows the various structures supported by the combination of onDone and onQNames:

3.3.1 Supported Repeating Section Examples

It's far easier to discuss the solution with a few examples of the problem:

Alternating and Repeating Elements

```
  <root>
    <nested>
      <ofInterest> <!-- Collect all of these -->
        <lotsOfInterestingSubTree>
        </lotsOfInterestingSubTree>
      </ofInterest>
      <alsoOfInterest> <!-- Collect all of these -->
        just some text
      </alsoOfInterest>
    </nested>
...
    <nested>
....
  </root>
```

It should be noted that monadic serial composition of onQNames would also work here, onDone is not absolutely necessary, although as we will see it is more general..

Grouped Repeating

```
  <root>
    <nested>
      <ofInterest> <!-- Collect all of these -->
        <lotsOfInterestingSubTree>
        </lotsOfInterestingSubTree>
      </ofInterest>
    </nested>
...
    <nested>
      <alsoOfInterest> <!-- Collect all of these -->
        just some text
      </alsoOfInterest>
    </nested>
....
  </root>
```

Repeating Nested

```
  <root>
    <nested>
      <ofInterest> <!-- Collect all of these -->
        <lotsOfInterestingSubTree>
          <smallKeyValues> <!-- Collect all of these -->
            <key>toLock</key>
            <value>fred</value>
          </smallKeyValues>
        </lotsOfInterestingSubTree>
      </ofInterest>
    </nested>
...
    <nested>
....
  </root>
```

Sectioned Grouped Repeating

```
  <root>
    <section>
      <!-- Necessary for processing the below events -->
      <sectionHeader>header 1</sectionHeader>

      <ofInterest> <!-- Collect all of these -->
        <lotsOfInterestingSubTree>
          <value>1</value>
        </lotsOfInterestingSubTree>
      </ofInterest>
      <ofInterest> <!-- Collect all of these -->
        <lotsOfInterestingSubTree>
          <value>2</value>
        </lotsOfInterestingSubTree>
      </ofInterest>
      <ofInterest> <!-- Collect all of these -->
        <lotsOfInterestingSubTree>
          <value>3</value>
        </lotsOfInterestingSubTree>
      </ofInterest>
    </sectionHeader>
...
    <sectionHeader>
      <!-- Necessary for processing the below events -->
      <sectionHeader>header 2</sectionHeader>
....
  </root>
```

3.3.2 Pull Parsing ResumableIter'atees

ResumableIter is an Iteratee over E that instead of returning just a Done[R] returns Done[(R, NextResumableIter)]. The next ResumableIter stores the calculation up until the point of returning, allowing the calculation to be resumed.

To process the above examples we make use of this and the onDone Iteratee. This takes a list of ResumableIter and applies the input element to each of the Iteratees in that list, Done here returns both a list of the Iteratees which evaluate to Done for that input and (of course) the next continuation of onDone.

A simple, and recommended, way to leverage onDone is with the foldOnDone function:

```
  val Headers = List("root"l,"section"l,"sectionHeader"l)
  val OfInterest = List("root"l,"section"l,"ofInterest"l)

  val ofInterestOnDone = onDone(List(onQNames(Headers), onQNames(OfInterest)))

  val total = foldOnDone(xml)( (0, 0), ofInterestOnDone ){
    (t, qnamesMatch) =>
    if (qnamesMatch.size == 0) {
      t // no matches
    } else {
      // only one at a time possible for xml matches (unless multiple identical onQNames are passed to onDone).
      assertEquals(1, qnamesMatch.size)
      val head = qnamesMatch.head
      assertTrue("Should have been defined",head._2.isDefined)

      // we should never have more than one child in the parent
      // and thats us
      assertEquals(1, head._2.get.zipUp.children.size)

      val i = text(head._2.get).toInt
      // onQNames always returns the list as well as the XmlPath to allow matching against the input.
      if (head._1 eq Headers) {
        assertEquals(t._1, t._2)
        // get new section
        (i, 1)
      } else (t._1, i)
    }
  }

  assertEquals(total._1, total._2)
```

Last update: May 10, 2024 12:48:10

Created: May 10, 2024 12:48:10

3.4 Async Pull

Traditional push models (DOM/SAX parsing) and the StAX pull standard both block on streams when they need more data. The [FasterXML/aato-xml Aalto XML] project aims to change that.

Instead of blocking Aalto XML enhances the StAX parser adding an event type EVENT_INCOMPLETE to signal that no more data could be read. Scales Xml adds the AsyncParser to wrap this behaviour (and a number of supporting io classes) providing a single function to capture the interaction:

```
def nextInput(d: DataChunk): Input[EphemeralStream[PullType]]
```

where the [./doc/scales/utils/io/DataChunk.html DataChunk] ADT is either EOFData, EmptyData or an array of Byte. The resulting Scalaz IterV.Input provides the mirror of either EOF, Empty or El for a given stream of PullType.

This allows a chunked parsing approach, the developer can fully control the use of the resulting xml. Calling nextInput with a filled DataChunk won't necessarily return an El if not enough bytes were "pushed" into nextInput. The resulting EphemeraStream can only be reliably traversed once - and is used as a safer memory usage stream only.

An example direct usage of this api:

```
import scales.utils.io._

var channel: java.nio.channels.ReadableByteChannel = ??? // a channel
var wrappedChannel: DataChunker[DataChunk] = channel.wrapped
var b: DataChunk = EmptyData

while(b != EOFData) { // real code could return thread to a pool with another thread selecting on multiple channels
  b = wrappedChannel.nextChunk
  val s = parser.nextInput(b)
  s(
    el = e => { // use stream of PullTypes
      var st = e
      while(!st.isEmpty) {
        val pullType = st.head()
        // use pullType
        st = st.tail()
      }
    },
    empty = ,// needs more data
    eof = // xml message is now finished - no more events possible
  )
}
```

The '''wrapped''' method is an implicit converter that lifts a java.nio channel.ReadableByteChannel into a DataChunker[DataChunk]. This interface provides nextChunk and a CloseOnNeed with the redundant type parameter allowing an Enumerator to be created over DataChunker (Enumerators can only enumerate over a shape F[_]).

3.4.1 Integrating With Enumeratees - enumToMany

ResumableIter'atees within Scales allow calculations to be suspended and resumed with intermediate results which is very useful for streamed XML processing. When a new value is available a Done((value, cont)) is returned and when more data is required a Cont. The same standard Iteratee semantics (Done or Cont) can be used to "map" over an Iteratee to convert one sequence of typed events into sequences of other types. This mapping process is modelled by Enumeratees.

The key Enumeratee provided by Scales is enumToMany, a mapping Enumeratee:

```
def enumToMany[E, A, R]( dest: ResumableIter[A,R])(
  toMany: ResumableIter[E, EphemeralStream[A]]): ResumableIter[E, R]
```

The interesting part is the EphemeralStream of type A returned by the mapping Iteratee 'toMany', this allows any number of results for a single input of type E. If toMany or indeed dest returns EOF, so must the resulting Iteratee.

The following simple example shows how enumToMany can work:

```
def iTo(lower: Int, upper: Int): EphemeralStream[Int] =
  if (lower > upper) EphemeralStream.empty else EphemeralStream.cons(lower, iTo(lower + 1, upper))

val i = List(1,2,3,4).iterator
```

```
val (res, cont) = enumToMany(sum[Int])( mapTo( (i: Int) => El(iTo(1, i)) ) )(i).run
assertEquals(20, res)
assertTrue("should have been done", isDone(cont))
```

The toMany Iteratee here is mapTo called over the iTo function, for each input i it returns 1 -> i. The destination Iteratee sums the resulting stream, so the list 1 -> 4 then provides a sequence of 1, 1, 2, 1, 2 3, 1, 2, 3, 4 totalling 20.

In the above example sum and mapTo are simple IterV's that are, in the mapTo case ''restarted'' for each input of "i", and in sum's case is run until EOF is sent. This restarting with ResumableIter's allows the computation to be continued after intermediate results are returned, making them ideal for XML processing.

3.4.2 Async Pull with enumToMany

A simple streaming example can be seen below:

```
val url = scales.utils.resource(this, "/data/BaseXmlTest.xml")

val channel : DataChunker[DataChunk] = Channels.newChannel(url.openStream()).wrapped

val parser = AsyncParser()

val strout = new java.io.StringWriter()
val (closer, iter) = pushXmlIter( strout )

val enumeratee = enumToMany(iter)(parser.iteratee)
val ((out, thrown), cont) = enumeratee(channel).run

assertFalse( "shouldn't have thrown", thrown.isDefined)
assertTrue("should have been auto closed", closer.isClosed)
assertTrue("Channel itself should have been auto closed", channel.isClosed)
```

The .iteratee method calls (by default) the AsyncParser.parse function which wraps the AsyncParser into a ResumableIter[DataChunk, EphemeralStream[PullType]], where the stream itself is the result of calling AsyncParser.nextInput. It will return Done with a stream of events when enough data is processed. As can be seen from the type it fits perfectly with toMany parameter of enumToMany, for a given chunk it may return many PullTypes.

The pushXmlIter is a serializing IterV that pushes PullType into an outputstream. enumToMany then joins these two Iteratees to provide streaming.

The example also shows that the resources are all automatically closed upon completion. The parser and DataChunker resources themselves are CloseOnNeed instances and therefore also early closing (for example when there has been enough data processed).

Last update: May 10, 2024 12:48:10
Created: May 10, 2024 12:48:10

4. Accessing and Querying Data

4.1 XPath Embedded DSL

The XML XPath specifications allows navigation of XML documents via a DSL that describes routes through a document using a combination of axe, steps and predicates. It has a limited number of these abstractions but together they create a powerful direct, whilst remaining simple to use, querying language.

Scales provides this power via both a traditional string based approach and an embedded DSL that leverages the power of Scalas syntactical flexibility to mimic the XPath syntax.

The DSL uses the existing Scales abstractions to the full, and works via a zipper over the XmlTree itself. Each navigation step through the tree creates new zippers and new paths through the tree.

In every case possible (with the exception of the namespace:: axis) the range of behaviours closely follows the specification, like for like queries matching 100%. Instead of matching on prefixes Scales uses fully qualified expanded QNames (qualifiedName in the QName Functions) to match against, not requiring a prefix context within which to evaluate.

Internally, perhaps unsurprisingly, XPath is implemented as a combination of filter, map and flatMap. When retrieving results (e.g. converting to an Iterable) the results are sorted into Document order, this can be expensive for large result sets (see Unsorted Results for alternatives).

4.1.1 Simple Usage Examples

Given the following document:

```
val ns = Namespace("test:uri")
val nsa = Namespace("test:uri:attribs")
val nsp = nsa.prefixed("pre")

val builder =
  ns("Elem") /@ (nsa("pre", "attr1") -> "val1",
                 "attr2" -> "val2",
                 nsp("attr3") -> "val3") /(
    ns("Child"),
    "Mixed Content",
    ns("Child2") /( ns("Subchild") ~> "text" )
  )
```

we can easily query for the Subchild:

```
// top produces a Path from a Tree, in this case an XPath
val path = top(builder)

val res = path \* ns("Child2") \* ns("Subchild")
res.size // 1

string(res) // text
qname(res) // Subchild
```

4.1.2 XPath Crash Course

Scales Xml follows the XPath spec fairly closely and accordingly represents the concepts of context, location steps and axe, full details of which can be found in the XPath Standard.

The context, which can be thought of as current "place" in the document, is represented by the following:

- a notional node - where we are in the document - an element, a text node, an attribute etc.
- the position - index within a context and the size of this context

Location steps are a combination of axe, node test and predicates e.g. /*fred which represents the child axe, element node test and a predicate against a no-namespace local name of "fred".

As the XPath adds more axe, steps and predicates the context changes, reducing or expanding possible matches as it develops. Scales Xml's XPath DSL represents that context with the XPath class, where each operation on that class returns another immutable instance for the next context.

As with XPath, Scales Xml predicates, axe and node tests can be chained with the current context (the self axe in XPath) always represented by the resulting Scales XPath object. Only when the underlying results are used (for example by string or qname functions) do they leave the XPath object and get transformed into a, by default, ordered list of matching nodes.

4.1.3 XPath Axe

Scales supports the complete useful XPath axe, each of which can be used against a given context (an instance of Scales XPath), for the full XPath axe details find the spec here:

{|class="genTable" !XPath Axis !Scales DSL !Details |- |ancestor||ancestor_::||All the parents of this context |- |ancestor-or-self|| ancestor_or_self::||All the parents of this context and this node |- |attribute||*@||All the attributes for a given context, is often combined directly with a name |-

|child||\ or + to expand XmlItems||Children of this context. NB: \ alone in Scales DSL simply removes the initialNode setting required by \. If the children should be expanded (e.g. to use .filter directly) then + will "unpack" the child nodes. |- |descendant|| descendant_::||All children, and their children |- |descendant-or-self||descendant_or_self_::||This node and all descendants, also known as \ |-

|following||following_::||All nodes that follow this context in document order without child nodes of this context |- |following-sibling||following_sibling_::||All direct children of this contexts parent node that follow in document order. |- |parent||\^||The parent context of this context. For elements it represents the parent eleemnt and for attributes the containing element. |-

|preceding||preceding_::||All nodes that precede this context in document order excluding the parent nodes |- |preceding-sibling|| preceding_sibling_::||All previous children of the parent in the current context in document order. |- |self||The XPath object itself via '''.'''||The current context node within a document. |}

A commonly used abbreviation not listed above is of course \, which means descendant_or_self_::. The difference being that \ also supports possible eager evaluation and as per the spec the notion of [http://www.w3.org/TR/xpath20/#id-path-expressions \ in the beginning expression].

''NB Scales Embedded XPath DSL does not support the namespace axis - if you have a requirement for it then it can be looked at (please send an email to [mailto:scales-xml@googlegroups.com the mailing list] to discuss possible improvements)''

4.1.4 Node Tests

Scales embedded XPath DSL views the majority of node tests as predicates

{|class="genTable" !XPath Node Test !Scales DSL !Details |- |node()||.+||Returns a new context for all the children below a given context |- |text()||.text||Returns a new context for all the text and cdata below a given context |- |comment()||.comment||Returns a new context for all the comments below a given context |}

Scales XML also adds:

- .textOnly - filters out CData, just giving text nodes
- .cdata - provides CData nodes
- .pi - provides processing instructions

4.1.5 Predicates

There are three areas allowing for predicates within XPaths:

- Attributes
- Elements
- General

The first two are special cased, as in the XPath spec, as they are the most heavily used predicates (using the above example document):

```
// QName based match
val attributeNamePredicates = path \@ nsp("attr3")
string(attributeNamePredicates) // "val3"

// predicate based match
val attributePredicates = path \@ ( string(_) == "val3" )
qualifiedName(attributePredicates) // {test:uri:attribs}attr3

// Find child descendants that contain a Subchild
val elemsWithASubchild = path \\* ( _ \* ns("Subchild"))
string(elemsWithASubchild) // text
qualifiedName(elemsWithASubchild) // {test:uri}Child2
```

In each case the XmlPath (or AttributePath) is passed to the predicate with a number of shortcuts for the common QName based matches and positional matches for elements:

```
val second = path \*(2) // path \* 2 is also valid but doesn't read like \*[2]
qname(second) // Child2
```

The developer can chose to ignore namespaces (not recommended) by using the *:* and *:@ predicates instead (equivalent to string xpath /*[local-name() = "x"]).

Predicate Construction

All the predicates in Scales are built from two simple building blocks:

1. XmlPath => Boolean - via the XPath.filter function
2. AttributePath => Boolean - via the AttributeAxis.*@ function

The various base node types and filters are based on these functions, for example the element predicate * is implemented as:

```
def *(pred : XmlPath => Boolean) : XPath[T] =
  filter(x => x.isItem == false && pred(x))
```

In turn * can be seen as a combination of the \ child step and the * predicate (via xflatMap) and is provided as syntactic sugar.

Similarly text is implemented using filter.

All of the standard set of predicates (and axis combinations) can be found in the XPath ScalaDoc. Clicking the right arrow for many of the functions will lead you to the Definition Classes docs and their code.

Chaining Predicates

Predicates can be chained on the context itself, i.e. the XPath object, for example:

```
val pathsCombinedPredicates =
    root.\*(ns("Child")).
      *(_.\@( nsp("attr3") )) // context is still Child matches, but has additionally reduced it to only items with an attribute of attr3
```

This represents /root/*ns:Child[.@nsp:attr3] where the * Scales Xml element predicate allows matching on the self axis. The same chaining is available on the attribute axis represented by the [./doc/scales/xml/xpath/AttributePaths.html AttributePaths] class.

Positional Predicates

{|class="genTable" !XPath Position Function !Scales DSL !Details |- |position()||pos_<, pos_==, pos() and pos_>||Functions to work against the current position within a context |- |last()||last_<, last_== and last_>||Functions that work against the size of a given context |- |position() == last()||pos_eq_last||Take the last item in a context |}

These, more difficult to model, positional tests can be leveraged the same way as position() and last() can be in XPath.

So, for example:

```
// /*[position() = last()]
val theLast = path.\.pos_eq_last
qname(theLast) // Elem

// //*[position() = last()]
val allLasts = path.\\*.pos_eq_last
allLasts map(qname(_)) // List(Elem, Child2, Subchild)

// all elems with more than one child
// //*[ ./*[last() > 1]]
val moreThanOne = path.\\*( _.\*.last_>(1) )
qname(moreThanOne) // Elem

// all elems that aren't the first child
// //*[ position() > 1]
val notFirst = path.\\*.pos_>(1)
qname(notFirst) // Child2
```

Direct Filtering

The xflatMap, xmap, xfilter and filter methods allow extra predicate usage where the existing XPath 1.0 functions don't suffice.

The filter method accepts a simple XmlPath => Boolean, whereas the other varieties work on the matching sets themselves.

It is not recommended to use these functions for general use as they primarily exist for internal re-use.

4.1.6 Unsorted Results and Views

In order to meet XPath expected usage results are sorted in Document order and checked for duplicates. If this is not necessary - but speed of matching over a result set is (for example lazy querying over a large set) - then the raw functions (either raw or rawLazy) are good choices.

The viewed function however uses views as its default type and may help add further lazy evaluation. Whilst tests have shown lazy evaluation takes place its worth profiling your application to see if it actually impacts performance in an expected fashion.

See the XmlPaths trait for more information.

Last update: May 10, 2024 12:48:10
Created: May 10, 2024 12:48:10

4.2 XPath Functions

The XPath specs define a number of useful functions which are specific to XML, other string or number related functionality is provided by Scala itself and is not replicated by the XPath DSL.

4.2.1 Organisation

The XPath Functions are organised into three main type classes:

1. Names
2. TextValue
3. Boolean

Names provides a type class for any type that can represent a QName:

- Attributes
- Elems
- XmlTrees
- XPaths (name of the first node, "empty" otherwise)
- QNames

TextValue, representing everything that can produce a text value, has a similar base list:

- XmlTrees
- Attributes
- XmlItems
- XPaths (value of the first node)

In general if any logical combination thereof is possible they are also supported (for example XmlPaths themselves)

The boolean family of type class instances:

- XPaths (is it non-empty)
- Iterables (is it non-empty)
- String (length > 0)
- Number (value > 0) and of course
- Boolean

4.2.2 QName Functions

The full list of functions is available here.

Some examples:

```
val attr : Attribute = "attr" -> "value"

qname(attr) // attr
pqName(attr) // {}attr
namespaceUri(attr) // ""

val ns = Namespace("uri:namespace")
val pre = ns.prefixed("pre")

val prefixedAttr : Attribute = pre("prefixed") -> "prefixed value"

qname(prefixedAttr) // pre:prefixed
pqName(prefixedAttr) // pre:{uri:namespace}prefixed
namespaceUri(prefixedAttr) // uri:namespace

val elem = Elem(pre("prefixedElem"))
```

```
qname(elem) // pre:prefixedElem
pqName(elem) // pre:{uri:namespace}prefixedElem
namespaceUri(elem) // uri:namespace

val tree = elem /( elem ~> "\ndeep  value\n" )

pqName(tree) // pre:{uri:namespace}prefixedElem
```

Using the `name` function with a non QName XPath (e.g. an XmlItem) will result in an exception.

4.2.3 Text Functions

The full list of functions is available here.

Some examples (using the above definitions):

```
value(attr) // value
value(prefixedAttr) // prefixed value

// won't compile as there is no meaningful way to get an elems value
// value(elem)

value(tree.toTree) // \ndeep  value\n
normalizeSpace(tree) // deep value, but using type class directly
```

4.2.4 Boolean Function

```
boolean("") // false
boolean("value") // true

boolean(true) // true
```

Last update: May 10, 2024 12:48:10

Created: May 10, 2024 12:48:10

4.3 XPath 1.0 String Evaluation

The embedded XPath DSL provides a very flexible and performant way to query xml. However, being an embedded DSL, it has the negative of being wed to the compilation cycle. For those seeking a more direct XPath string parsing based experience Scales provides access via the Jaxen project.

The scales-jaxen project is a sub project and must be included separately as a library dependency.

4.3.1 How To Use

The following dependency must be used (instead of scales-xml):

```
"scales" %% "scales-jaxen" % "0.3" // or 0.4 for a Scalaz 6.0.4 dependency
```

After importing the package use the ScalesXPath constructor with either a map of prefix to uri, or a list of PrefixedNamespaces (similar to Elem construction):

```
import scales.xml.jaxen._

// prefix mappings are needed for context
val aPath = ScalesXPath("//*[1]", Map("pre" -> "urn:prefix"))
// the type annotation is for illustrative reasons only :->
val result : Iterable[Either[AttributePath, XmlPath]] = aPath.evaluate(anXmlPath)

// direct XPath results use get
val anotherPath = ScalesXPath("string(//*[1])", Map("pre" -> "urn:prefix"))
// This uses casts to retrieve values, it can throw.
val string = anotherPath.get[String](anXmlPath)
```

Results are always returned in Document order.

Rather than dealing with an Either for results, when the developer probably knows what type he wants, xmlPaths and attributePaths can be used:

```
val ns = Namespace("test:uri")
val nsa = Namespace("test:uri:attribs")
val nsp = nsa.prefixed("pre")

val builder =
  ns("Elem") /@ (nsa("pre", "attr1") -> "val1",
                 "attr2" -> "val2",
                 nsp("attr3") -> "val3") /(
    ns("Child"),
    "Mixed Content",
    ns("Child2") /( ns("Subchild") ~> "text" )
  )

val elems = ScalesXPath("//*")
elems.xmlPaths(path).map(qname(_)) // List(Elem, Child, Child2, Subchild)
elems.attributePaths(path).map(qname(_)) // List()

val attribs = ScalesXPath("//@*")
attribs.xmlPaths(path).map(qname(_)) // List()
attribs.attributePaths(path).map(qname(_)) // List(pre:attr3, attr2, pre:attr1)
```

As a shortcut to ignore namespaces (instead of using prefixes) use:

```
ScalesXPath("/html/body/p[2]/table[2]/tr/td/table/tr/td[1]/a/font").withNameConversion(ScalesXPath.localOnly)
```

The function passed to withNameConversion is of type:

```
QName => QName$
```

allowing further mappings as needed.

4.3.2 Other Jaxen Tricks

Jaxen provides various extensions to straight forward querying, including adding java extension functions. If possible, of course, use the embedded DSL, if not ScalesXPath is a Jaxen XPath implementation, allowing calls to setFunctionContext, variableContext etc.

Last update: May 10, 2024 12:48:10
Created: May 10, 2024 12:48:10

5. Serializing & Transforming XML

5.1 An Introduction to Serializing Xml With Scales

Scales Xml delegates as much serializing as possible to the DOML3 LSSerializer framework. There are a number of pain points, due to incompatibilities between jaxp implementations and jvm versions, that Scales smoothes out for the developer.

The main aims with Scales serialisation are:

- A unified interface
- Correctness ''especially with a view to encoding issues''

The unified interface is one main function:

```
def serialize[T: SerializeableXml](pout: XmlOutput)(it: T) : Option[Throwable]
```

All XML types that can be serialized provide a SerializeableXml type class instance and the user supplies the XmlOutput object. The users main interaction is through the SerializerData object allowing users to supply the output java.io.Writer and override both the encoding and xml version used.

Typically one of the following functions will be used:

- writeTo(it, out)
- it writeTo out
- asString(it)
- itemAsString(xmlItem)
- printTree(it)

The first two (writeTo) are the most general and the later three useful for debugging.

No SerializeableXml type class instances are defined for XmlPaths as they can be either an XmlItem or Tree. As such itemAsString exists for those times you really want to debug just the XmlItem.

In all cases the xml is written out using the serialize function, which always writes out the xml declaration with the ''pout : XmlOutput'' parameter. The more general writeTo approach will take the documents declaration unless specifically overridden.

5.1.1 writeTo & writeTo

The writeTo function:

```
def writeTo[T](it: T, output: Writer,
    version: Option[XmlVersion] = None, encoding: Option[Charset] = None)
    (implicit serializerFI: SerializerFactory, sxml: SerializeableXml[T])
    : Option[Throwable]
```

requires two parameters, the item to be serialized and the output Writer. When the remaining two parameters are None (or simply left as default) the encoding and xml version are taken from the items document.

To make life easier in the common case the WriteTo class (and implicits from ScalaXml._) allows a simpler:

```
val testXml = loadXml(...)
val str = asString(testXml)

val out = new java.io.StringWriter()
testXml writeTo out

assertEquals(str, out.toString)
```

5.1.2 What Can Be Serialized?

The following types can be serialized:

- XmlTree
- DslBuilder
- Doc
- Elem - an empty elem
- Iterator[PullType] - requires a full stream but works for all flavours of Pull

With the itemAsString allowing simple debug output.

If an object can be meaningfully written as Xml it is suggested to wrap streamSerializeable directly, converting your object into a stream as required.

Last update: May 10, 2024 12:48:10
Created: May 10, 2024 12:48:10

5.2 Folding Xml

Scales Xml provides a unique transformation option based on the premise of XmlPaths having document order. It is therefore possible to navigate between two paths. If we can do that then we can transform lists of paths within a same document by folding over them.

Once a given path is modified it effectively refers to a new Xml tree, the trick is then to move the zipper to the next paths relative position in the old document.

A number of transformations are available based on FoldOperation:

- AddAfter - ''add the nodes to the parent after the current path''
- AddBefore - ''add the nodes to the parent before the current path''
- AsIs - ''no-op''
- Remove - ''remove the current node''
- Replace - ''replace the current node''
- ReplaceWith - ''replace the current node with the results of another fold''

AsIs and ReplaceWith deserve explanation before examples. When performing transformations it is often useful to query or test against the resulting nodes, if the node should not be changed then AsIs() will make the fold a no-op.

ReplaceWith effectively allows nested folds which an important part of the composing transformations.

5.2.1 PathFoldR - Catchy Result Type

The PathFoldR type is a (for Xml):

```
XmlPath => Either[XmlPath, FoldError] // Either is the FoldR
```

Each foldPosition call is then resulting in either a new XmlPath or a reason as to why the transformation could not complete. Valid reasons are:

- NoPaths - ''you didn't find any nodes with the path''
- NoSingleRoot - ''if the input nodes don't share a single root path how can we join them''
- RemovedRoot - ''you can't return changed nodes if the root element was deleted''
- AddedBeforeOrAfterRoot - ''you can't add nodes around the root element''

5.2.2 Examples

The below examples will use the following base xml and definitions:

```
val ns = Namespace("test:uri")
val nsa = Namespace("test:uri:attribs")
val nsp = nsa.prefixed("pre")

val builder =
  ns("Elem") /@ (nsa("pre", "attr1") -> "val1",
                 "attr2" -> "val2",
                 nsp("attr3") -> "val3") /(
    ns("Child"),
    "Mixed Content",
    ns("Child2") /( ns("Subchild") ~> "text" )
  )
```

For a full set examples see the DslBuilderTests.scala.

Adding Children

The following example will add nodes around the existing nodes:

```
  val nodes = top(builder) \*

  nodes.map(qname(_)) // Child, Child2

  val res = foldPositions( nodes  ){
    case path if (!path.hasPreviousSibling) => AddBefore("start"l)
    case path if (!path.hasNextSibling) => AddAfter("end"l)
    // will throw a MatchError if no _ see AsIs
  }

  asString(res.left.get.tree)
```

Will return the following XML (formatting added for readability):

```
<?xml version="1.0" encoding="UTF-8"?>
<Elem xmlns="test:uri" xmlns:pre="test:uri:attribs" pre:attr3="val3" attr2="val2" pre:attr1="val1">
  <start xmlns=""/>
  <Child/>
  Mixed Content
  <Child2>
    <Subchild>text</Subchild>
  </Child2>
  <end xmlns=""/>
</Elem>
```

AsIs

In the [Adding_Children Adding Children] section we've left a possible match error, for example, if we choose not to act on the last child:

```
  // oops, surprising
  val res = foldPositions( nodes  ){
    case path if (!path.hasPreviousSibling) => AddBefore("start"l)
    // will throw a MatchError
  }
```

However, AsIs can be used to make sure we take normal actions if we have no match:

```
  val res = foldPositions( nodes  ){
    case path if (!path.hasPreviousSibling) => AddBefore("start"l)
    case _ => AsIs
  }

  asString(res.left.get.tree)
```

Giving:

```
<?xml version="1.0" encoding="UTF-8"?>
<Elem xmlns="test:uri" xmlns:pre="test:uri:attribs" pre:attr3="val3" attr2="val2" pre:attr1="val1">
  <start xmlns=""/>
  <Child/>
  Mixed Content
  <Child2>
    <Subchild>text</Subchild>
  </Child2>
</Elem>
```

Removing Children

In this example we'll remove the SubChild element:

```
  val nodes = top(builder) \\* ns("Subchild")

  nodes.map(qname(_)) // Subchild

  val res = foldPositions( nodes  ){
    _ => Remove()
  }

  asString(res.left.get.tree)
```

Giving:

```
<?xml version="1.0" encoding="UTF-8"?>
<Elem xmlns="test:uri" xmlns:pre="test:uri:attribs" pre:attr3="val3" attr2="val2" pre:attr1="val1">
  <start xmlns=""/>
  <Child/>
  Mixed Content
```

```
  <Child2/>
</Elem>
```

Replacing Children

This example changes the text in Subchild:

```
  val nodes = top(builder). \\*(ns("Subchild")). \+.text

  nodes.map(string(_)) // Subchild

  val res = foldPositions( nodes  ){
    _ => Replace("another value")
  }

  asString(res.left.get.tree)
```

yields:

```
<?xml version="1.0" encoding="UTF-8"?>
<Elem xmlns="test:uri" xmlns:pre="test:uri:attribs" pre:attr3="val3" attr2="val2" pre:attr1="val1">
  <start xmlns=""/>
  <Child/>
  Mixed Content
  <Child2>another value</Child2>
</Elem>
```

5.2.3 Composing Transformations

Transformations, like the rest of Scales, should also be composable. It is possible to chain transformations allowing some to fail if they can't find matches - NoPaths - (| - Try The Next) to find matches or stopping at the earliest failure (& - Fail Early).

In addition, they can be nested, performing transformations within transformations (ReplaceWith).

ReplaceWith - Nested

ReplaceWith aims to mimic the nesting of matching templates in xslt (via call-template) whereas using the pattern matcher directly more closely resembles apply-templates.

Using this replace as a basis:

```
  // for every child element add a text child that contains the qname of the elem
  def addTextNodes( op : XmlPath ) =
    foldPositions( op.\* ) {
      p => Replace( p.tree / qname(p) )
    }

  val allReplaced = addTextNodes( top(builder) )

  asString(allReplaced.left.get.tree)
```

yielding:

```
<?xml version="1.0" encoding="UTF-8"?>
<Elem xmlns="test:uri" xmlns:pre="test:uri:attribs" pre:attr3="val3" attr2="val2" pre:attr1="val1">
  <start xmlns=""/>
  <Child>Child</Child>
  Mixed Content
  <Child2>
    <Subchild>text</Subchild>
    Child2
  </Child2>
</Elem>
```

Now we can replace just the Subchild with:

```
  val nodes = top(builder). \\*(ns("Child2"))

  val res = foldPositions( nodes  ){
    _ => ReplaceWith(x => addTextNodes(top(x.tree)))
  }

  asString(res.left.get.tree)
```

yielding:

```
<?xml version="1.0" encoding="UTF-8"?>
<Elem xmlns="test:uri" xmlns:pre="test:uri:attribs" pre:attr3="val3" attr2="val2" pre:attr1="val1">
  <Child/>
  Mixed Content
  <Child2>
    <Subchild>textSubchild</Subchild>
  </Child2>
</Elem>
```

& - Fail Early

The "and" chained transformation will stop when it hits any failure.

Using the same base document as before:

```
  val wontFindAnyNodes = ( op : XmlPath ) =>
    foldPositions( op \* ns("NotAChild") ) {
      p => Replace( p.tree / qname(p) )
    }

  val willFindANode = ( op : XmlPath ) =>
    foldPositions( op \* ns("Child2") ) {
      p => Replace( p.tree / qname(p) )
    }

  val root = top(builder)

  val combined = wontFindAnyNodes & willFindANode

  val noPaths = combined( root ) // Will be Right(NoPaths)
```

whereas:

```
  val alsoFindsANode = ( op : XmlPath ) =>
    foldPositions( op \* ns("Child") ) {
      p => Replace( p.tree / qname(p) )
    }

  val andOk = willFindANode & alsoFindsANode

  val result = andOk( root )

  asString(result.left.get.tree)
```

yields:

```
<?xml version="1.0" encoding="UTF-8"?>
<Elem xmlns="test:uri" xmlns:pre="test:uri:attribs" pre:attr3="val3" attr2="val2" pre:attr1="val1">
  <Child>Child</Child>
  Mixed Content
  <Child2>
    <Subchild>text</Subchild>
    Child2
  </Child2>
</Elem>
```

| - Try The Next

The "or" chained transformation will try the next transformation if NoPaths is returned. This allows safe chaining always passing the result through until the first failing transformation.

Using the examples from & above:

```
  val orWorks = wontFindAnyNodes | willFindANode

  val orResult = orWorks( root )

  asString(orResult.left.get.tree)
```

yields:

```
<?xml version="1.0" encoding="UTF-8"?>
<Elem xmlns="test:uri" xmlns:pre="test:uri:attribs" pre:attr3="val3" attr2="val2" pre:attr1="val1">
  <Child/>
  Mixed Content
  <Child2>
```

```
    <Subchild>text</Subchild>
    Child2
  </Child2>
</Elem>
```

Last update: May 10, 2024 12:48:10

Created: May 10, 2024 12:48:10

5.3 TrAX & XSLT Support

JAXP's TrAX API provides both serialisation and transformation to the jvm. Scales leverages this api to allow xslt transformation and conversions to other DOMs. Additionally Scales works around known compatibility issues with the Sun JVM and the Xerces implementations (with a nod to the Saxon for its extra support).

Scales will attempt to, when the JVM/JAXP impl supports it, provide a StAXSource enabling transformation without intermediary objects. This too, unfortunately, doesn't always correctly work across JAXP implementations, some versions ignoring prolog content (some even NPEing when any is present).

As with most Scales JAXP integration the fall-back position of correctly serializing first is always available via the scales.traxSourceShouldSerialize property, when defined and true serialisation will occur before attempting a transformation. When its not defined a best effort based on the transformer class name is used.

The use of TrAX is generally simple enough not to require further wrapping, and Scales offers Folding for a more fitting transformation approach.

Developers wishing pretty printing can also use TrAX to achieve this (as with a DOM object).

5.3.1 Simple Usage Example

Roundtripping with trax:

```
  val elem = Elem("trax"l)
  val doc = Doc( elem / elem )

  import javax.xml.transform._
  val trax = TransformerFactory.
    newInstance.newTransformer

  val wr = new java.io.StringWriter()
  val str = new stream.StreamResult(wr)
  trax.transform(doc, str)
  println("source only " + wr.toString)

  val sr = ScalesResult()
  trax.transform(doc, sr)

   println("roundtrip " + asString(sr.doc))
```

yields:

```
source only <?xml version="1.0" encoding="UTF-8"?><trax><trax/></trax>
roundtrip <?xml version="1.0" encoding="UTF-16"?><trax><trax/></trax>
```

Note the utf-16 encoding, this was dictated by TrAX, and kept by Scales in the resulting Doc.

Last update: May 10, 2024 12:48:10

Created: May 10, 2024 12:48:10

6. Advanced

6.1 Technical Details

6.1.1 Scales Xml Optimisation Details

ImmutableArrayProxy

In order to reduce memory consumption Scales uses an abstraction over array like structures - ImmutableArrayProxy. Vector is appropriate for large structures (> 30 children) but inappropriate for smaller collections, taking many Mb of unnecessary memory usage.

The same is true of a simple immutable array, the costs are too high for items less than 4 in size.

As such Scales provides a One, Two and Three Seq, an ImmutableArray for less than 32 and a Vector wrapper for greater than 32. There is also ImmutableArrayAll, which reduces offset information to further reduce memory usage.

The builder itself is also optimised to allocate as little as possible and allow for re-use in the common case.

EitherLike

A simple concession to memory performance (and CPU performance) was the replacement of Either as a container and the creating of EitherLike. EitherLike has the similar properties but is a trait applied to classes, fold and projections are still present but the memory usage from the level of indirection is removed. Tests showed that 10-15% of the memory usage was held purely by the Either ADT, and performance was around 4-5% impact across the board due to the one level of indirection.

EitherLike is currently not used for pull parsing due to Scalas / JVMs understandable erasure limitation of inheriting twice with different type parameters.

TreeOptimisation

The TreeOptimisation trait provides a simple way to mix in optimisations on Tree, see the docs and source for example implementations (QNameTreeOptimisation and QNameElemTreeOptimisation).

The base Tree type is itself (as of 0.3) just an interface, allowing reduced memory usage in common cases (eg. repeated name value style elements) and a simple xml object conversion approach, where children may be lazily mapped.

The ParsingPerformance tests also demonstrate the possibility to optimise away the element itself if a given document often repeats element content.

When an optimised tree is modified (via copy) then it may still be further optimised or default to a normal Tree when no simple optimisation is possible.

QName and Elem Memory Usage

The QName and Elem structures are optimised to only keep references to items that are needed. NoNamespaceQName will only have a single data member reference to the local name and UnprefixedQName has no prefixed stored.

Elem is more interesting, given it has 4 relevant states, but takes the same approach. If the attributes are empty but there are namespaces upon creation then the resulting Elem (NoAttribsElem) will not contain the reference.

These simple optimisations positively affect memory usage considerably for large documents.

TreeProxies and Builders

The Scales XmlParser infrastructure attempts to cache builders at each level of the tree. If there are more than 3 children at any given element then the underlying builder (VectorBuilder / Pointer or ImmutableArray) cannot be re-used.

The vast majority of XML will typically end at the leafs with an elem that has only one logical text node, the builders generating these leaves can be re-used.

This level based builder caching and the heavily inlined ImmutableArrayProxy builders and TreeProxies together result in somewhat non idiomatic code but increases performance by 5-8%.

Last update: May 10, 2024 12:48:10
Created: May 10, 2024 12:48:10

6.1.2 Serialising - Nitty Gritty

While Serialising provides a high-level overview of serializing in Scales, this section gets into the specifics of how its implemented and compatibility issues.

Scales attempts the following:

1. Work around a number of bugs and behavioural differences with the various LSSerializer implementations
2. Stop as soon as possible with usable error messages - and give the user the tools to deal with that situation
3. Correctness checking with both XmlVersion and CData handling along with encoding and markup generation
4. Only perform serialisation encoding checks once
5. Correctly re-use Charset encodings
6. Allow plugable Serializers and sensible base implementations - if you don't like the way its working you can enhance it

Encoding

XML NAMES

When serializing XML there are two traditional problems with inter-operation between creator and reader:

- Characterset of stream
- Encoding of document

The xml parsers themselves *should* handle the former (i.e. via BOM and declarations etc), but the latter has more to do with the writing of the document itself.

XML 1.0 allows the use of a large enough range of unicode names that umlauts are valid but the encoding might not be:

```
<?xml version="1.0" encoding="US-ASCII"?>
<anrüchig/>
```

This document is itself not valid, as the ü in the markup cannot be represented by US-ASCII.

The default LSSerializerSerializerFactory provides an implementation of the encF function which abstracts away this issue. For a given character set it attempts to encode the string replying an Option[Throwable], with None indicating that it can be serialized with that encoding. Each serialized QName has its parts checked against the shared global cache, to reduce costs in creating encoders and their use.

A LSSerializerNoCacheFactory can be used if the caching behaviour is not desirable.

TEXT DATA

All normal character data serialized by the LSSerializer instances will be safe across encodings, as per the spec they will be escaped as character references. Which actual escaping used (hex or plain numeric) depends on the underlying JAXP implementation.

OTHER MARKUP CHARACTER DATA

CData, Comments and PI cannot have character references within their content. As such they behave similarly to the encoding of XML Names, with respective exceptions of CDataCannotBeEncoded, CommentCannotBeEncoded and PICannotBeEncoded.

CData deserves a special mention as many libraries (including the LS spec that Scales leverages) add splitting of cdata to allow extra character encodings. In part due to the incompatible/incomplete jaxp implementation approaches Scales takes a simple design decision - either the CData can be serialised or not.

Developers are free to customise this approach or indeed translate CData to Text wherever recipient systems will accept it.

Creating a SerializerFactory

It is advised to use the default factory wherever possible, however the LSSerializerBase and LSSerializer traits provides a useful extension point. The LSSerializerFactoryXHTML shows an example of how it can be extended.

The interface for SerializerFactory defines a single function:

```
trait SerializerFactory {
  def apply[R](thunk: Serializer => R)(data: SerializerData): R
}
```

The serialisation itself is performed in the context of thunk, where thunk accepts a Serializer. The serializer itself contains a very small interface, with only one quirk:

```
  path: List[QName]
```

this allows even the Serializer to make assumptions about the structure based on its XPath (the same structure to the pull parsing onQName function). In the case of element starting (emptyElement/startElement) the parameters are calculated by the serialize function itself. This means that whilst the Serializer instance is free to choose how to serializer it is freed from making decisions on what attributes or namespaces or default namespaces are correctly defined for that scope.

Overriding startElement or emptyElement would be the ideal place for applications requiring attributes be ordered.

Last update: May 10, 2024 12:48:10
Created: May 10, 2024 12:48:10

6.2 XML Equality

6.2.1 XML Equality Basics

The Scales Xml equality framework aims to help with both testing applications (similar to XmlUnit) that use Scales and also for runtime comparison activities, e.g. if an element has this attribute with this value do X.

The Scales equality framework does not throw, but returns information allowing decisions to be made. In the case of `===` a simple boolean is returned, in the case of `compare` a full ADT and path can be returned diagnosing the first difference found.

The `compare` function is used by the Equal instances and is documented here.

NB Scala 2.8.x support for the Equality framework is experimental. Importing FromEqualsImplicit._ enables the use of === from Scalaz. Unfortunately due to 2.8.x compiler issues the implicit resolution does not correctly function and may cause compiler crashes.

How To Use

Scales Xml Equality leverages two type classes, XmlComparison and the Scalaz Equal typeclass to provide comparison via a simple === . As such Scalaz must be imported (best after Scales imports to avoid Tree import issues):

```
import scalaz._
import Scalaz._
```

Testing equality is therefore as simple as:

```
val t1 = Text("fred")
val t2 = Text("fred")

assertTrue("t1 == t2", t1 == t2) // equals
assertTrue("t1 === t2", t1 === t2) // Scalaz Equal type class

assertTrue("t1 compare t2",
  compare(Nil, t1, t2).isEmpty ) // XmlComparison type class
```

Wherever an XmlComparison exists an Equal instance can be created. The results of a compare include both a path to the difference and a fully pattern matchable XML difference ADT.

Types Covered

The full set of the Scales Xml Model is covered by the equality framework:

- QName
- Attribute
- Attributes
- XmlItem
- Elem
- XmlTree
- XmlPath
- Anything that can be seen as Iterator[PullType]
- Doc and various DocLike implementations

QNames by default do not compare with the prefix(unlike Canonical Xml, where string comparisons including prefixes are expected), only the namespace (as per =:=). This implies that documents created by different systems using different prefixes are still comparable, a different implicit default Equal[QName] instance can change that behaviour.

XmlTrees/XmlPath's etc are converted to Iterator[PullType] in order to compare. No attempt to match DTDs or encoding are made, but the rest of a given document (Doc and DocLike implementations) will be.

Within the comparison framework the comparison for all the types are combined, the QName Equal typeclass is used throughout, including for the Attribute comparison, which is used in turn by the Elem - which is finally used by Stream comparisons.

This lookup is performed implicitly, allowing for individual parts to be swapped out, if the developer wants prefixes to be tested. Either use name based overriding in the relevant scope or mix the traits differently to provide custom behaviour (and not import ScalesXml._)

Note The three different kinds of QNames each have a different type and, as such, using === to compare different types will not work. Using compare, however, will:

```
val ns = Namespace("uri:prefixed")

val p = ns.prefixed("p")

val nonPrefixedQName = ns("a1")
// prefixed
val prefixedQName = p("a1")

// both of the above are semantically the same
assertTrue("compare(nonPrefixedQName, prefixedQName).isEmpty", compare(nonPrefixedQName, prefixedQName).isEmpty)
```

Why Join Adjacent Text and CData?

Scales Xml equality makes three default design decisions, prefixes aren't generally relevant only the namespace is (unless you tell it to use QName Token comparison) and to join adjacent CData and Text.

The reason for joining adjacent CData and Text nodes is to simplify the comparison of text. CData can always be written as Text nodes, and a parser is free to "split" a single logical Text node into multiple smaller text nodes. Scales neither forces joining of the text nodes at parse time nor when adding child nodes, as such to usefully compare they must be joined.

This also allows testing content from different sources without issue.

Removing Comments And PIs

The default comparison logic treats both Comments and PIs as relevant for comparison. This design choice meets expectations for the majority of XML documents.

In the event that Comments and PIs should not be compared the implicits can be overridden in scope with:

```
val root = po("root")
val child = po("child")

import LogicalFilters._

implicit def toDefaultStreamComparison[T](
         implicit tv : T => StreamComparable[T],
         ic : XmlComparison[XmlItem],
         ec : XmlComparison[Elem],
         qe : Equal[QName], , qe : Equal[QName],
         qnameTokenComparison : Option[(ComparisonContext, String, String) => Boolean]) : XmlComparison[T] =
    new StreamComparisonWrapper(
      new StreamComparison(
        x => removePIAndComments(joinText(x))
      )( ic, ec, qe, qnameTokenComparison)
    )

val x1 =
    <(root) /( "0", "1", CData("2"), Comment("c2"), "3", "4", PI("i","s"),
      child /(
        "s1", CData("s2"), Comment("cs2"), "s3"
      ),
      child /(
        CData("s22"), Comment("cs22"), PI("i","s"), "s23"
      ),
      PI("i","s"), "5", CData("6"), Comment("c6") )


val x2 =
    <(root) /( "0","1",CData("2"), "3","4",
      child /(
          "s1", CData("s2"), "s3"
      ),
      child /(
          CData("s22"), "s23"
      ),
      "5", CData("6") )
```

```
assertTrue( " x1 === x2 ", x1 === x2)
```

Why Not Use Canonical XML?

Testing Xml Equality is not always straightforward, a standard approach however exists : Canonical Xml - a defined w3c standard approach to serialization. Canonical Xml treats QName prefixes themselves as relevant, if an XML processor changes a prefix, that document is no longer comparable under Canonical Xml (No Namespace Prefix Rewriting).

Whilst the justifications for the prefix rewriting rule in Canonical Xml is, within the context of embedded XPaths or XML QNames (their prefixes only make sense within that document), understandable Scales takes the position that this is far rarer an occasion than simple Xml as a data transport usage. However, as with the rest of Scales, this default too is customisable.

The problem, and it stops the Canonical Xml reasoning as well, is that a producing application can re-write the prefixes for embedded QNames or XPaths before sending. WSDM applications often meet this (see Apache Muse for examples of this). Scales is of course, by default, not aware of such usage - see here for details on how to configure Scales to be QName token aware. In short, it can be as simple as declaring this in the correct scope:

```
implicit val defaultQNameTokenComparison = Option(qnamesEqual _)
```

Canonical Xml also forces redundant namespace declarations to be removed (Superfluous Namespace Declarations). Scales typically only uses namespace declarations for predictable document processing - i.e. loading and saving should be 1:1 in usage - however it also can if necessary leverage declarations for QName Token handling in both Attribute values and Text/CData nodes.

Similarly the following approaches to Canonical Xml actually may break data assumptions of equality:

- normalize line feeds
- normalize attribute values
- Default attributes are added to each element

The latter may cause issues with certain receiving processors and depends on a validation / schema enrichment working. Scales concerns itself with the documents actually being compared.

In short - Scales offers a typed and more flexible approach to equality than Canonical Xml handling.

Last update: May 10, 2024 12:48:10
Created: May 10, 2024 12:48:10

6.2.2 XmlComparison - What, Where & How Was It Different

The XmlComparison typeclass is fairly simple with one function:

```
def compare( calculate : Boolean , context : ComparisonContext, left : T, right : T) : Option[(XmlDifference[_], ComparisonContext)]
```

All instances are created via defs and implicit lookup. As such, where runtime performance of comparison is a concern you should cache an appropriately scoped instance via `implicitly`.

The reason for using defs is that the behaviour of any one XmlComparison instance is configurable. Substituting the `defaultQNameTokenComparison` in a given scope will affect both XmlItem and Attribute value comparison.

To retrieve the full information about what changed use the compare function (mixed into the scales.xml package):

```
def compare[T : XmlComparison]( left : T, right : T) : Option[(XmlDifference[_], ComparisonContext)] =
  implicitly[XmlComparison[T]].compare(true, ComparisonContext(), left, right)
```

As can be seen, this simply provides a starting context and informs the framework to calculate both the path to a difference and a detailed ADT for what was different (XmlDifference).

The compare Function

Any implementing instance of XmlComparison must provide a compare implementation, for example a user provided XmlItem comparison, it is recommended to respect the following conventions:

1. When calculate is false, its a signal that no detailed result is expected and is likely called from ===
2. Use the comparison context to handle namespace lookups
3. Respect defaultQNameTokenComparison, if code has defined it in scope you must assume the developer wants at least the context to be generated (irrespective of calculate being false or not).

THE CALCULATE PARAMETER

The calculate parameter when false indicates that any difference returned as scales.xml.equals.SomeDifference.noCalculation, i.e. a dummy Some value. The purpose of the false parameter is for a simple comparison via === and the derived Equal type class.

A value of false will also disable, unless a defaultQNameTokenComparison is defined in scope, the generation of relevant ComparisonContexts, further reducing allocation for simple equality checks.

However when set true it instructs both the use of ComparisonContexts and that the return value is as detailed as possible.

COMPARISONCONTEXT

ComparisonContext is a stack of potential namespace contexts (for both the left and right side), the parent context and the BasicPath from the start of the compare. BasicPath is defined as:

```
// {ns}Local -> count
type BasicPathA = (QName, Map[String, Int])
type BasicPath = List[BasicPathA]
```

and maintains a count against each QName as it navigates down a given tree. ComparisonContext provides a simplified string path via the pathString function, for example the following output:

```
/{}root[1]/{uri:prefixed}elem[2]/{uri:prefixed}elem[1]
```

In addition to identifying where something is different it could be used to decide if the difference is relevant within a custom XmlComparison instance.

NamespaceContext is used by both XmlComparison and by the serialisation mechanisms and acts a stack of prefix -> namespace mappings that have been defined in the given trees. The namespace prefixes used by PrefixedQName attributes and element are combined with any defined prefixes from the elements namespace map.

When comparing a stream the position within that iterator is also set upon returning from comparison - streamPosition. Calling a ComparisonContexts toDifferenceAsStream with one of the compared xml objects will provide a Stream[PullType] with the xml from the start of the document to the diffence. Callers are responsible for ensuring the input is restartable (i.e. if it was over an http stream that a bufferred stream was used) and that the conversion function matches (by default it follows the same convention of joinTextAndCData that the stream comparison uses). For example:

```
val x = loadXml(scales.utils.resource(this, "/data/Nested.xml")).rootElem

// create a difference
val y = x.fold_!( _.\*.\*("urn:default"::"ShouldRedeclare") )(_ => Remove())

val Some((diff, context)) = compare[XmlTree](x, y)

// the stream from the start of the document to the difference
val upTo = context.toDifferenceAsStream(x)

// as a string
val upToStr = asString(upTo.iterator)

assertEquals(232, upToStr.size)
```

RETURN VALUE

The return value indicates:

1. The presence of a difference (Some vs None)
2. The difference itself (XmlDifference), and
3. The relevant context for this difference (allowing access to namespaces and path)

If the calculate parameter is false the default implementations return scales.xml.equals.SomeDifference.noCalculation. Custom XmlComparison instances my choose to return other values but it is not recommended.

XmlDifference

The XmlDifference ADT provides information about the type of difference and provides the objects themselves that contained the difference. The XmlComparison framework attempts to provide, when using calculate true, finely detailed information about what was different via case classes, allowing simplified pattern matching to analyse differences.

The full ADT is present via the [./doc/scales/xml/equals/XmlDifference.html scala docs, Known Subclasses] or directly via [./api.sxr/scales/xml/equals/XmlDifference.scala.html the source]. To aid explanation the following are presented:

- DifferentTypes(left : PullType, right : PullType)
- AttributeValueDifference(left : Attribute, right : Attribute)
- DifferentNumberOfMiscs(left : Miscs, right : Miscs, isProlog : Boolean)
- ElemAttributeDifference(left : Elem, right : Elem, attributesDifference : AttributesDifference)

DifferentTypes is returned when a given left PullType is of different type to the right PullType. AttributeValueDifference indicates the names are the same but the values differ between the two attributes.

DifferentNumberOfMiscs indicates that the prolog (isProlog == true) or end miscallaeneous (isProlog == false) have a different count and provides the Miscs themselves for further investigation. ElemAttributeDifference contains the elements that contained an AttributesDifference, which in turn has a number of possible matching types (DifferentNumberOfAttributes, DifferentValueAttributes and MissingAttributes).

QName Token Handling

QName Tokens are attributes or Text nodes that contain a prefixed qname such as:

```
<elem xmlns:pre="uri:test" value="pre:alocal">pre:local</elem>
```

Applications are free to redefine the prefix used between runs, making things difficult for comparison. Scales provides two simple mechanism to help solve this problem:

1. ComparisonContext
2. defaultQNameTokenComparison

The former is covered above and provides the NamespaceContext's for the left and right side objects under comparison. This information is calculated either when the defaultQNameTokenComparison is defined or calculate is true.

The defaultQNameTokenComparison is defined as:

```
implicit val defaultQNameTokenComparison : Option[(ComparisonContext, String, String) => Boolean] = None
```

Simply defining this function will also enable the creation of the contexts (irrespective of the value of calculate). The parameters are simply the ComparisonContext together with the left and right object's string values.

As such a simple way to enable qname comparisons is to define and override, within an appropriate scope:

```
implicit val defaultQNameTokenComparison = Option(qnamesEqual _)
```

qnamesEqual is defined as:

```
def qnamesEqual(context : ComparisonContext, str : String, str2 : String) = {
  // split both, if one has and the other not, then its false anyway
  val sp1 = str.split(":")
  val sp2 = str2.split(":")
  if (sp1.size == 2 && sp2.size == 2) {
    sp1(1) == sp2(1) && { // values match
      // look up prefixes
      (for{ lnc <- context.leftNamespaceContext
           rnc <- context.rightNamespaceContext
           lns <- lnc.mappings.get(sp1(0))
           rns <- rnc.mappings.get(sp2(0))
         } yield lns == rns
      ).
      getOrElse(false)
    }
  } else
    str == str2
}
```

Which should show an obvious limitation, it can only work when there is a single qname in the text. This would stop it being able to handle embedded XPaths for example.

As such the functionality is exposed to the library users to customise. Any useful other implementations are more than welcome as contributions :-)

Last update: May 10, 2024 12:48:10

Created: May 10, 2024 12:48:10